Computer Sciences Department

Reducing Concurrent Analysis Under a Context Bound to Sequential Analysis

Akash Lal
Thomas Reps

Technical Report #1629

January 2008 (Revised February 2008)

Reducing Concurrent Analysis Under a Context Bound to Sequential Analysis

Akash Lal and Thomas Reps

University of Wisconsin; Madison, WI; USA. {akash, reps}@cs.wisc.edu

Abstract. This paper addresses the analysis of concurrent programs with shared memory. Such an analysis is undecidable in the presence of multiple procedures. One approach used in recent work obtains decidability by providing only a partial guarantee of correctness: the approach bounds the number of context switches allowed in the concurrent program, and aims to prove safety, or find bugs, under the given bound. In this paper, we show how to obtain simple and efficient algorithms for the analysis of concurrent programs with a context bound. We give a general reduction from a *concurrent* program P, and a given context bound K, to a slightly larger *sequential* program P_s^K such that the analysis of P_s^K can be used to prove properties about P. The reduction introduces symbolic constants and `assume` statements in P_s^K. Thus, any sequential analysis that can deal with these two additions can be extended to handle concurrent programs as well, under the context bound. We give instances of the reduction for common program models used in model checking, such as Boolean programs, pushdown systems (PDSs), and symbolic PDSs.

1 Introduction

The analysis of concurrent programs is a challenging problem. While, in general, the analysis of both concurrent and sequential programs is undecidable, what makes concurrency hard is the fact that even for simple program models, the presence of concurrency makes their analysis computationally very expensive. When the model of each thread is a finite-state automaton, the analysis of such systems is PSPACE-complete; when the model is a pushdown system, the analysis becomes undecidable [20]. This is unfortunate because it does not allow the advancements made on such models in the sequential setting, i.e., when the program has only one thread, to be applied in the presence of concurrency.

A high-level goal of our work in recent years has been to automatically extend analyses for sequential programs to analyses for concurrent programs. This paper addresses this problem under a bound on the number of context switches.[1] We refer to analysis of concurrent programs under a context bound as *context-bounded analysis* (CBA). Previous work has shown the value of CBA: KISS [19], a model checker for CBA with a fixed context bound of 2, found numerous bugs

[1] A context switch is said to occur when execution control is passed from one thread to another.

in device drivers; a study with explicit-state model checkers [15] found more bugs with slightly higher context bounds. It also showed that the state-space covered with each increment to the context-bound decreases as the context bound increases. Thus, even a small context bound is sufficient to cover many program behaviors, and proving safety under a context bound should provide confidence towards the reliability of the program. Unlike the above-mentioned work, this paper addresses CBA with any given context bound and with different program abstractions (for which explicit-state model checkers would not terminate).

The decidability of CBA, when each program thread is abstracted as a *pushdown system* (PDS)—which serve as a general model for recursive programs with finite-state data—was shown in [18]. These results were extended to PDSs with bounded heaps in [3] and to weighted PDSs (WPDSs) in [11]. All of this work required devising new algorithms. Moreover, each of the algorithms have certain disadvantages that make them impractical to implement.

To explain the disadvantages, consider the analysis of program models with finite-state data, such as Boolean programs [1].[2] For such programs, the number of valuations of variables is exponential in the number of variables. For practical implementations, the use of symbolic techniques, such as BDDs, are crucial for scalability. Similar models also arise in other work [24, 6] and BDDs are used there as well.

With the algorithms of [18, 3], it is not clear if symbolic techniques can be applied. Those algorithms require the enumeration of all reachable states of the shared memory at a context switch. This can potentially be very expensive. However, those algorithms have the nice property that they only consider those states that actually arise during valid (abstract) executions of the model. (We call this *lazy* exploration of the state space.)

Our recent paper [11] showed how to extend the algorithm of [18] to use symbolic techniques. However, the disadvantage there is that it requires computing auxiliary information for computing the reachable state space. (We call this *eager* exploration of the state space.) The auxiliary information summarizes the effect of executing a thread from any control location to any other control location. Such summarizations may consider many more program behaviors than can actually occur (whence the term "eager").

This problem can also be illustrated by considering interprocedural analysis of sequential programs: for a procedure, it is possible to construct a summary for the procedure that describes the effect of executing it for any possible inputs to the procedure (eager computation of the summary). It is also possible to construct the summary lazily (also called partial transfer functions [14]) by only describing the effect of executing the procedure for input states under which it is called during the analysis of the program. The former (eager) approach has been successfully applied to Boolean programs [1, 23], but the latter (lazy) approach is often desirable in the presence of more complex abstractions, especially those

[2] Boolean programs can be understood as imperative programs with only the Boolean datatype.

that contain pointers (based on the intuition that only a few aliasing scenarios occur during abstract execution).

Contributions. This paper makes two main contributions. First, we give simpler decidability results for CBA by reducing a concurrent program to a sequential program that simulates all its executions for a given number of context switches. Hence, new algorithms do not have to be developed; instead, algorithms for analyzing sequential programs can be used for CBA. In fact, for PDSs, we obtain better asymptotic complexity than previous algorithms, just by using the standard PDS algorithms (§4): we obtain algorithms that scale linearly with respect to the size of the local state space; moreover, we show how to obtain algorithms that scale linearly with the number of threads (whereas previous algorithms scaled exponentially).

Our reduction introduces symbolic constants and `assume` statements. Thus, any sequential analysis that can deal with these two additions can be extended to handle concurrent programs as well (under a context bound).

These additions are only associated with the shared data in the program. This implies that when only a finite amount of data is shared between threads of a program (e.g., there are only a finite number of locks), *any* sequential analysis, even of programs with pointers or integers, can be extended to perform CBA of concurrent programs.

When the shared data is not finite, our reduction still applies. Numeric analysis, like [5], can be extended for CBA of concurrent programs. This shows the benefit of our reduction towards obtaining new algorithms for CBA.

Previous results [3, 18] can be obtained as a direct instance of our reduction without requiring new algorithms. We give instances of our reduction for Boolean programs, PDSs, and symbolic PDSs. The former shows that the use of PDS-based technology, which seemed crucial in previous work, is not necessary: the standard interprocedural algorithms [21, 25, 8] can also be used for CBA.

Second, we show how to obtain a symbolic and lazy algorithm for CBA on Boolean programs (§5). This combines the best of previous algorithms: the algorithms of [18, 3] are lazy but not symbolic, and the algorithm of [11] is symbolic but not lazy.

The rest of the paper is organized as follows: §2 gives an overview of our reduction from concurrent to sequential programs; §3 specializes the reduction to Boolean programs; §4 specializes the reduction to PDSs and symbolic PDSs; §5 gives a lazy symbolic algorithms for CBA on Boolean programs; §6 shows some early results with our algorithms; §7 discusses related work. Proofs can be found in App. A.

2 An Overview of the Reduction

This section gives an overview of how we perform the reduction from concurrent programs to sequential programs under the context bound. This reduction transforms the non-determinism in control, which arises because of concurrency,

to non-determinism on data. (The motivation is that the latter problem is understood much better than the former one.)

The execution of a concurrent program proceeds in a sequence of *execution contexts*, defined as the time between consecutive context switches during which only a single thread has control. In this paper, we do not consider dynamic creation of threads, and assume that a concurrent program is given as a fixed set of threads, with one thread identified as the starting thread.

Suppose that a program has two threads, T_1 and T_2, and that the context bound is $2K-1$. Then any execution of the program under this bound will have up to $2K$ execution contexts, with control alternating between the two threads, informally written as $T_1; T_2; T_1, \cdots$. Each thread has control for at most K execution contexts. Consider three consecutive execution contexts $T_1; T_2; T_1$. When T_1 finishes executing the first of these, it gets swapped out and its local state, say l_1, is stored. Then T_2 gets to run, and when it is swapped out, T_1 has to resume execution from l_1 (along with the global store, which T_2 might have changed).

The requirement of resuming from the same local state is one difficulty that makes analysis of concurrent programs hard—during the analysis of T_2, the local state of T_1 has to be remembered (even though it is unchanging). This forces one to consider the cross product of the local states of the threads, causing exponential blowup when the local state space is finite, and undecidability when the local state includes a stack. The advantage of introducing a context bound is that such a cross-product need not be considered. The algorithms of [18, 11, 3] scale polynomially with the size $|L|$ of the local state space: the algorithms of [18, 3] are $\mathcal{O}(|L|^5)$, and [11] is $\mathcal{O}(|L|^K)$. Our algorithm, for PDSs and Boolean programs, is $\mathcal{O}(|L|)$. (Strictly speaking, in each of these, $|L|$ is the size of the local transition system.)

Previous approaches still have some flavor of being designed to remember the local state. For analysis of programs with multiple procedures, the local state consists of the program stack. To be able to remember a stack (or, more generally, a set of stacks), previous work made use of PDS machinery. Our reduction shows that PDS machinery is not essential, and that more standard interprocedural analysis techniques [25, 8, 21], which do not manipulate the program stack, can also be used for CBA.

Our key observation is the following: for analyzing the three execution contexts $T_1; T_2; T_1$, we modify the threads so that we only have to analyze $T_1; T_1; T_2$, which eliminates the requirement of having to drag along the local state of T_1 during the analysis of T_2. For this, we *assume* the effect that T_2 might have on the shared memory, apply it while T_1 is executing, and then *check* our assumption after analyzing T_2.

Consider the general case when each of the two threads have K execution contexts. We refer to the state of shared memory as the *global state*. First, we guess $K-1$ (arbitrary) global states, say $s_1, s_2, \cdots, s_{K-1}$. We run T_1, so that it starts executing from the initial state s_0 of the shared memory. At a non-deterministically chosen time, we record the current global state s'_1, change it to

s_1, and resume execution of T_1. Again, at a non-deterministically chosen time, we record the current global state s'_2, change it to s_2, and resume execution of T_1. This continues $K-1$ times. Implicitly, this implies that we assumed that the execution of T_2 will change the global state from s'_i to s_i in its i^{th} execution context. Next, we repeat this for T_2: we start executing T_2 from s'_1. At a non-deterministically chosen time, we record the global state s''_1, we change it to s'_2 and repeat $K-1$ times. Finally, we verify our assumption: we check that $s''_i = s_{i+1}$ for all i between 1 and $K-1$. If these checks pass, we have the guarantee that VAR_G^K can have a valuation s if and only if the concurrent program can have the global state s in K execution contexts per thread.

The fact that we do not alternate between T_1 and T_2 implies the linear scalability with respect to the local state. Because the above process has to be repeated for all valid guesses, our approach scales as $\mathcal{O}(|G|^K)$, where G is the global state space, which is still polynomial because K is fixed. In general, the exponential complexity with respect to K may not be avoidable because the problem is NP-complete when the input has K written in unary [10]. However, symbolic techniques can be used for a practical implementation.

Instead of designing a new algorithm for the above assume-guarantee process, we show how to reduce it to the problem of analyzing a single thread. We add more variables to the program, initialized with symbolic constants, to represent our guesses. The switch from one global state to another is made by switching the set of variables being accessed by the program. We verify the guesses by inserting `assume` statements at the end.

The reduction. Consider a concurrent program P with two threads T_1 and T_2 that only has scalar variables (i.e., no pointers, arrays, or heap).[3] We assume that the threads share their global variables, i.e., they have the same set of global variables. Let VAR_G be the set of global variables of P. Let $2K-1$ be the bound on the number of context switches.

The result of our reduction will be a sequential program P^s. It has three parts, performed in sequence: the first part T_1^s is a reduction of T_1; the second part T_2^s is a reduction of T_2; and the third part, `Checker`, consists of multiple `assume` statements to verify that a correct interleaving was performed. Let L_i be a label preceding the i^{th} part. Thus, P_s has the form shown in the first column of Fig. 1.

P^s has K copies of VAR_G as its set of global variables. If $\text{VAR}_G = \{\mathtt{x}_1, \cdots, \mathtt{x}_n\}$, then let $\text{VAR}_G^i = \{\mathtt{x}_1^i, \cdots, \mathtt{x}_n^i\}$. The initial values of VAR_G^i are used to store the i^{th} guess s_i. It has an additional global variable `k`, which will take values between 1 and $K+1$. It tracks the current execution context of a thread: at any time P^s can only read and write to variables in $\text{VAR}_G^{\mathtt{k}}$. The local variables of T_1^s are the same as those of T_1, and similarly for T_2^s.

Let $\tau(\mathtt{x}, i) = \mathtt{x}^i$. If `st` is a program statement in P, let $\tau(\mathtt{st}, i)$ be the statement in which each global variable `x` is replaced with $\tau(\mathtt{x}, i)$, and the local vari-

[3] Such program models are often used in model checking, as well as for numerical programs.

Program P^s	$\texttt{st} \in T_i$	Checker

$L_1 : T_1^s$;
$L_2 : T_2^s$;
L_3 : Checker
	if k = 1 **then**
	  $\tau(\texttt{st}, 1)$;
	else if k = 2 **then**
	  $\tau(\texttt{st}, 2)$;
	  ...
	else if k = K **then**
	  $\tau(\texttt{st}, K)$;
	end if
	if k $\leq K$ and $*$ **then**
	  k ++
	end if
	if k = $K+1$ **then**
	  k = 1
	  goto L_{i+1}
	end if
		for $i = 1$ to $K-1$ **do**
		  for $j = 1$ to n **do**
		  assume $(\texttt{x}_j^i = v_j^{i+1})$
		  end for
		end for

Fig. 1. The reduction for general concurrent programs under a context bound $2K-1$. In the reduction, $*$ stands for a nondeterministic Boolean value. Furthermore, the variables VAR_G^1 are initialized to the same values as those in VAR_G for P, the rest of the variables $\texttt{x}_j^i$ are initialized to the symbolic value v_j^i, and k is initialized to 1.

ables remain unchanged. The reduction constructs T_i^s from T_i by replacing each statement st by what is shown in the second column of Fig. 1. The third column shows Checker.

The fact that local variables are not replicated captures the constraint that a thread starts executing from the local state it was in when it was swapped out at a context switch.

The Checker enforces a correct interleaving of the threads. It checks that the values of global variables when T_1 starts its $(i+1)^{\text{th}}$ execution context are the same as the values produced by T_2 when T_2 finished executing its i^{th} execution context. (Because the execution of T_2^s happens after T_1^s, each execution context of T_2^s is guaranteed to use the global state produced by the corresponding execution context of T_1^s.)

The reduction ensures the following property: when P_s finishes execution, the variables VAR_G^K can have a valuation s if and only if the variables VAR in P can have the same valuation after $2K-1$ context switches.

Symbolic constants. One way to deal with symbolic constants is to consider all possible values for them (eager computation). We show instances of this strategy for Boolean programs (§3) and for PDSs (§4). Another way is to lazily consider the set of values they may actually take on during the (abstract) execution of the concurrent program, i.e., only consider those values that pass the Checker. We show an instance of this strategy for Boolean programs (§5).

Multiple threads. If there are multiple threads, say n, then a precise reasoning for K context switches would require one to consider all possible thread

schedulings, e.g., $(T_1;T_2;T_1;T_3)$, $(T_1;T_3;T_2;T_3)$, etc. There are $\mathcal{O}((n-1)^K)$ such schedulings. Previous analyses [18, 11, 3] enumerate explicitly all these schedulings, and thus have $\mathcal{O}((n-1)^K)$ complexity even in the best case. We avoid this exponential factor as follows: we only consider the round-robin thread schedule $T_1;T_2;\cdots T_n;T_1;T_2;\cdots$ for CBA. Because a thread is allowed to not perform any steps during its execution context, CBA still considers other schedules. For example, when $n=3$, the schedule $T_1;T_2;T_1;T_3$ will be considered by CBA only when $K=5$ (in the round-robin schedule, T_3 does nothing in its first execution context, and T_2 does nothing in its second execution context).

Setting the bound on the length of the round-robin schedule to nK allows our analysis to consider all thread schedulings with K context switches (as well as some schedulings with more than K context switches). Such a schedule has K execution contexts per thread. The reduction for multiple threads proceeds in a similar way to the reduction for two threads. The global variables are copied K times. Each thread T_i is transformed to T_i^s, as shown in Fig. 1, and P_s calls the T_i^s in sequence followed by the `Checker`. `Checker` remains the same (it only has to check that the state after the execution of T_n^s agrees with the symbolic constants).

The advantages of this approach are as follows: (i) we avoid an explicit enumeration of $\mathcal{O}((n-1)^K)$ thread schedules, thus, allowing our analysis to be more efficient in the common case; (ii) we explore more of the program behavior with a round-robin bound of nK than with a context-switch bound of K with all schedules; and (iii) the cost of analyzing the round-robin schedule of length nK is about the same (in fact, better) than what previous analyses take for exploring one schedule with a context bound of K (see §4). These advantages allow our analysis to scale much better in the presence of multiple threads than previous analyses.

In the rest of the paper, we only consider two threads because the extension to multiple threads is straightforward for round-robin scheduling.

Applicability of the reduction to different analyses. Certain analysis, like affine-relation analysis (ARA) over integers, as developed in previous papers [12, 13], cannot make use of this reduction. The presence of `assume` statements makes the ARA problem undecidable. However, any abstraction prepared to deal with branching conditions can also handle `assume` statements.

It is harder to make a general claim as to whether most sequential analysis can handle symbolic values. One place where symbolic values are used in sequential analysis is for the analysis of recursive procedures. Eager computation of a procedure summary is similar to analyzing the procedure while assuming symbolic values for the parameters of the procedure.

It is easy to see that our reduction applies quite easily to concurrent programs that only share finite-state data. In this case, the symbolic constants can only take a finite number of values. Thus, any sequential analysis can be extended for CBA by simply enumerating all their values (or considering them lazily using techniques similar to the ones presented in §5). This implies that sequential

analyses of programs with pointers, arrays, and/or integers can be extended to perform CBA of such programs when only finite-state data (e.g., finite number of locks) is shared between the threads. This shows the value of our reduction for creating new algorithms for CBA.

The reduction also applies when the shared data is not finite state, although, in this case, values of symbolic constants cannot be enumerated. For instance, our reduction can take a concurrent numeric program (defined as one having multiple threads, each manipulating some number of potentially unbounded integers), and produce a sequential numeric program. Then most numeric analysis, like polyhedral analysis, can be applied on the program. Such analyses are typically able to handle symbolic constants [5].

3 The Reduction for Boolean programs

Boolean Programs. A Boolean program consists of a set of procedures, represented using their control-flow graphs (CFGs). The program has a set of global variables, and each procedure has a set of local variables, where each variable can only receive a Boolean value. Each edge in the CFG is labeled with a statement that can read from and write to variables in scope, or call a procedure. An example is shown in Fig. 2.

For ease of exposition, we assume that all procedures have the same number of local variables, and that they do not have any parameters. Furthermore, the global variables can have any value when program execution starts, and similarly for the local variables when a procedure is invoked.

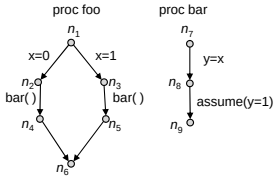

Fig. 2. A Boolean program

Let G be the set of valuations of the global variables, and L be the set of valuations of the local variables. A program *data-state* is an element of $G \times L$. Each program statement $\mathtt{st}$ can be associated with a relation $[\![\mathtt{st}]\!] \subseteq (G \times L) \times (G \times L)$ such that $(g_0, l_0, g_1, l_1) \in [\![\mathtt{st}]\!]$ when the execution of $\mathtt{st}$ on the state (g_0, l_0) can lead to the state (g_1, l_1). For instance, in a procedure with one global variable $\mathtt{x}_1$ and one local variable $\mathtt{x}_2$, $[\![\mathtt{x}_1 = \mathtt{x}_2]\!] = \{(a, b, b, b) \mid a, b \in \{0, 1\}\}$ and $[\![\mathtt{assume}(\mathtt{x}_1 = \mathtt{x}_2)]\!] = \{(a, a, a, a) \mid a \in \{0, 1\}\}$.

The goal of analyzing such programs is to compute the set of data-states that can reach a program node. This is done using the rules shown in Fig. 3 [1]. These rules follow standard interprocedural analyses [21, 25]. Let entry($\mathtt{f}$) be the entry node of procedure $\mathtt{f}$, proc(n) the procedure that contains node n, ep(n) = entry(proc(n)), and exitnode(n) is true when n is the exit node of its procedure. Let Pr be the set of procedures of the program, which includes a distinguished procedure $\mathtt{main}$. The rules compute three types of relations: $H_n(g_0, l_0, g_1, l_1)$ denotes the fact that if (g_0, l_0) is the data state at entry(n), then the data state (g_1, l_1) can reach node n; $S_{\mathtt{f}}$ is the summary relation for procedure $\mathtt{f}$, which captures the net transformation that an invocation of the

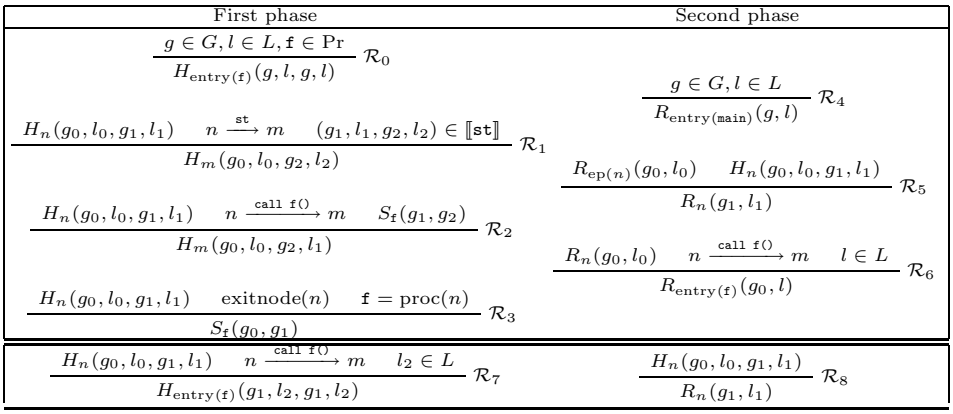

Fig. 3. Rules for the analysis of Boolean programs.

procedure can have on the global state; R_n is the set of data states that can reach node n. All relations are initialized to be empty.

Eager analysis. Rules $\mathcal{R}_0$ to $\mathcal{R}_6$ describe an eager analysis. The analysis proceeds in two phases. In the first phase, the rules $\mathcal{R}_0$ to $\mathcal{R}_3$ are used to saturate the relations H and S. In the next phase, this information is used to build the relation R using rules $\mathcal{R}_4$ to $\mathcal{R}_6$.

Lazy analysis. Let rule $\mathcal{R}_0'$ be the same as $\mathcal{R}_0$ but restricted to just the `main` procedure. Then the rules $\mathcal{R}_0, \mathcal{R}_1, \mathcal{R}_2, \mathcal{R}_3, \mathcal{R}_7, \mathcal{R}_8$ describe a lazy analysis. The rule $\mathcal{R}_7$ restricts the analysis of a procedure to only those states it is called in. As a result, the second phase gets simplified and consists of only the rule $\mathcal{R}_8$.

Practical implementations [1, 23] use BDDs to encode each of the relations H, S, and R and the rule applications are changed into BDD operations. For example, rule $\mathcal{R}_1$ is simply the relational composition of relations H_n and $[\![\mathtt{st}]\!]$, which can be implemented efficiently using BDDs.

Concurrent Boolean Programs. A concurrent Boolean program consists of multiple Boolean programs, one per thread. The Boolean programs share their set of global variables. In this case, we can apply the reduction presented in §2, with slight changes, to obtain a single Boolean program: (*i*) the variable `k` is modeled using a vector of $\log(K)$ Boolean variables, and the increment operation implemented using a simple Boolean circuit on these variables; (*ii*) the **if** conditions are modeled using `assume` statements; and (*iii*) the symbolic constants are modeled using additional global variables that are not modified in the program. Running any sequential analysis algorithm, and projecting out the values of the K^{th} set of global variables from R_n gives the precise set of reachable global states at node n in the concurrent program.

The worst-case complexity of analyzing a Boolean program P with $|G|$ number of valuations of global variables and $|L|$ number of valuations of local

variables is $\mathcal{O}(|P||G|^3|L|^2)$, where $|P|$ is the number of program statements. The complexity of analyzing a concurrent Boolean program P_c with n threads, where each thread gets K execution contexts (with round-robin scheduling), is $\mathcal{O}(|P_c||G|^{4K}|L|^2)$ (it has one extra factor of $|G|^K$ for choosing the initial values of the symbolic constants).

Note that there is nothing special about the use of Boolean programs in our reduction, except that the number of data-states is finite. Our reduction applies to any model that works with finite-state data, which includes Boolean programs with references [2, 16]. In such models, the heap is assumed to be bounded in size. The heap is included in the global state of the program, hence, our reduction would create multiple copies of the heap, initialized with symbolic values. Our experiments (§6) used such programs.

Such a process of duplicating the heap can be expensive when the number of heap configurations that actually arise in the concurrent program is very small compared to the total number of heap configurations possible. The lazy version of our algorithm (§5) addresses this issue.

4 The Reduction for PDSs

PDSs are also popular models of programs. The motivation for presenting the reduction for PDSs is that it allows one to apply the numerous algorithms developed for PDSs to concurrent programs under a context bound. For instance, one can use backward analysis of PDSs to get a backward analysis on the concurrent program. In previous work [9], we showed how to precisely compute the *error projection*, i.e., the set of all nodes that lie on an error trace, when the program is modeled as a (weighted) PDS. Directly applying this algorithm to the PDS produced by the following reduction, we can compute the error projection for concurrent programs under a context bound.

Definition 1. *A **pushdown system** is a triple $\mathcal{P} = (P, \Gamma, \Delta)$, where P is a finite set of states, Γ is a finite set of stack symbols, and $\Delta \subseteq P \times \Gamma \times P \times \Gamma^*$ is a finite set of rules. A **configuration** of $\mathcal{P}$ is a pair $\langle p, u\rangle$ where $p \in P$ and $u \in \Gamma^*$. A rule $r \in \Delta$ is written as $\langle p, \gamma\rangle \hookrightarrow \langle p', u\rangle$, where $p, p' \in P$, $\gamma \in \Gamma$ and $u \in \Gamma^*$. These rules define a transition relation $\Rightarrow_{\mathcal{P}}$ on configurations of $\mathcal{P}$ as follows: If $r = \langle p, \gamma\rangle \hookrightarrow \langle p', u'\rangle$ then $\langle p, \gamma u''\rangle \Rightarrow_{\mathcal{P}} \langle p', u'u''\rangle$ for all $u'' \in \Gamma^*$. We drop the subscript from $\Rightarrow_{\mathcal{P}}$ when it is clear from the context. The reflexive transitive closure of $\Rightarrow$ is denoted by $\Rightarrow^*$. For a set of configurations C, we define $pre^*(C) = \{c' \mid \exists c \in C : c' \Rightarrow^* c\}$ and $post^*(C) = \{c' \mid \exists c \in C : c \Rightarrow^* c'\}$, which are just backward and forward reachability under $\Rightarrow$.*

Without loss of generality, we restrict the PDS rules to have at most two stack symbols on the right-hand side [24].

The standard way of modeling control-flow of programs using PDSs is as follows: the set P consists of a single state $\{p\}$; the set Γ consists of program nodes, and Δ has one rule per edge in the control-flow graph as follows: $\langle p, u\rangle \hookrightarrow \langle p, v\rangle$ for an intraprocedural edge (u, v); $\langle p, u\rangle \hookrightarrow \langle p, e\ v\rangle$ for a procedure call at

For each $\langle p, \gamma\rangle \hookrightarrow \langle p', u\rangle \in (\Delta_1 \cup \Delta_2)$ and for all $p_i \in P, k \in \{1, \cdots, K\}$:
$\langle (k, p_1, \cdots, p_{k-1}, p, p_{k+1}, \cdots, p_K), \gamma\rangle \hookrightarrow \langle (k, p_1, \cdots, p_{k-1}, p', p_{k+1}, \cdots, p_K), u\rangle$

For each $\gamma \in \Gamma_j$ and for all $p_i \in P, k \in \{1, \cdots, K\}$:	
$\langle (k, p_1, \cdots, p_K), \gamma\rangle$	$\hookrightarrow \langle (k+1, p_1, \cdots, p_K), \gamma\rangle$
$\langle (K+1, p_1, \cdots, p_K), \gamma\rangle$	$\hookrightarrow \langle (1, p_1, \cdots, p_K), e_{j+1}\ \gamma\rangle$

Fig. 4. PDS rules for $\mathcal{P}_s$.

node u that returns to v and calls the procedure starting at e; $\langle p, u\rangle \hookrightarrow \langle p, \varepsilon\rangle$ if u is the exit node of a procedure. Finite-state data is encoded by expanding P to be the set of global states, and expanding Γ by including valuations of local variables. Under such an encoding, a configuration $\langle p, \gamma_1\gamma_2\cdots\gamma_n\rangle$ represents the instantaneous state of the program: p is the valuation of global variables, γ_1 has the current program location and values of local variables in scope, and $\gamma_2\cdots\gamma_n$ store the return addresses and values of local variables for unfinished calls.

A concurrent program with two threads is represented with two PDSs that share their global state: $\mathcal{P}_1 = (P, \Gamma_1, \Delta_1), \mathcal{P}_2 = (P, \Gamma_2, \Delta_2)$. A configuration of such a system is the triplet $\langle p, u_1, u_2\rangle$ where $p \in P, u_1 \in \Gamma_1^*, u_2 \in \Gamma_2^*$. Define two transition systems: if $\langle p, u_i\rangle \Rightarrow_{\mathcal{P}_i} \langle p', u_i'\rangle$ then $\langle p, u_1, u\rangle \Rightarrow_1 \langle p', u_1', u\rangle$ and $\langle p, u, u_2\rangle \Rightarrow_2 \langle p', u, u_2'\rangle$ for all u. The problem of interest with concurrent programs, under a context bound $2K-1$, is to find the reachable states under the transition system $(\Rightarrow_1^*; \Rightarrow_2^*)^K$ (here the semicolon denotes relational composition, and exponentiation is repeated relational composition).

We now show how to obtain a single PDS $\mathcal{P}_s = (P_s, \Gamma_s, \Delta_s)$, such that reachability under this PDS simulates reachability under the above transition system. Let P_s be the set of all $K+1$ tuples whose first component is a number between 1 and K, and the rest are from the set P, i.e., $P_s = \{1, \cdots, K\} \times P \times P \times \cdots \times P$. This set relates to the reduction from §2 as follows: an element $(k, p_1, \cdots, p_K) \in P_s$ represents that the value of the variable `k` is k; and p_i encodes a valuation of the variables $\textsc{Var}_G^i$. When $\mathcal{P}_s$ is in such a state, its rules would only modify p_k.

Let $e_i \in \Gamma_i$ be the starting node of the i^{th} thread. Let Γ_s be the disjoint union of Γ_1, Γ_2 and an additional symbol $\{e_3\}$. $\mathcal{P}_s$ does not have an explicit checking phase. The rules Δ_s are defined in Fig. 4.

We deviate slightly from the reduction presented in §2 by changing the **goto** statement, which passes control from the first thread to the second, into a procedure call. This is to ensure that the stack of the first thread is left intact when control is passed to the next thread. Furthermore, we assume that the PDSs cannot empty their stacks, i.e., it is not possible that $\langle p, e_1\rangle \Rightarrow_{\mathcal{P}_1}^* \langle p', \varepsilon\rangle$ or $\langle p, e_2\rangle \Rightarrow_{\mathcal{P}_2}^* \langle p', \varepsilon\rangle$ for all $p, p' \in P$ (in other words, the `main` procedure should not return). This can be enforced by introducing new symbols e_i', e_i'' in $\mathcal{P}_i$ such that e_i' calls e_i, pushing e_i'' on the stack, and ensuring that no rule can fire on e_i''.

Theorem 1. *Starting execution of the concurrent program* $(\mathcal{P}_1, \mathcal{P}_2)$ *from the state* $\langle p, e_1, e_2\rangle$ *can lead to the state* $\langle p', c_1, c_2\rangle$ *under the transition system* $(\Rightarrow_1^*; \Rightarrow_2^*)^K$ *if and only if there exist states* $p_2, \cdots, p_K \in P$ *such that* $\langle (1, p, p_2, \cdots, p_K), e_1\rangle \Rightarrow_{\mathcal{P}_s} \langle (1, p_2, p_3, \cdots, p_K, p'), e_3\ c_2\ c_1\rangle$.

Note that the checking phase is implicit in the statement of Thm. 1. (One can also make the PDS $\mathcal{P}_s$ have an explicit checking phase, starting at node e_3.)

Complexity. Using our reduction, one can find the set of all reachable configurations of the concurrent program in time $\mathcal{O}(K^2|P|^{2K}|Proc||\Delta_1 + \Delta_2|)$, where $|Proc|$ is the number of procedures in the program[4] (see App. A). Using backward reachability algorithms, one can verify if a given configuration in reachable in time $\mathcal{O}(K^3|P|^{2K}|\Delta_1+\Delta_2|)$. Both these complexities are asymptotically better than those of previous algorithms for PDSs [18, 11], with the latter being linear in the program size $|\Delta_1 + \Delta_2|$.

A similar reduction works for multiple threads as well (under round-robin scheduling), giving rise to a theorem similar to Thm. 1. Moreover, the complexity of finding all reachable states under a bound of nK with n threads, using a standard PDS algorithm, is $\mathcal{O}(K^3|P|^{4K}|Proc||\Delta|)$, where $|\Delta| = \Sigma_{i=1}^{n}|\Delta_i|$ is the total number of rules in the concurrent program. This implies that when the number of execution contexts per thread (i.e., the value K) is held fixed, our analysis scales linearly in the number of threads.

This reduction produces a large number of rules ($\mathcal{O}(|P|^K|\Delta|)$) in the resultant PDS, but we can leverage work on *symbolic PDSs* (SPDSs) [24] or *weighted PDSs* [22] to obtain symbolic implementations. We show next how to use the former.

4.1 The Reduction for Symbolic PDSs

A SPDS is a triple $(\mathcal{P}, G, val)$, where $\mathcal{P} = (\{p\}, \Gamma, \Delta)$ is a single-state PDS, G is a finite set, and $val : \Delta \to (G \times G)$ assigns a binary relation on G to each PDS rule. val is extended to a sequence of rules as follows: $val([r_1, \cdots, r_n]) = val(r_1); val(r_2); \cdots; val(r_n)$. For a rule sequence $\sigma \in \Delta^*$ and PDS configurations c_1 and c_2, we say $c_1 \Rightarrow^\sigma c_2$ if applying those rules on c_1 results in c_2. The reachability question is extended to computing the join-over-all-paths (JOP) value between two sets of configurations:

$$\text{JOP}(C_1, C_2) = \bigcup\{val(\sigma) \mid c_1 \Rightarrow^\sigma c_2, c_1 \in C_1, c_2 \in C_2\}$$

PDSs and SPDSs have equivalent theoretical power; each can be converted into the other. SPDSs are used for efficiently analyzing PDSs. For a PDS $\mathcal{P} = (P, \Gamma, \Delta)$, one constructs an SPDS as follows: it consists of a PDS $(\{p\}, \Gamma, \Delta')$ and $G = P$. The rules Δ' and their assigned relations are defined as follows: for each $\gamma \in \Gamma, u \in \Gamma^*$, include rule $\langle p, \gamma\rangle \hookrightarrow \langle p, u\rangle$ with the relation $\{(p_1, p_2) \mid \langle p_1, \gamma\rangle \hookrightarrow \langle p_2, u\rangle \in \Delta\}$, if the relation is non-empty. The SPDS captures all

[4] The number of procedures of a PDS is defined as the number of symbols appearing as the first of the two stack symbols on the right-hand side of a call rule.

state changes in the relations associated with the rules. Under this conversion: $\langle p_1, u_1\rangle \Rightarrow_{\mathcal{P}} \langle p_2, u_2\rangle$ if and only if $(p_1, p_2) \in \text{JOP}(\{\langle p, u_1\rangle\}, \{\langle p, u_2\rangle\})$.

The advantage of using SPDSs is that the relations can be encoded using BDDs, and operations such as relational composition and union can be performed efficiently using BDD operations. This allows scalability to large data-state spaces [24]. (SPDSs can also encode part of the local state in the relations, but we do not discuss that issue in this paper.)

The reverse construction can be used to encode an SPDS as a PDS: given an SPDS $((\{p\}, \Gamma, \Delta), G, val)$, construct a PDS $\mathcal{P} = (G, \Gamma, \Delta')$ with rules: $\{\langle g_1, \gamma\rangle \hookrightarrow \langle g_2, u\rangle \mid r = \langle p, \gamma\rangle \hookrightarrow \langle p, u\rangle, r \in \Delta, (g_1, g_2) \in val(r)\}$. Then $(g_1, g_2) \in \text{JOP}(\{\langle p, u_1\rangle\}, \{\langle p, u_2\rangle\})$ if and only if $\langle g_1, u_1\rangle \Rightarrow^* \langle g_2, u_2\rangle$.

Concurrent SPDSs. A concurrent SPDS with two threads consists of two SPDSs $\mathcal{S}_1 = ((\{p\}, \Gamma_1, \Delta_1), G, val_1)$ and $\mathcal{S}_2 = ((\{p\}, \Gamma_1, \Delta_1), G, val_1)$ with the same set G. The transition relation $\Rightarrow_c = (\Rightarrow_1^*; \Rightarrow_2^*)^K$, which describes all paths in the concurrent program for $2K-1$ context switches, is defined in the same manner as for PDS, using the transition relations of the two PDSs. Let $\langle p, e_1, e_2\rangle$ be the starting configuration of the concurrent SPDS. The problem of interest is to compute the following relation for a given set of configurations C:

$$R_C = \text{JOP}(\langle p, e_1, e_2\rangle, C) = \bigcup\{val(\sigma) \mid \langle p, e_1, e_2\rangle \Rightarrow_c^\sigma c, c \in C\}.$$

A concurrent SPDS can be reduced to a single SPDS using the constructions presented earlier: (i) convert the SPDSs $\mathcal{S}_i$ to PDSs $\mathcal{P}_i$; (ii) convert the concurrent PDS system $(\mathcal{P}_1, \mathcal{P}_2)$ to a single PDS $\mathcal{P}_s$; and (iii) convert the PDS $\mathcal{P}_s$ to an SPDS $\mathcal{S}_s$. The rules of $\mathcal{S}_s$ will have binary relations on the set G^K (K-fold cartesian product of G). Recall that the rules of $\mathcal{P}_s$ change the global state in only one component. Thus, the BDDs that represent the relations of rules in $\mathcal{S}_s$ would only be $\log(K)$ times larger than the BDDs for relations in $\mathcal{S}_1$ and $\mathcal{S}_2$ (the identity relation on n elements can be represented with a BDD of size $\log(n)$ [24]).

Let $C' = \{\langle p, e_3\ u_2\ u_1\rangle \mid \langle p, u_1, u_2\rangle \in C\}$. On $\mathcal{S}_s$, one can solve for the value $R = \text{JOP}(\langle p, e_1\rangle, C')$. Then $R_C = \{(g, g') \mid ((g, g_2, \cdots, g_K), (g_2, \cdots, g_K, g')) \in R\}$ (note the similarity to Thm. 1).

5 Lazy CBA of Concurrent Boolean Programs

In this section, we give a lazy analysis for CBA of concurrent Boolean programs. In the reduction presented in §3, the analysis of the generated sequential program had to assume all possible values for the symbolic constants. The lazy analysis will have the property that at any time, if the analysis considers the K-tuple $(g_1, \cdots, g_K)$ of valuations of the symbolic constants, then there is a *single* valid execution of the concurrent program in which the global state is g_i at the end of the i^{th} execution context of the first thread for all $1 \leq i \leq K$.

The idea is to build up iteratively the effect that each thread can have on the global state in their K execution contexts. Note that T_1^s (or T_2^s) does not need to know the values of VAR_G^i when $\mathtt{k} < i$. Hence, the analysis proceeds by making

no assumptions on the values of $\textsc{Var}_G^i$ when $i >$ `k`. When `k` is incremented to $k+1$ in the analysis of T_1^s, it consults a table E^2 that stores the effect that T_2^s can have in its first k execution contexts. Using that table, it figures out a valuation of $\textsc{Var}_G^{k+1}$ to continue the analysis of T_1^s, and stores the effect that T_1^s can have in its first k execution contexts in table E^1. These tables are built iteratively. More technically, if the analysis can deduce that T_1^s, when started in state $(1, g_1, \cdots, g_k)$, can reach the state $(k, g'_1, \cdots, g'_k)$, and T_2^s, when started in state $(1, g'_1, \cdots, g'_k)$ can reach $(k, g_2, g_3, \cdots, g_k, g_{k+1})$, then an increment of `k` in T_1^s produces the global state $s = (k+1, g'_1, \cdots, g'_k, g_{k+1})$. Moreover, s can be reached when T_1^s is started in state $(1, g_1, \cdots, g_{k+1})$ because T_1^s could not have touched $\textsc{Var}_G^{k+1}$ before the increment that changed `k` to $k+1$.

The algorithm is shown in Fig. 5. The entities used in Fig. 5 have the following meanings:

- Let $\overline{G} = \cup_{i=1}^{K} G^i$, where G is the set of global states. An element from the set $\overline{G}$ is written as $\overline{g}$.
- Let L be the set of local states (all procedures have the same number of local variables).
- The relation H_n^j is related to program node n of the j^{th} thread. It is a subset of $\{1, \cdots, K\} \times \overline{G} \times \overline{G} \times L \times \overline{G} \times L$. If $H_n^j(k, \overline{g_0}, \overline{g_1}, l_1, \overline{g_2}, l_2)$ holds, then each of the $\overline{g_i}$ are an element of G^k (i.e., a k-tuple of global states), and the thread T_j is in its k^{th} execution context. Moreover, if the valuation of $\textsc{Var}_G^i$, $1 \leq i \leq k$, was $\overline{g_0}$ when T_j^s (the reduction of T_j) started executing, and if the node $\text{ep}(n)$ could be reached in data state $(\overline{g_1}, l_1)$, then n can be reached in data state $(\overline{g_2}, l_2)$, and the variables $\textsc{Var}_G^i$, $i > k$ are not touched (hence, there is no need to know their values).
- The relation $S_{\texttt{f}}$ captures the summary of procedure `f`.
- The relations E^j store the *effect* of executing a thread. If $E^j(k, \overline{g_0}, \overline{g_1})$ holds, then $\overline{g_0}, \overline{g_1} \in G^k$, and the execution of thread T_j^s, starting from $\overline{g_0}$ can lead to $\overline{g_1}$, without touching variables in $\textsc{Var}_G^i$, $i > k$.
- The function $\texttt{check}(k, (g_1, \cdots, g_k), (g'_1, \cdots, g'_k))$ returns g'_k if $g_{i+1} = g'_i$ for $1 \leq i \leq k-1$, and is undefined otherwise. This function checks for the correct handing off of the global state when T_2 stops and T_1 starts an execution context.
- Let $[(g_1, \cdots, g_i), (g_{i+1}, \cdots g_j)] = (g_1, \cdots, g_j)$, and we sometimes write g to mean (g), i.e., $[(g_1, \cdots, g_i), g] = (g_1, \cdots, g_i, g)$.

Understanding the rules. The rules $\mathcal{R}'_1, \mathcal{R}'_2, \mathcal{R}'_3$, and $\mathcal{R}'_7$ describe intra-thread computation, and are similar to the corresponding unprimed rules in Fig. 3. The rule $\mathcal{R}_{10}$ initializes the variables for the first execution context of T_1. The rule $\mathcal{R}_{11}$ initializes the variables for the first execution context of T_2. The rules $\mathcal{R}_8$ and $\mathcal{R}_9$ ensure proper hand off of the global state from one thread to another. These two are the only rules that change the value of k. For example, consider rule $\mathcal{R}_8$. It ensures that the global state at the end of k^{th} execution context of T_2 is passed to the $(k+1)^{\text{th}}$ execution context of T_1, using the function `check`. The value g returned by this function represents a reachable valuation

$$\frac{H_n^j(k,\overline{g_0},\overline{g_1},l_1,[\overline{g_2},g_3],l_3) \quad n \xrightarrow{\texttt{st}} m \quad (g_3,l_3,g_4,l_4) \in [\![\texttt{st}]\!]}{H_m^j(k,\overline{g_0},\overline{g_1},l_1,[\overline{g_2},g_4],l_4)}\ \mathcal{R}_1'$$

$$\frac{H_n^j(k,\overline{g_0},\overline{g_1},l_1,\overline{g_2},l_2) \quad n \xrightarrow{\texttt{call f()}} m \quad S_{\texttt{f}}(k+i,[\overline{g_2},\overline{g}],[\overline{g_3},\overline{g'}])}{H_m^j(k+i,[\overline{g_0},\overline{g}],[\overline{g_1},\overline{g}],l_1,[\overline{g_3},\overline{g'}],l_2)}\ \mathcal{R}_2'$$

$$\frac{H_n^j(k,\overline{g_0},\overline{g_1},l_1,\overline{g_2},l_2) \quad \text{exitnode}(n) \quad \texttt{f} = \text{proc}(n)}{S_{\texttt{f}}(k,\overline{g_1},\overline{g_2})}\ \mathcal{R}_3' \qquad \frac{g \in G, l \in L, e = \text{entry}(\texttt{main})}{H_e^1(1,g,g,l,g,l)}\ \mathcal{R}_{10}$$

$$\frac{H_n^j(k,\overline{g_0},\overline{g_1},l_1,\overline{g_2},l_2) \quad n \xrightarrow{\texttt{call f()}} m \quad l_3 \in L}{H^j_{\text{entry(f)}}(k,\overline{g_0},\overline{g_2},l_3,\overline{g_2},l_3)}\ \mathcal{R}_7' \qquad \frac{H_n^j(k,\overline{g_0},\overline{g_1},l_1,\overline{g_2},l_2)}{E^j(k,\overline{g_0},\overline{g_2})}\ \mathcal{R}_{11}$$

$$\frac{H_n^1(k,\overline{g_0},\overline{g_1},l_1,\overline{g_2},l_2) \quad E^2(k,\overline{g_2},\overline{g_3}) \quad g = \texttt{check}(\overline{g_0},\overline{g_3})}{H_n^1(k+1,[\overline{g_0},g],[\overline{g_1},g],l_1,[\overline{g_2},g],l_2)}\ \mathcal{R}_8 \qquad \frac{E^1(1,g_0,g_1), l \in L}{H_{e_2}^2(1,g_1,g_1,l,g_1,l)}\ \mathcal{R}_{12}$$

$$\frac{H_n^2(k,\overline{g_0},\overline{g_1},l_1,\overline{g_2},l_2) \quad E^1(k+1,[g_3,\overline{g_2}],[\overline{g_0},g_4])}{H_n^2(k+1,[\overline{g_0},g_4],[\overline{g_1},g_4],l_1,[\overline{g_2},g_4],l_2)}\ \mathcal{R}_9$$

Fig. 5. Rules for lazy analysis of concurrent Boolean programs.

of the global variables when T_1 starts its $(k+1)^{\text{th}}$ execution context. (In fact, for $(g_1,\cdots,g_k,g_{k+1}) = [\overline{g_0},g]$, there must be an execution of the concurrent program in which the global state is g_i when T_i begins its i^{th} execution context, $1 \le i \le k+1$.)

The following theorem shows that the relations E^1 and E^2 are built lazily, i.e., they only contain relevant information.

Theorem 2. *After running the algorithm described in Fig. 5, $E^1(k,(g_1,\cdots,g_k),(g_1',\cdots,g_k'))$ and $E^2(k,(g_1',\cdots,g_k'),(g_2,\cdots,g_k,g))$ hold if and only if there is an execution on the concurrent program with $2k-1$ context switches that starts in state g_1 and ends in state g, and the global state is g_i at the start of the i^{th} execution context of T_1 and g_i' at the start of the i^{th} execution context of T_2. In particular, the set of reachable global states of the concurrent program in $2K-1$ context switches are all $g \in G$ such that $E^2(K,\overline{g_1},[\overline{g_2},g])$ holds.*

6 Experiments

We did a proof-of-concept implementation of the eager algorithm for Boolean programs, presented in §3, using the model checker Moped [23]. In most cases, we took sequential programs and assumed that there were two copies of the program running concurrently (expect for `BlueT`). The input programs are obtained from a variety of sources: `BlueT` is a model of a Bluetooth driver [19]; `Java*` are the result of abstracting Java programs [2]; `Reg*` are from the regression suite of

MOPED; `Toy` is a toy program we wrote for checking correctness. Some programs, especially ones obtained from Java programs, have pointers and a bounded heap (which is accounted for in the number of variables). We wanted to verify if a certain program node was reachable by finding the set of reachable data-states at the node. In most cases, we modified the programs to have both positive and negative instances.

Prog	Inst	$2K$	Time (s)	|Prog|	#gvars	#lvars
`Toy`	pos	20	0.3	12	5	0
`Reg-blast1`	neg	20	3.9	19	7	21
`Reg-blast1`	pos	20	4.1	19	7	21
`Reg-slam1`	pos	20	19.6	19	1	10
`BlueT`	neg	20	7.2	30	10	1
`BlueT`	pos	10	7.6	30	10	1
`JavaMeeting`	neg	10	168.5	537	16	64
`JavaMeeting`	pos	10	361.3	537	16	64
`JavaChange`	neg	10	770.8	601	24	38
`JavaChange`	pos	10	1134.4	601	24	38

Fig. 6. Experiments on finite-data-state models.

The results are shown in Fig. 6. The last three columns give the size of the program (total number of CFG edges), the number of global variables, and the maximum number of local variables in a procedure, respectively. They show that our algorithms are practical—the data-state space of the last program has about 2^{158} possible states. Other techniques that address CBA would not have scaled to these programs. Also note that negative cases take less time than positive cases. This is because of the way we have implemented the BDD operations. (In some cases, we can conclude that a set is empty, i.e., a node is not reachable, without applying all the required operations. For positive cases, this never happens, and all the operations have to be applied.)

We do not compare against other methods of analyzing concurrent programs that do not address CBA. That study is beyond the scope of this paper. (Note that with CBA, one can precisely handle recursive programs, but their analysis would be undecidable in the absence of a context bound.)

7 Related Work

Most of the related work on CBA has been covered in the body of the paper. A reduction from concurrent programs to sequential programs was given in [19] for the case of two threads and two context switches (it has a restricted extension to multiple threads as well). In such a case, the only thread interleaving is $T_1; T_2; T_1$. The context switch from T_1 to T_2 is simulated by a procedure call. Then T_2 is executed on the program stack of T_1, and at the next context switch, the stack of T_2 is popped off to resume execution in T_1. Because the stack of T_2 is destroyed, the analysis cannot return to T_2 (hence the context bound of 2). Their algorithm cannot be generalized to an arbitrary context bound.

Analysis of message-passing concurrent systems, as opposed to ones having shared memory, has been considered in [4]. They bound the number of messages that can be communicated, similar to bound the number of contexts.

There has been a large body of work on verification of concurrent programs. Some recent work is [7, 17]. However, CBA is different because it allows for precise analysis of complicated program models, including recursion. As future

work, it would be interesting to explore CBA with the abstractions used in the aforementioned work.

References

1. T. Ball and S. Rajamani. Bebop: A symbolic model checker for Boolean programs. In *SPIN*, 2000.
2. F. Berger, S. Schwoon, and D. Suwimonteerabuth. jMoped, 2005. `http://www.informatik.uni-stuttgart.de/fmi/szs/tools/moped/jmoped/`.
3. A. Bouajjani, S. Fratani, and S. Qadeer. Context-bounded analysis of multithreaded programs with dynamic linked structures. In *CAV*, 2007.
4. S. Chaki, E. M. Clarke, N. Kidd, T. W. Reps, and T. Touili. Verifying concurrent message-passing C programs with recursive calls. In *TACAS*, 2006.
5. P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *POPL*, 1978.
6. T. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *POPL*, 2002.
7. T. A. Henzinger, R. Jhala, and R. Majumdar. Race checking by context inference. In *PLDI*, 2004.
8. J. Knoop and B. Steffen. The interprocedural coincidence theorem. In *CC*, 1992.
9. A. Lal, N. Kidd, T. Reps, and T. Touili. Abstract error projection. In *SAS*, 2007.
10. A. Lal, T. Touili, N. Kidd, and T. Reps. Interprocedural analysis of concurrent programs under a context bound. TR-1598, University of Wisconsin, July 2007.
11. A. Lal, T. Touili, N. Kidd, and T. Reps. Interprocedural analysis of concurrent programs under a context bound. In *TACAS*, 2008.
12. M. Müller-Olm and H. Seidl. Precise interprocedural analysis through linear algebra. In *POPL*, 2004.
13. M. Müller-Olm and H. Seidl. Analysis of modular arithmetic. In *ESOP*, 2005.
14. B. Murphy and M. Lam. Program analysis with partial transfer functions. In *PEPM*, 2000.
15. M. Musuvathi and S. Qadeer. Iterative context bounding for systematic testing of multithreaded programs. In *PLDI*, 2007.
16. S. Qadeer and S. Rajamani. Deciding assertions in programs with references. Technical Report MSR-TR-2005-08, Microsoft Research, Redmond, Jan. 2005.
17. S. Qadeer, S. K. Rajamani, and J. Rehof. Summarizing procedures in concurrent programs. In *POPL*, 2004.
18. S. Qadeer and J. Rehof. Context-bounded model checking of concurrent software. In *TACAS*, 2005.
19. S. Qadeer and D. Wu. KISS: Keep it simple and sequential. In *PLDI*, 2004.
20. G. Ramalingam. Context-sensitive synchronization-sensitive analysis is undecidable. In *TOPLAS*, 2000.
21. T. Reps, S. Horwitz, and M. Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *POPL*, 1995.
22. T. Reps, S. Schwoon, S. Jha, and D. Melski. Weighted pushdown systems and their application to interprocedural dataflow analysis. In *SCP*, volume 58, 2005.
23. S. Schwoon. Moped. `http://www.fmi.uni-stuttgart.de/szs/tools/moped/`.
24. S. Schwoon. *Model-Checking Pushdown Systems*. PhD thesis, Technical Univ. of Munich, Munich, Germany, July 2002.
25. M. Sharir and A. Pnueli. Two approaches to interprocedural data flow analysis. In *Program Flow Analysis: Theory and Applications*. Prentice-Hall, 1981.

A Proofs

A.1 Proof of Thm. 1

($\Leftarrow$) First, we show that a path of the concurrent program can be simulated by a path in the sequential program. (In this proof, we will deviate from the notation of the theorem to make the proof more clear.) Let $c_0 = e_1$, and $d_0 = e_2$. If the configuration $\langle p_0, c_0, d_0\rangle$ can lead to $\langle p_{2K}, c_K, d_K\rangle$ under the transition system $(\Rightarrow_1^*; \Rightarrow_2^*)^K$, then we show that there exist states $p_2, p_4, \cdots, p_{2K-2} \in P$ such that $\langle (1, p_0, p_2, \cdots, p_{2K-2}), c_0\rangle \Rightarrow_{\mathcal{P}_s} \langle (1, p_2, p_4, \cdots, p_{2K}), e_3\ d_K\ c_K\rangle$.

If a sequence of rules σ take a configuration c to a configuration c' under the transition system $\Rightarrow$, then we say $c \Rightarrow^\sigma c'$. For a rule $r \in \Delta_i$, $r = \langle p, \gamma\rangle \hookrightarrow \langle p', u\rangle$, let $r^s[k, p_1, \cdots, p_{k-1}, p_{k+1}, \cdots p_K] \in \Delta_s$ be the rule $\langle (k, p_1, \cdots, p_{k-1}, p', p_{k+1}, \cdots, p_K), \gamma\rangle \hookrightarrow \langle (k, p_1, \cdots, p_{k-1}, p', p_{k+1}, \cdots, p_K), u\rangle$. We extend this notation to rule sequences as well, and drop the p_i, when they are clear from the configuration the rules are applied on. Let $r_{\text{inc}}[k]$ stand for a rule of $\mathcal{P}_s$ that increments the value of k (note that it can fire with anything on the top of the stack). Let $r_{1\to 2}$ stand for the rules that call from the first PDS to the second, and $r_{2\to 3}$ stand for the rules that call e_3.

A path in $(\Rightarrow_1^*; \Rightarrow_2^*)^K$ can be broken down at each switch from $\Rightarrow_1$ to $\Rightarrow_2$, and from $\Rightarrow_2$ to $\Rightarrow_1$. Hence, there must exist c_i, d_i, $1 \le i \le K-1$; p_j, $1 \le j \le 2K-1$; and σ_h, $1 \le h \le 2K$, such that a path in the concurrent program can be broken down as shown in Fig. 7(a). Then the path shown in Fig. 7(b) is a valid run of $\mathcal{P}_s$ that establishes the required property.

($\Rightarrow$) For the reverse direction, a path σ in $\Rightarrow_{\mathcal{P}_s}$, from $\langle (1, p_0, p_2, \cdots, p_{2K-2}), c_0\rangle$ to $\langle (1, p_2, p_4, \cdots, p_{2K}), e_3\ d_K\ c_K\rangle$ can be broken down as $\sigma = \sigma_A\ r_{1\to 2}\ \sigma_B\ r_{2\to 3}$. (This is because one must use the rules $r_{1\to 2}$ and $r_{2\to 3}$, in order, to push e_3 on the stack, after which no rules can fire.) Hence we must have the following (for some states $p_1, p_3, \cdots, p_{2K-1}$):

$$\begin{aligned}
\langle (1, p_0, p_2, \cdots, p_{2K-2}), c_0\rangle &\Rightarrow_{\mathcal{P}_s}^{\sigma_A} \langle (K+1, p_1, p_3, \cdots, p_{2K-1}), c_K\rangle \\
&\Rightarrow_{\mathcal{P}_s}^{r_{1\to 2}} \langle (1, p_1, p_3, \cdots, p_{2K-1}), d_0\ c_K\rangle \\
&\Rightarrow_{\mathcal{P}_s}^{\sigma_B} \langle (K+1, p_2, p_4, \cdots, p_{2K}), d_K\ c_K\rangle \\
&\Rightarrow_{\mathcal{P}_s}^{r_{2\to 3}} \langle (1, p_2, p_4, \cdots, p_{2K}), e_3\ d_K\ c_K\rangle
\end{aligned}$$

Because σ_A changes the value of $\mathtt{k}$ from 1 to $K+1$, it must have $K+1$ uses of r_{inc}. Hence, it can be written as: $\sigma_A = \sigma_1^s[1]\ r_{\text{inc}}[1]\ \sigma_3^s[2]\ r_{\text{inc}}[2] \cdots r_{\text{inc}}[K-1]\ \sigma_{2K-1}^s[K]\ r_{\text{inc}}[K]$. Because only $\sigma^s[i]$ can change the i^{th} state component, we must have the following:

$$\begin{aligned}
\langle (1, p_0, p_2, \cdots, p_{2K-2}), c_0\rangle &\Rightarrow_{\mathcal{P}_s}^{\sigma_1^s[1]} \langle (1, p_1, p_2, \cdots, p_{2K-2}), c_1\rangle \\
&\Rightarrow_{\mathcal{P}_s}^{r_{\text{inc}}[1]} \langle (2, p_1, p_2, \cdots, p_{2K-2}), c_1\rangle \\
&\cdots \\
&\Rightarrow_{\mathcal{P}_s}^{\sigma_{2K-1}^s[K]} \langle (K, p_1, p_3, \cdots, p_{2K-1}), c_K\rangle \\
&\Rightarrow_{\mathcal{P}_s}^{r_{\text{inc}}[K]} \langle (K+1, p_1, p_3, \cdots, p_{2K-1}), c_K\rangle
\end{aligned}$$

	$\langle p_0, c_0, d_0\rangle$
$\Rightarrow_1^{\sigma_1}$	$\langle p_1, c_1, d_0\rangle$
$\Rightarrow_2^{\sigma_2}$	$\langle p_2, c_1, d_1\rangle$
$\Rightarrow_1^{\sigma_3}$	$\langle p_3, c_2, d_1\rangle$
$\Rightarrow_2^{\sigma_4}$	$\langle p_4, c_2, d_2\rangle$
$\cdots$	
$\Rightarrow_1^{\sigma_{2K-1}}$	$\langle p_{2K-1}, c_K, d_{K-1}\rangle$
$\Rightarrow_2^{\sigma_{2K}}$	$\langle p_{2K}, c_K, d_K\rangle$
(a)	

	$\langle(1, p_0, p_2, \cdots, p_{2K-2}), c_0\rangle$
$\Rightarrow^{\sigma_1^s[1]}$	$\langle(1, p_1, p_2, p_4, \cdots, p_{2K-2}), c_1\rangle$
$\Rightarrow^{r_{\text{inc}}[1]}$	$\langle(2, p_1, p_2, p_4, \cdots, p_{2K-2}), c_1\rangle$
$\Rightarrow^{\sigma_3^s[2]}$	$\langle(2, p_1, p_3, p_4, \cdots, p_{2K-2}), c_2\rangle$
$\Rightarrow^{r_{\text{inc}}[2]}$	$\langle(3, p_1, p_3, p_4, \cdots, p_{2K-2}), c_2\rangle$
$\cdots$	
$\Rightarrow^{r_{\text{inc}}[K-1]}$	$\langle(K, p_1, p_3, p_5, \cdots, p_{2K-3}, p_{2K-2}), c_{K-1}\rangle$
$\Rightarrow^{\sigma_{2K-1}^s[K]}$	$\langle(K, p_1, p_3, p_5, \cdots, p_{2K-3}, p_{2K-1}), c_K\rangle$
$\Rightarrow^{r_{\text{inc}}[K]}$	$\langle(K+1, p_1, p_3, p_5, \cdots, p_{2K-3}, p_{2K-1}), c_K\rangle$
$\Rightarrow^{r_{1\to 2}}$	$\langle(1, p_1, p_3, p_5, \cdots, p_{2K-1}), d_0\ c_K\rangle$
$\Rightarrow^{\sigma_2^s[1]}$	$\langle(1, p_2, p_3, p_5, \cdots, p_{2K-1}), d_1\ c_K\rangle$
$\Rightarrow^{r_{\text{inc}}[1]}$	$\langle(2, p_2, p_3, p_5, \cdots, p_{2K-1}), d_1\ c_K\rangle$
$\cdots$	
$\Rightarrow^{r_{\text{inc}}[K-1]}$	$\langle(K, p_2, p_4, p_6, \cdots, p_{2K-2}, p_{2K-1}), d_{K-1}\ c_K\rangle$
$\Rightarrow^{\sigma_{2K}^s[K]}$	$\langle(K, p_2, p_4, p_6, \cdots, p_{2K-2}, p_{2K}), d_K\ c_K\rangle$
$\Rightarrow^{r_{\text{inc}}[K]}$	$\langle(K+1, p_2, p_4, p_6, \cdots, p_{2K-2}, p_{2K}), d_K\ c_K\rangle$
$\Rightarrow^{r_{2\to 3}}$	$\langle(1, p_2, p_4, p_6, \cdots, p_{2K-2}, p_{2K}), e_3\ d_K\ c_K\rangle$
(b)	

Fig. 7. Simulation of a concurrent PDS run by a single PDS. For clarity, we write $\Rightarrow$ to mean $\Rightarrow_{\mathcal{P}_s}$ in (b).

Similarly, $\sigma_B = \sigma_2^s[1]\ r_{\text{inc}}[1]\ \sigma_4^s[2]\ r_{\text{inc}}[2] \cdots r_{\text{inc}}[K-1]\ \sigma_{2K}^s[K]\ r_{\text{inc}}[K]$. The reader can verify that the rule sequence $\sigma_1\ \sigma_2 \cdots \sigma_{2K-1}\ \sigma_{2K}$ describes a path in $(\Rightarrow_1^*; \Rightarrow_2^*)^K$ and takes the configuration $\langle p_0, c_0, d_0\rangle$ to $\langle p_{2K}, c_K, d_K\rangle$.

A.2 Complexity argument for Thm. 1

A PDS can have infinite number of configurations. Hence, sets of configurations are represented using automata [24]. We do not go into the details of such automata, but only present the running-time complexity arguments. Given an automata $\mathcal{A}$, and a PDS $(P_{\text{in}}, \Gamma_{\text{in}}, \Delta_{\text{in}})$, the set of configurations forward reachable from those represented by $\mathcal{A}$ can be calculated in time $\mathcal{O}(|P_{\text{in}}||\Delta_{\text{in}}|(|Q| + |P_{\text{in}}||Proc_{\text{in}}|) + |P_{\text{in}}||\rightarrow_{\mathcal{A}}|)$, where Q is the set of states of $\mathcal{A}$, and $\rightarrow_{\mathcal{A}}$ is the set of its transitions [24]. We call the algorithm from [24] *poststar*, and its output, which is also an automaton, *poststar*($\mathcal{A}$).

For the PDS $\mathcal{P}_s$, obtained from a concurrent PDS with n threads $(\mathcal{P}_1, \mathcal{P}_2, \cdots, \mathcal{P}_n)$, $|P_s| = K|P|^K, |\Delta_s| = K|P|^{K-1}|\Delta|, |Proc_s| = |Proc|$, where $\Delta = \cup_{i=1}^n \Delta_i$ and $|Proc| = \sum_{i=1}^n |Proc_i|$. To obtain the set of forward reachable configurations from $\langle p, e_1, e_2, \cdots, e_n\rangle$, we will solve *poststar*($\mathcal{A}$) for each $\mathcal{A}$ that represents the singleton set of configurations $\{\langle(1, p, p_2, \cdots, p_K), e_1\rangle\}$, i.e., $|P|^{K-1}$ separate calls to *poststar*. In the result, we can project out all configurations that do not have $(1, p_2, \cdots, p_K, p')$ as their state, for some p'. Directly using the above complexity result, we get a total running time of $\mathcal{O}(K^3|P|^{4K}|\Delta||Proc|)$.

For the case of two threads, we use a more sophisticated argument to calculate the running time.

When asking for the set of reachable configurations of $\mathcal{P}_s$, we are only interested in some particular configurations: when starting from $\langle(1, p, p_2, \cdots, p_K), e_1\rangle$, we only want configurations of the form $\langle(1, p_2, \cdots, p_K, p'), u\rangle$. Hence, when we run *poststar*, starting from the above configuration, we remove some rules from Δ_s: we remove all rules with left-hand side $\langle(k, p'_2, p'_3, \cdots, p'_K, p'), \gamma\rangle$ if $\gamma \in \Gamma_2$ and $p_i \neq p'_i$ for some i between 1 and $k-1$, both inclusive. We statically know that removing such rules would not affect the result.

Further, we make two observations about the algorithm from [24]: (i) if an automaton $\mathcal{A}$ is split into two automata $\mathcal{A}_1$ and $\mathcal{A}_2$, such that the union of the transitions (represented configurations) of $\mathcal{A}_1$ and $\mathcal{A}_2$ equals the set of transitions (represented configurations) of $\mathcal{A}$, then the running time of *poststar*($\mathcal{A}$) is strictly smaller than than the sum of the running times of *poststar*($\mathcal{A}_1$) and *poststar*($\mathcal{A}_2$). (ii) splitting the set of PDS rules Δ into two (Δ_1 and Δ_2) such that no rule in Δ_1 can fire after a rule of Δ_2 is applied, then the running time of $\mathit{poststar}_{\Delta_2}(\mathit{poststar}_{\Delta_1}(\mathcal{A}))$ is the same as the running time of $\mathit{poststar}_{\Delta}(\mathcal{A})$, where the *poststar* algorithm is subscripted with the set of rules it operates on. Using these two observations, we show that running *poststar* using $\mathcal{P}_s$ takes less time than the above-mentioned complexity.

Let $\Delta^i \subseteq \Delta_s$ be the set of rules that operate when the first component of the state (the value of `k`) is i, and $\Delta_{\text{call}} \subseteq \Delta_s$ be the set of rules that call to e_2 (from Γ_1) or e_3. We know that any path in $\mathcal{P}_s$ can be decomposed into a rule sequence from $\mathcal{S} = \Delta^{1^*}\ \Delta^{2^*} \cdots \Delta^{K^*}\ \Delta_{\text{call}}\ \Delta^{1^*}\ \Delta^{2^*} \cdots \Delta^{K^*}\ \Delta_{\text{call}}$. Using observation (ii) above, we break the running of *poststar* on Δ_s into a series operating on each of the above sets, in order. Next, after running *poststar* on one of Δ^{i^*}, we split the resultant automaton $\mathcal{A}$ into as many automata as the number of states in the configurations of $\mathcal{A}$, e.g., if $\mathcal{A}$ represents the set $\{\langle\overline{p}_1, c_1\rangle, \langle\overline{p}_2, c_2\rangle, \langle\overline{p}_2, c_3\rangle\}$, then we split it into two automata representing the sets $\{\langle\overline{p}_1, c_1\rangle\}$ and $\{\langle\overline{p}_2, c_2\rangle, \langle\overline{p}_2, c_3\rangle\}$, respectively. Observation (i) shows that this splitting only increases the running time.

Tab. 1 shows the running time for performing *poststar* on the first K of the Δ^{i^*} from $\mathcal{S}$. The column "Iter" shows which Δ^i is being processed. The column "Num" is the number of *poststar* that have to be run using Δ^i. The column "$|\rightarrow|$" shows the upper bound on the number of transitions in the automaton *poststar* is run on. The column "Time" is the running time of *poststar* on such automata. The column "Split" is an upper bound on the the number of automata the result is split into, and the last column in the number of states in each of the resultant automata. For example, there are $|P|^{i-1}$ number of invocations to *poststar* with rule set Δ^i, each on an automata with at most $(i-1)|P||\Delta||\mathit{Proc}|$ transitions, taking time $2(i-1)|P|^2|\Delta||\mathit{Proc}|$. Each result is split into $|P|$ different automata, each with at most $i|P||\mathit{Proc}|$ states. The reader can inductively verify the correctness of the table.

Thus, this requires a total running time of $\mathcal{O}(K|P|^{K+1}|\Delta||\mathit{Proc}|)$. Next, we use the rules in Δ_{call} and repeat the above process for the last K of the se-

Iter	Num	$\mid \rightarrow \mid$	Time	Split	$\mid Q \mid$
1	1	1	$\mid P \mid^2 \mid \Delta \mid\mid Proc \mid$	$\mid P \mid$	$\mid P \mid\mid Proc \mid$
2	$\mid P \mid$	$\mid P \mid\mid \Delta \mid\mid Proc \mid$	$2\mid P \mid^2 \mid \Delta \mid\mid Proc \mid$	$\mid P \mid$	$2\mid P \mid\mid Proc \mid$
i	$\mid P \mid^{i-1}$	$(i-1)\mid P \mid\mid \Delta \mid\mid Proc \mid$	$2(i-1)\mid P \mid^2 \mid \Delta \mid\mid Proc \mid$	$\mid P \mid$	$i\mid P \mid\mid Proc \mid$
K	$\mid P \mid^{K-1}$	$(K-1)\mid P \mid\mid \Delta \mid\mid Proc \mid$	$2(K-1)\mid P \mid^2 \mid \Delta \mid\mid Proc \mid$	$\mid P \mid$	$K\mid P \mid\mid Proc \mid$

Table 1. Running times for different stages of *poststar* on $\mathcal{P}_s$.

quence $\mathcal{S}$. However, in this case, no splitting is necessary, because we know the desired target state, and have already removed some rules from Δ_s. For example, if the initial state chosen was $(1, p, p_2, \cdots, p_K)$, and after performing the computation of Tab. 1, we obtain an automaton $\mathcal{A}$ that has the single state $(1, p'_1, \cdots .p'_K)$ for all configurations represented by it. After processing $\mathcal{A}$ with Δ^1 suppose the result is $\mathcal{A}'$. There is no need to split $\mathcal{A}'$ because of the rules removed from Δ^2. The rules of Δ^2 would only fire on configurations that have the state $(2, p_2, p'_2, p'_3, \cdots, p'_K)$. Thus, splitting is not necessary, and the time required to process each of the $|P|^{K-1}$ automata obtained from Tab. 1 using Δ^i is $2(K+i-1)|P|^2|\Delta||Proc|$. Hence, the time required to process the entire $\mathcal{S}$ is $\mathcal{O}(K^2|P|^{K+1}|\Delta||Proc|)$. Because we have to repeat for $|P|^{K-1}$ initial states, the running time of *poststar* on $\mathcal{P}_s$ with two threads can be bounded by $\mathcal{O}(K^2|P|^{2K}|\Delta||Proc|)$.

Backward analysis from a set of configurations represented by an automaton $\mathcal{A}$ with $|Q|$ states can be performed in time $\mathcal{O}(K|P|^{2K}(K|P|^K + |Q|)^2|\Delta|)$ for multiple threads, and $\mathcal{O}(K|P|^{2K}(K|P| + |Q|)^2|\Delta|)$ for two threads.

A.3 Proof of Thm. 2

For proving Thm. 2, we will make use of the fact that our reduction to a (sequential) Boolean program is correct. Let T_1^s be the reduction of the first thread, and T_2^s be the reduction of the second thread. First, we show that given an execution ρ of T_1^s, and certain facts about E^2 (which summarizes the effect of the second thread), ρ can be simulated by the subset of rules from Fig. 5 that apply to the first thread. Formally, suppose that ρ is the execution shown in Fig. 8 (where n_0 is the entry point of the thread).

The execution ρ is broken at the points where the value of `k` is incremented. Note that this execution implies that in the concurrent program the global state, when T_1 begins its i^{th} execution context, is g_i, and when T_2 begins its i^{th} execution context, it is g'_i. Further, suppose that the following facts hold: $E^2(i, (g'_1, g'_2, \cdots, g'_i), (g_2, g_3, \cdots, g_k))$ for $1 \leq i \leq k-1$. Given these, we will show that rules for the first thread can be used to establish that $H^1_{n_k}(k, (g_1, \cdots, g_k), \overline{g}, l, (g'_1, \cdots, g'_k))$ holds, for some $\overline{g}$ and l.

Corresponding to the execution ρ, there would be a sequence of deductions, using the rules from Fig. 3 on T_1^s that derives the state at n_k. These rules simply perform an interprocedural analysis on T_1^s (the symbolic constants can take any value when program execution starts). We formalize the notation of

$$
\begin{array}{c}
n_0 : (1, g_1, g_2, \cdots, g_k, g_{k+1}, \cdots, g_K, l_0) \\
\Downarrow \\
n_1 : (1, g_1', g_2, \cdots, g_k, g_{k+1}, \cdots, g_K, l_1) \\
\Downarrow \\
n_1 : (2, g_1', g_2, \cdots, g_k, g_{k+1}, \cdots, g_K, l_1) \\
\Downarrow \\
n_2 : (2, g_1', g_2', \cdots, g_k, g_{k+1}, \cdots, g_K, l_2) \\
\Downarrow \\
\vdots \\
\Downarrow \\
n_{k-1} : (k-1, g_1', g_2', \cdots, g_{k-1}', g_k, g_{k+1}, \cdots, g_K, l_{k-1}) \\
\Downarrow \\
n_k : (k, g_1', g_2', \cdots, g_k', g_{k+1}, \cdots, g_K, l_k)
\end{array}
$$

Fig. 8. An execution in T_1^s.

using these rules on T_1^s. Let the rules operate on the relations H^s and S^s. These relations are of the form: $H_n^s([k_1, \overline{g_1}], l_1, [k_2, \overline{g_2}], l_2)$, which semantically means that if the data state at $\mathrm{ep}(n)$ was $([k_1, \overline{g_1}], l_1)$, then the data state at n can be $([k_2, \overline{g_2}], l_2)$.; and the summary relation would be $S_{\mathtt{f}}^s([k_1, \overline{g_1}], [k_2, \overline{g_2}])$. For a statement `st` in T_1, its translation in T_1^s encodes the transformer: $\{((k, g_1, \cdots, g_k, \cdots, g_K), l, (k, g_1, \cdots, g_k', \cdots, g_K), l') \mid (g_k, l, g_k', l') \in \mathtt{st}\}$. Additionally, one has a self-loop edge associated with a transformer that increments the value of `k`: $\{([k, \overline{g}], l, [k+1, \overline{g}], l) \mid 1 \leq k \leq K\}$. Given a proof tree π for ρ, we build a proof tree π' using rules of Fig. 5 by induction on the bottom-most rule of π.

When $k = 1$ in ρ, the conversion is straightforward: just replace a rule $\mathcal{R}$ in π with the primed rule $\mathcal{R}'$ from Fig. 5. An example is shown in Fig. 9 for a program path $n_0 \xrightarrow{\mathtt{st_1}} n_1 \xrightarrow{\mathtt{call\ f}} n_2$, where the call to `f` takes the path $n_3 \xrightarrow{\mathtt{st_2}} n_4$. Let $(g_1, \cdots, g_{k+i})|_k = (g_1, \cdots, g_k)$.

The induction hypothesis is as follows: given ρ, as shown in Fig. 8, if there is a proof tree π that derives $H_{n_k}^s([k_1, \overline{g}], l, [k, (g_1', \cdots, g_k', g_{k+1}, \cdots, g_K)], l')$ then one can derive $H_{n_k}^1(k, (g_1, \cdots, g_k), \overline{g}|_k, l, (g_1', \cdots, g_k'), l')$. Note that in this case, the last $(K - k_1)$ components of $\overline{g}$ must be $(g_{k_1+1}, \cdots, g_K)$ because T_1^s could not have modified them. We have already proved the base case above. Fix $\overline{g}_{\mathrm{init}} = (g_1, \cdots, g_k)$ and $\overline{g}_{\mathrm{final}} = (g_1', \cdots, g_k', g_{k+1}, \cdots, g_K)$.

The bottom-most rule of π can be $\mathcal{R}_1, \mathcal{R}_2$ or $\mathcal{R}_7$. For the rule $\mathcal{R}_1$, one can either use a statement transformer, or increment the value of `k`. All these cases, and the way to obtain π' are shown in Fig. 10.

One can prove a similar result for T_2^s. Note that $H_{n_k}^1(k, \overline{g}_{\mathrm{init}}, \overline{g}|_k, l, \overline{g}_{\mathrm{final}}|_k, l')$ implies $E^1(k, \overline{g}_{\mathrm{init}}, \overline{g}_{\mathrm{final}}|_k)$. Thus, these results are sufficient to prove one side of the theorem: given an execution of the concurrent program, we can obtain executions of T_1^s and T_2^s, and then use the above results together to show that the rules in Fig. 5 can simulate the execution of the concurrent program.

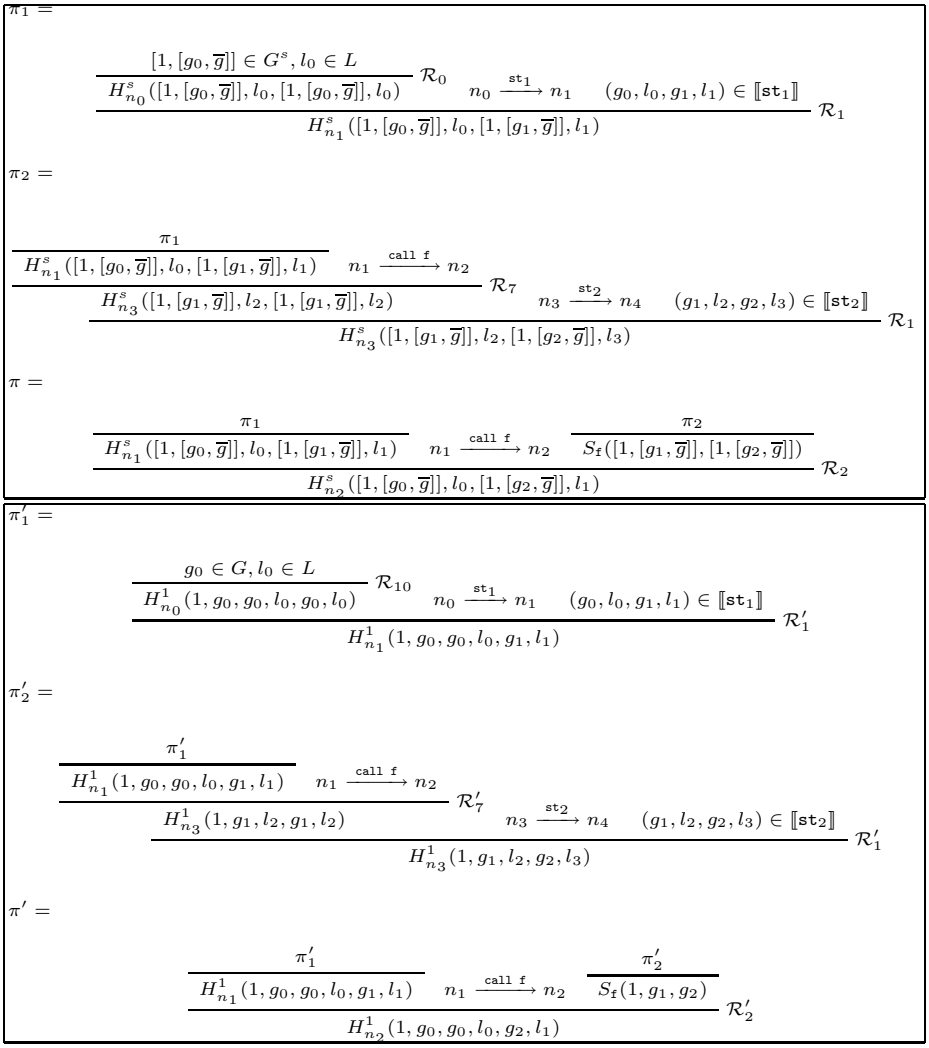

Fig. 9. An example of converting from proof π to proof π'. For brevity, we use `st` to mean a statement in the thread T_1 (and not its translated version in T_1^s).

Going the other way is quite similar. A deduction on H^1 can be converted into an interprocedural path of T_1^s. The rule $\mathcal{R}_8$ corresponds to incrementing the value of `k`, and must be used a bounded number of times in a derivation of H^1 fact. The E^2 assumptions used in a derivation have to be of the form $E^2(1,g'_1,g_2), E^2(2,(g'_1,g'_2),(g_2,g_3)),\cdots,E^2(i,(g'_1,\cdots,g'_i),(g_2,\cdots,g_{i+1}))$. This is because the second component of H^1 is only extended, but never modified, and once `k` is incremented, the first `k` components cannot be modified either. Now, we

can use the conversions of Fig. 10 in the opposite direction to prove the reverse direction of the theorem.

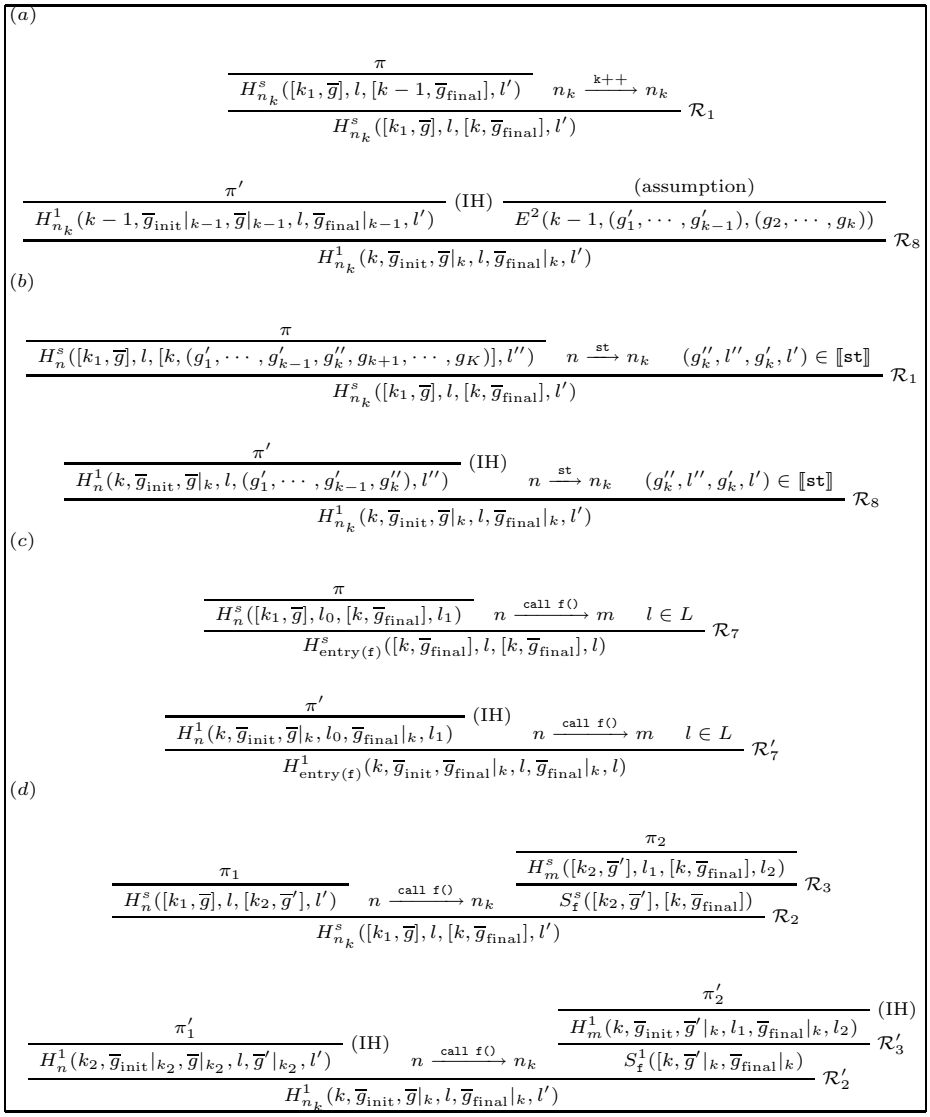

Fig. 10. Simulation of run ρ using rules in Fig. 5. In case (a), $g_k = \texttt{check}(\overline{g}_{\text{init}}|_{k-1},(g_2,\cdots,g_k))$ and $g'_k = g_k$ (because ρ does not edit these set of variables). In case (d), exitnode(m) holds, $\texttt{f} = \text{proc}(m)$, $k_1 \le k_2 \le k$, the k_2+1 to k components of $\overline{g}'$ are $(g_{k_2+1},\cdots,g_k)$ because it arises when $\texttt{k} = k_2$, and the k_1+1 to k components of $\overline{g}$ are $(g_{k_1+1},\cdots,g_k)$ for the same reason.