Reducing Concurrent Analysis Under a Context Bound to Sequential Analysis*

Akash Lal[1]** and Thomas Reps[1,2]

[1] University of Wisconsin; Madison, WI; USA. {akash, reps}@cs.wisc.edu
[2] GrammaTech, Inc.; Ithaca, NY; USA.

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 *sequential* program P_s^K such that the analysis of P_s^K can be used to prove properties about P. We give instances of the reduction for common program models used in model checking, such as Boolean programs and pushdown systems.

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 [18]. 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.

This paper addresses the problem of automatically extending analyses for sequential programs to analyses for concurrent programs under a bound on the number of context switches.[3] 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 [17], a model checker for CBA with a fixed context bound of 2, found numerous bugs in device drivers; a study with explicit-state

* Supported by NSF under grants CCF-0540955 and CCF-0524051 and by AFRL under contract FA8750-06-C-0249.
** Supported by a Microsoft Research Fellowship.
[3] A context switch occurs when execution control passes from one thread to another.

model checkers [13] 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 serves as a general model for a recursive program with finite-state data—was shown in [16]. These results were extended to PDSs with bounded heaps in [3] and to weighted PDSs (WPDSs) in [10]. All of this work required devising new algorithms. Moreover, each of the algorithms have certain disadvantages that make them impractical to implement.

In the sequential setting, model checkers, such as those described in [1, 21, 5], use symbolic techniques in the form of BDDs for scalability. With the CBA algorithms of [16, 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 [10] showed how to extend the algorithm of [16] to use symbolic techniques. However, the disadvantage there is that it requires computing auxiliary information for exploring 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 [12]) 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[4] [1], but the latter (lazy) approach is often desirable in the presence of more complex abstractions, especially those that contain pointers (based on the intuition that only a few aliasing scenarios occur during abstract execution). The option of switching between eager and lazy exploration exists in the model checkers described in [1, 20].

Contributions. This paper makes two main contributions. First, we show how to reduce a concurrent program to a sequential one that simulates all its executions for a given number of context switches. This has the following advantages:

[4] Boolean programs are imperative programs with only the Boolean datatype (§3).

– It allows one to obtain algorithms for CBA using different program abstractions. We specialize the reduction to Boolean programs (§3), PDSs (§4), and symbolic PDSs (see [8]). The former shows that the use of PDS-based technology, which seemed crucial in previous work, is not necessary: standard interprocedural algorithms [19, 22, 7] can also be used for CBA. Moreover, it allows one to carry over symbolic techniques designed for sequential programs for CBA.
– The 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).
– For the case in which a PDS is used to model each thread, we obtain better asymptotic complexity than previous algorithms, just by using the standard PDS algorithms (§4).
– The reduction shows how to obtain algorithms that scale linearly with the number of threads (whereas previous algorithms scaled exponentially).

Second, we show how to obtain a lazy symbolic algorithm for CBA on Boolean programs (§5). This combines the best of previous algorithms: the algorithms of [16, 3] are lazy but not symbolic, and the algorithm of [10] is symbolic but not lazy.

The rest of the paper is organized as follows: §2 gives a general reduction from concurrent to sequential programs; §3 specializes the reduction to Boolean programs; §4 specializes the reduction to PDSs; §5 gives a lazy symbolic algorithm for CBA on Boolean programs; §6 reports early results with our algorithms; §7 discusses related work. Additional details and proofs can be found in [8].

2 A General Reduction

This section gives a general reduction from concurrent programs to sequential programs under a given 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, is stored. Then T_2 gets to run, and when it is swapped out, T_1 has to resume execution from l (along with the global store produced by T_2).

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, which causes exponential blowup when the local state space is finite, and undecidability when the local state includes a stack. An advantage of introducing a context bound is the reduced complexity with respect to the size $|L|$ of the local state space: the algorithms of [16, 3] scale as $\mathcal{O}(|L|^5)$, and [10] scales as $\mathcal{O}(|L|^K)$. Our algorithm, for PDSs, is $\mathcal{O}(|L|)$. (Strictly speaking, in each of these, $|L|$ is the size of the local transition system.)

The key observation is the following: for analyzing $T_1; T_2; T_1$, 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 T_2 can reach state s if and only if the concurrent program can have the global state s after 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 $|L|$. 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. 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 [9]. However, symbolic techniques can be used for a practical implementation.

We show how to reduce the above assume-guarantee process into one of analyzing a sequential program. 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).[5] We assume that the threads share their global variables, i.e., they have the same set of global

[5] Such models are often used in model checking and numeric program analysis.

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**
	  $\tau(\texttt{st}, 2)$;
	  $\cdots$
	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 second column, $*$ stands for a nondeterministic Boolean value.

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 is 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 the label preceding the i^{th} part. P^s has the form shown in the first column of Fig. 1.

The global variables of P^s are K copies of VAR_G. If $\text{VAR}_G = \{\texttt{x}_1, \cdots, \texttt{x}_n\}$, then let $\text{VAR}_G^i = \{\texttt{x}_1^i, \cdots, \texttt{x}_n^i\}$. The initial values of VAR_G^i are a set of symbolic constants that represent the i^{th} guess s_i. P^s 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 $\text{VAR}_G^{\texttt{k}}$. The local variables of T_i^s are the same as those of T_i.

Let $\tau(\texttt{x}, i) = \texttt{x}^i$. If st is a program statement in P, let $\tau(\texttt{st}, i)$ be the statement in which each global variable x is replaced with $\tau(\texttt{x}, i)$, and the local variables 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`. Variables VAR_G^1 are initialized to the same values as VAR_G in P. Variable $\texttt{x}_j^i$, when $i \neq 1$, is initialized to the symbolic constant v_j^i (which is later referenced inside `Checker`), and k is initialized to 1.

Because local variables are not replicated, a thread resumes execution from the same 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 $i+1^{\text{st}}$ 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_G 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 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 n threads, $n > 2$, 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 [16, 10, 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, and bound the length of this schedule instead of bounding the number of context switches. Because a thread is allowed to perform no 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 CBA to consider all thread schedulings with K context switches (as well as some schedulings with more than K context switches). Under such a bound, 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 `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; 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 [11], 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 analyses can handle symbolic values. One place where symbolic values are used in sequential analyses is to construct summaries for recursive procedures. Eager computation of a procedure summary is similar to analyzing the procedure while assuming symbolic values for the parameters of the procedure.

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.

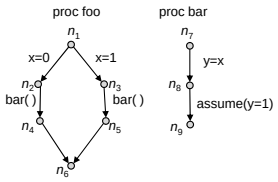

Fig. 2. A Boolean program

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 `st` can be associated with a relation $[\![\texttt{st}]\!] \subseteq (G \times L) \times (G \times L)$ such that $(g_0, l_0, g_1, l_1) \in [\![\texttt{st}]\!]$ when the execution of `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 $\texttt{x}_1$ and one local variable $\texttt{x}_2$, $[\![\texttt{x}_1 = \texttt{x}_2]\!] = \{(a, b, b, b) \mid a, b \in \{0, 1\}\}$ and $[\![\texttt{assume}(\texttt{x}_1 = \texttt{x}_2)]\!] = \{(a, a, a, a) \mid a \in \{0, 1\}\}$.

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 [19, 22]. Let entry(`f`) be the entry node of procedure `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 `main`. The rules of Fig. 3 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; $S_{\texttt{f}}$ is the summary relation for procedure `f`, which captures the net transformation that an invocation of the 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 $\mathcal{R}_6$.

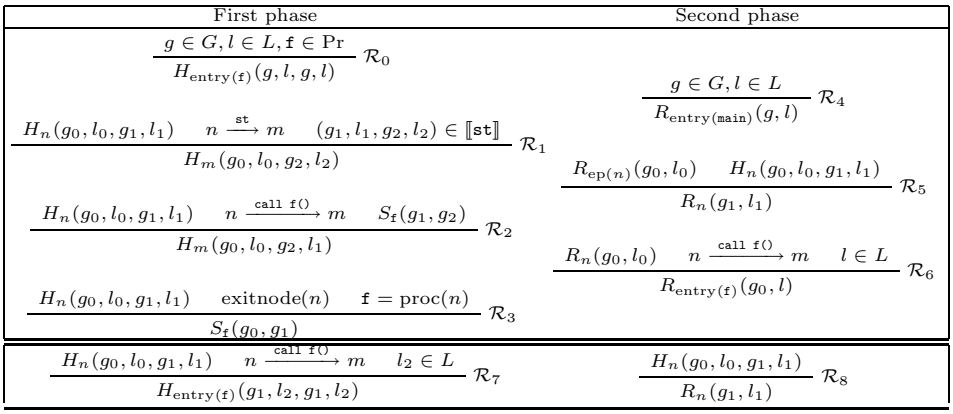

Fig. 3. Rules for the analysis of Boolean programs.

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, 20] 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 one Boolean program per thread. The Boolean programs share their set of global variables. In this case, we can apply the reduction presented in §2 to obtain a single Boolean program by making the following changes to the reduction: (*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 is bounded by $\mathcal{O}(|P||G|^3|L|^2)$, where $|P|$ is the number of program statements. Thus, using our approach, a concurrent Boolean program P_c with n threads, and K execution contexts per thread (with round-robin scheduling), can be analyzed in time $\mathcal{O}(K|P_c|(K|G|^K)^3|L|^2|G|^K)$: the size of the sequential program obtained from P_c is $K|P_c|$; it has the same number of local variables, and its global variables have $K|G|^K$ number of valuations. Additionally, the symbolic constants can take $|G|^K$ number of valuations, adding an extra multiplicative factor of $|G|^K$. The analysis scales linearly with the number of threads ($|P_c|$ is $\mathcal{O}(n)$).

This reduction actually applies to any model that works with finite-state data, which includes Boolean programs with references [2, 14]. 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 models.

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 for CBA. For instance, one can use backward analysis of PDSs to get a backward analysis on concurrent programs.

Definition 1. *A **pushdown system** is a triple $\mathcal{P} = (P, \Gamma, \Delta)$, where P is a set of states, Γ is a 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 $u'' \in \Gamma^*$. The reflexive transitive closure of $\Rightarrow_{\mathcal{P}}$ is denoted by $\Rightarrow^*_{\mathcal{P}}$.*

Without loss of generality, we restrict the PDS rules to have at most two stack symbols on the right-hand side [21].

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 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).

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 $\mathcal{P}_s$.

We reduce the concurrent program $(\mathcal{P}_1, \mathcal{P}_2)$ to a single PDS $\mathcal{P}_s = (P_s, \Gamma_s, \Delta_s)$. 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 ensures 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 for arbitrary PDSs [8].

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.

Complexity. Using our reduction, one can find the set of all reachable configurations of the concurrent program $(\mathcal{P}_1, \mathcal{P}_2)$ in time $\mathcal{O}(K^2|P|^{2K}|Proc||\Delta_1 + \Delta_2|)$, where $|Proc|$ is the number of procedures in the program[6] [8]. 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 [16, 10], 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). Moreover, the complexity of finding all reachable states under a bound of nK with n threads, using a standard PDS reachability 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.

[6] 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.

This reduction produces a large number of rules in $\mathcal{P}_s$, but we can leverage work on *symbolic PDSs* [21] to obtain symbolic implementations [8].

5 Lazy 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 iteratively build up 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 $\textsc{Var}_G^i$ when $\mathtt{k} < i$. Hence, the analysis proceeds by making no assumptions on the values of $\textsc{Var}_G^i$ when $i > \mathtt{k}$. When $\mathtt{k}$ 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 $\mathtt{k}$ 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 $\mathtt{k}$ to $k+1$. The algorithm is shown in Fig. 5. The entities used in it 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.
- 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 $S_{\mathtt{f}}$ captures the summary of procedure $\mathtt{f}$.
- 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 $\mathtt{check}(k, (g_1, \cdots, g_k), (g'_1, \cdots, g'_k))$ 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 transfer of the global state from T_2 to T_1 at a context switch.
- Let $[(g_1, \cdots, g_i), (g_{i+1}, \cdots g_j)] = (g_1, \cdots, g_j)$. We sometimes write g to mean (g), i.e., $[(g_1, \cdots, g_i), g] = (g_1, \cdots, g_i, g)$.

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

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

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

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

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

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

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

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 $\mathcal{R}_{12}$ 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 of the global variables when T_1 starts its $(k+1)^{\text{th}}$ execution context.

The following theorem shows that the relations E^1 and E^2 are built lazily, i.e., they only contain relevant information. A proof can be found in [8].

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 of 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. The set of reachable global states of the 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 [20]. We took sequen-

tial programs and assumed that there were two copies of the program running concurrently (except for `BlueT`). The input programs are obtained from a variety of sources: `BlueT` is a model of a Bluetooth driver [17]; `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 verified 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 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 algorithm is practical—the data-state space of `JavaChange` has about 2^{158} possible states. The negative cases take less time than positive cases because of the way we 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 are applied.

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 [17] 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 [6, 15]. 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. T. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *POPL*, 2002.
6. T. A. Henzinger, R. Jhala, and R. Majumdar. Race checking by context inference. In *PLDI*, 2004.
7. J. Knoop and B. Steffen. The interprocedural coincidence theorem. In *CC*, 1992.
8. A. Lal and T. Reps. Reducing concurrent analysis under a context bound to sequential analysis. Technical Report 1629, University of Wisconsin, 2008.
9. 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.
10. A. Lal, T. Touili, N. Kidd, and T. Reps. Interprocedural analysis of concurrent programs under a context bound. In *TACAS*, 2008.
11. M. Müller-Olm and H. Seidl. Precise interprocedural analysis through linear algebra. In *POPL*, 2004.
12. B. Murphy and M. Lam. Program analysis with partial transfer functions. In *PEPM*, 2000.
13. M. Musuvathi and S. Qadeer. Iterative context bounding for systematic testing of multithreaded programs. In *PLDI*, 2007.
14. S. Qadeer and S. Rajamani. Deciding assertions in programs with references. Technical Report MSR-TR-2005-08, Microsoft Research, Redmond, Jan. 2005.
15. S. Qadeer, S. K. Rajamani, and J. Rehof. Summarizing procedures in concurrent programs. In *POPL*, 2004.
16. S. Qadeer and J. Rehof. Context-bounded model checking of concurrent software. In *TACAS*, 2005.
17. S. Qadeer and D. Wu. KISS: Keep it simple and sequential. In *PLDI*, 2004.
18. G. Ramalingam. Context-sensitive synchronization-sensitive analysis is undecidable. In *TOPLAS*, 2000.
19. T. Reps, S. Horwitz, and M. Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *POPL*, 1995.
20. S. Schwoon. Moped. `http://www.fmi.uni-stuttgart.de/szs/tools/moped/`.
21. S. Schwoon. *Model-Checking Pushdown Systems*. PhD thesis, Technical Univ. of Munich, Munich, Germany, July 2002.
22. M. Sharir and A. Pnueli. Two approaches to interprocedural data flow analysis. In *Program Flow Analysis: Theory and Applications*. Prentice-Hall, 1981.