Form Methods Syst Des (2009) 35: 73–97
DOI 10.1007/s10703-009-0078-9

Reducing concurrent analysis under a context bound to sequential analysis

Akash Lal · Thomas Reps

Published online: 7 July 2009

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. 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 a 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.

Keywords Verification · Concurrency · Context-bounding

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

Supported by NSF under grants CCF-0540955, CCF-0524051, and CCF-0810053, and by IARPA under AFRL contract FA8750-06-C-0249. Any opinions, findings, and conclusions or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of either NSF or IARPA.
Supported by a Microsoft Research Fellowship.

A. Lal (✉) · T. Reps
Computer Sciences Department, University of Wisconsin, Madison, WI 53706, USA
e-mail: akash@cs.wisc.edu

T. Reps
GrammaTech, Inc., Ithaca, NY 14850, USA

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 [22]. 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.[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 [21], a model checker for CBA with a fixed context bound of 2, found numerous bugs in device drivers; a study with explicit-state model checkers [16] 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 [20]. These results were extended to PDSs with bounded heaps in [4] and to weighted PDSs (WPDSs) in [13]. 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 [2, 7, 25], use symbolic techniques in the form of BDDs for scalability. With the CBA algorithms of [4, 20], 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 [13] showed how to extend the algorithm of [20] 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 [15]) 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[2] [2], but the latter (lazy) approach is often desirable in the presence of more complex abstractions, especially those that contain

[1] A context switch occurs when execution control passes from one thread to another.

[2] Boolean programs are imperative programs with only the Boolean datatype (Sect. 3).

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 [2, 9].

1.1 Contributions

This paper makes three 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:

- It allows one to obtain algorithms for CBA using different program abstractions. We specialize the reduction to Boolean programs (Sect. 3), PDSs (Sect. 4), and symbolic PDSs (Sect. 5). The former shows that the use of PDS-based technology, which seemed crucial in previous work, is not necessary: standard interprocedural algorithms [10, 23, 26] can also be used for CBA. Moreover, it allows one to carry over symbolic techniques designed for sequential programs for CBA.
- Our reduction provides a way to harness existing abstraction techniques to obtain new algorithms for CBA. The reduction introduces symbolic constants and `assume` statements. Thus, any sequential analysis that can deal with these two features can be extended to handle concurrent programs as well (under a context bound).

  Symbolic constants are only associated with the shared data in the program. When only a finite amount of data is shared between the 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; for instance, numeric analyses, such as polyhedral analysis [6], can be applied to CBA of concurrent programs.
- 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 (Sect. 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 (Sect. 6). This combines the best of previous algorithms: the algorithms of [4, 20] are lazy but not symbolic, and the algorithm of [13] is symbolic but not lazy.

Third, we implemented both eager and lazy algorithms for CBA on Boolean programs. We report the scalability of these algorithms on programs obtained from various sources and also show that most bugs can be found in a few context switches (Sect. 7).

The rest of the paper is organized as follows: Sect. 2 gives a general reduction from concurrent to sequential programs; Sect. 3 specializes the reduction to Boolean programs; Sect. 4 specializes the reduction to PDSs; Sect. 5 specializes the reduction to symbolic PDSs; Sect. 6 gives a lazy symbolic algorithm for CBA on Boolean programs; Sect. 7 reports experiments performed using both eager and lazy versions of the algorithms presented in this paper; Sect. 8 discusses related work. Proofs can be found in Appendix A.

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-switch 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 $T_1; T_2; T_1; \ldots$. 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 [4, 20] scale as $\mathcal{O}(|L|^5)$, and [13] 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 $s_1, s_2, \ldots, s_{K-1}$. 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 ith 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 [12]. 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.

2.1 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 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 $\text{VAR}_G = \{\mathtt{x}_1, \ldots, \mathtt{x}_n\}$, then let $\text{VAR}_G^i = \{\mathtt{x}_1^i, \ldots, \mathtt{x}_n^i\}$. 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 $\text{VAR}_G^{\mathtt{k}}$. The local variables of T_i^s are the same as those of T_i.

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 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 $\mathtt{x}_j^i$, 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.

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

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

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

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 $i+1$st execution context are the same as the values produced by T_2 when T_2 finished executing its ith 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.

2.2 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 (Sect. 3) and for PDSs (Sect. 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 (Sect. 6).

2.3 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 [4, 13, 20] 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 while analyzing a round-robin schedule of length 6 (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 Sect. 4). These advantages allow our analysis to scale much better in the presence of multiple threads than previous analyses. Our implementation tends to scale linearly with respect to the number of threads (Sect. 7).

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

2.4 Ability of the reduction to harness different analyses for CBA

The reduction introduces `assume` statements and symbolic constants. Any sequential analysis that can deal with these two features can be extended to handle concurrent programs as well (under a context bound).

Any abstraction prepared to interpret program conditions can also handle `assume` statements. Certain analysis, such as affine-relation analysis (ARA) over integers [14] cannot make use of the reduction: the presence of `assume` statements makes the ARA problem undecidable [14].

It is harder to make a general claim about whether most sequential analyses can handle symbolic constants. A variable initialized with a symbolic constant can be treated safely as an uninitialized variable; thus, any analysis that considers all possible values for an uninitialized variable can, in some sense, accommodate symbolic constants.

Another place where symbolic constants 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.

It is easy to see that our reduction applies to concurrent programs that only share finite-state data. In this case, the symbolic constants can only take on a finite number of values. Thus, any sequential analysis can be extended for CBA merely by enumerating all their values (or considering them lazily using techniques similar to the ones presented in Sect. 6). 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., a finite number of locks) is shared between the threads.

The reduction also applies when the shared data is not finite-state, although in this case the values of symbolic constants cannot be enumerated. For instance, the 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 analyses, such as polyhedral analysis [6], can be applied to the program. Such analyses are typically able to handle symbolic constants.

3 The reduction for Boolean programs

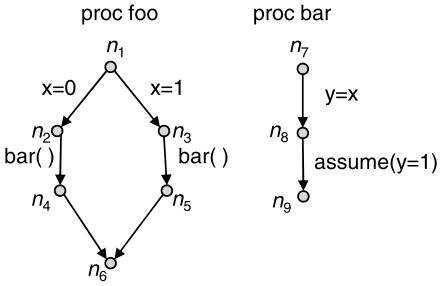
A Boolean program consists of a set of procedures, represented using their control-flow graphs (CFGs). The program has a set of global variables, and each procedure has a set of local variables, where each variable can only receive a Boolean value. Each edge in the CFG is labeled with a statement that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

Boolean programs are common program models used in the model-checking literature. They have been used in predicate-abstraction based model checkers like SLAM [1] and DDVERIFY [28]. They can also encode finite-state abstractions of programs, such as ones used in JMOPED [3] and the BEEM benchmark suite [17]. We used Boolean programs in our experiments (Sect. 7).

For ease of exposition, we assume that all procedures of a Boolean program 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.

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

Fig. 2 A Boolean program



First phase

$$\frac{g \in G, l \in L, \mathtt{f} \in \mathrm{Pr}}{H_{\mathrm{entry}(\mathtt{f})}(g,l,g,l)}\ \mathcal{R}_0$$

$$\frac{H_n(g_0,l_0,g_1,l_1) \qquad n \xrightarrow{\mathtt{st}} m \qquad (g_1,l_1,g_2,l_2) \in [\![\mathtt{st}]\!]}{H_m(g_0,l_0,g_2,l_2)}\ \mathcal{R}_1$$

$$\frac{H_n(g_0,l_0,g_1,l_1) \qquad n \xrightarrow{\mathtt{call\ f()}} m \qquad S_{\mathtt{f}}(g_1,g_2)}{H_m(g_0,l_0,g_2,l_1)}\ \mathcal{R}_2$$

$$\frac{H_n(g_0,l_0,g_1,l_1) \qquad \mathrm{exitnode}(n) \qquad \mathtt{f} = \mathrm{proc}(n)}{S_{\mathtt{f}}(g_0,g_1)}\ \mathcal{R}_3$$

Second phase

$$\frac{g \in G, l \in L}{R_{\mathrm{entry}(\mathtt{main})}(g,l)}\ \mathcal{R}_4$$

$$\frac{R_{\mathrm{ep}(n)}(g_0,l_0) \qquad H_n(g_0,l_0,g_1,l_1)}{R_n(g_1,l_1)}\ \mathcal{R}_5$$

$$\frac{R_n(g_0,l_0) \qquad n \xrightarrow{\mathtt{call\ f()}} m \qquad l \in L}{R_{\mathrm{entry}(\mathtt{f})}(g_0,l)}\ \mathcal{R}_6$$

$$\frac{H_n(g_0,l_0,g_1,l_1) \qquad n \xrightarrow{\mathtt{call\ f()}} m \qquad l_2 \in L}{H_{\mathrm{entry}(\mathtt{f})}(g_1,l_2,g_1,l_2)}\ \mathcal{R}_7$$

$$\frac{H_n(g_0,l_0,g_1,l_1)}{R_n(g_1,l_1)}\ \mathcal{R}_8$$

Fig. 3 Rules for the analysis of Boolean programs

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 $[\![\mathtt{assume}(\mathtt{x}_1 = \mathtt{x}_2)]\!] = \{((a,a),(a,a)) \mid a \in \{0,1\}\}$.

3.1 Analysis of sequential Boolean programs

The goal of analyzing Boolean 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 [2]. These rules follow standard interprocedural analyses [23, 26]. Let $\mathrm{entry}(\mathtt{f})$ be the entry node of procedure $\mathtt{f}$, $\mathrm{proc}(n)$ the procedure that contains node n, $\mathrm{ep}(n) = \mathrm{entry}(\mathrm{proc}(n))$, and $\mathrm{exitnode}(n)$ 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 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 $\mathrm{entry}(n)$, 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 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 $\mathtt{main}$ 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 [2, 9] 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 $[\![\texttt{st}]\!]$, which can be implemented efficiently using BDDs.

3.2 Context-bounded analysis of 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 Sect. 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 (uninitialized) global variables that are not modified in the program. Running any sequential analysis algorithm, and projecting out the values of the Kth 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 [3, 18]. 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 (Sect. 7) 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 (Sect. 6) 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 [11], 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.

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

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

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

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 [25].

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).

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., $P_s = \{1, \ldots, K\} \times P \times P \times \cdots \times P$. This set relates to the reduction from Sect. 2 as follows: an element $(k, p_1, \ldots, p_K) \in P_s$ represents that the value of the variable $\mathtt{k}$ 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 ith 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 Sect. 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 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 $p_2, \ldots, p_K \in P$ such that $\langle (1, p, p_2, \ldots, p_K), e_1\rangle \Rightarrow_{\mathcal{P}_s} \langle (1, p_2, p_3, \ldots, p_K, p'), e_3\ c_2\ c_1\rangle$.*

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

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[4] (see Appendix 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 [13, 20], 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.

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) [25] or *weighted* PDSs [24] to obtain symbolic implementations. We show next how to use the former.

5 The reduction for symbolic PDSs

A *symbolic pushdown system* (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: $val([r_1, \ldots, r_n]) = val(r_1); val(r_2); \cdots ; val(r_n)$ (here the semicolon denotes relational composition). 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:

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

PDSs and SPDSs have equivalent theoretical power; each can be converted to 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 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 $(p_1, p_2) \in \mathrm{JOP}(\{\langle p, u_1\rangle\}, \{\langle p, u_2\rangle\})$.

[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.

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 [25]. (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 $\langle g_1, u_1\rangle \Rightarrow^* \langle g_2, u_2\rangle$.

5.1 Context-bounded analysis of 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 $\log(n)$ [25]).

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 $R_C = \{(g, g') \mid ((g, g_2, \ldots, g_K), (g_2, \ldots, g_K, g')) \in R\}$ (note the similarity to Theorem 1).

6 Lazy CBA of concurrent Boolean programs

In the reduction presented in Sect. 3, the analysis of the generated sequential program had to assume all possible values for the symbolic constants. The lazy analysis has the property that at any time, if the analysis considers the K-tuple $(g_1, \ldots, g_K)$ of valuations of the symbolic constants, then there is at least one valid execution of the concurrent program in which the global state is g_i at the end of the ith 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 its K execution contexts. Note that T_1^s (or T_2^s) does not need to know the values of VAR_G^i when $i > \mathtt{k}$. Hence, the analysis proceeds by making no assumptions on the values of VAR_G^i when $\mathtt{k} < i$. 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 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 precisely, if the analysis can deduce that T_1^s, when started in state $(1, g_1, \ldots, g_k)$, can reach the state $(k, g_1', \ldots, g_k')$, and T_2^s, when started in state $(1, g_1', \ldots, g_k')$ can reach $(k, g_2, g_3, \ldots, g_k, g_{k+1})$, then an increment of $\mathtt{k}$ in T_1^s produces the global state $s = (k +$

$1, g'_1, \ldots, g'_k, g_{k+1})$. Moreover, s can be reached when T_1^s is started in state $(1, g_1, \ldots, g_{k+1})$ because T_1^s could not have touched VAR_G^{k+1} before the increment that changed `k` to $k+1$. The algorithm is shown in Fig. 5. The entities used in it have the following meanings:

- Let $\bar{G} = \cup_{i=1}^{K} G^i$, where G is the set of global states. An element from the set $\bar{G}$ is written as $\bar{g}$. Let L be the set of local states.
- The relation H_n^j is related to program node n of the jth thread. It is a subset of $\{1, \ldots, K\} \times \bar{G} \times \bar{G} \times L \times \bar{G} \times L$. If $H_n^j(k, \bar{g}_0, \bar{g}_1, l_1, \bar{g}_2, l_2)$ holds, then each of the $\bar{g}_i$ are an element of G^k (i.e., a k-tuple of global states), and the thread T_j is in its kth execution context. Moreover, if the valuation of VAR_G^i, $1 \leq i \leq k$, was $\bar{g}_0$ when T_j^s (the reduction of T_j) started executing, and if the node $\text{ep}(n)$, the entry node of the procedure containing n, could be reached in data state $(\bar{g}_1, l_1)$, then n can be reached in data state $(\bar{g}_2, l_2)$, and the variables 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 `f`.
- The relations E^j store the *effect* of executing a thread. If $E^j(k, \bar{g}_0, \bar{g}_1)$ holds, then $\bar{g}_0, \bar{g}_1 \in G^k$, and the execution of thread T_j^s, starting from $\bar{g}_0$ can lead to $\bar{g}_1$, without touching variables in VAR_G^i, $i > k$.
- The function $\texttt{check}(k, (g_1, \ldots, g_k), (g'_1, \ldots, g'_k))$ 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 $[(g_1, \ldots, g_i), (g_{i+1}, \ldots g_j)] = (g_1, \ldots, g_j)$. We sometimes write g to mean (g), i.e., $[(g_1, \ldots, g_i), g] = (g_1, \ldots, g_i, g)$.

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 $\mathcal{R}_{12}$ 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 kth execution context of T_2 is passed to the $(k+1)$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 $(k+1)$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 is given in Appendix A.

Theorem 2 *After running the algorithm described in Fig. 5, $E^1(k, (g_1, \ldots, g_k), (g'_1, \ldots, g'_k))$ and $E^2(k, (g'_1, \ldots, g'_k), (g_2, \ldots, g_k, g))$ 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 ith execution context of T_1 and g'_i at the start of the ith 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 $E^2(K, \bar{g}_1, [\bar{g}_2, g])$ holds.*

7 Experiments

We implemented both the lazy and eager analyses for concurrent Boolean programs by extending the model checker MOPED [9]. These implementations find the set of all reachable states of the shared memory after a given number of context switches. We could have implemented the eager version using a source-to-source transformation; however, we took a

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

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

$$\frac{H_n^j(k,\bar{g_0},\bar{g_1},l_1,\bar{g_2},l_2) \qquad \text{exitnode}(n) \qquad \texttt{f} = \text{proc}(n)}{S_{\texttt{f}}(k,\bar{g_1},\bar{g_2})}\ \mathcal{R}'_3$$

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

$$\frac{H_n^1(k,\bar{g_0},\bar{g_1},l_1,\bar{g_2},l_2) \qquad E^2(k,\bar{g_2},\bar{g_3}) \qquad g = \texttt{check}(\bar{g_0},\bar{g_3})}{H_n^1(k+1,[\bar{g_0},g],[\bar{g_1},g],l_1,[\bar{g_2},g],l_2)}\ \mathcal{R}_8$$

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

$$\frac{g \in G, l \in L, e = \text{entry}(\texttt{main})}{H_e^1(1,g,g,l,g,l)}\ \mathcal{R}_{10}$$

$$\frac{H_n^j(k,\bar{g_0},\bar{g_1},l_1,\bar{g_2},l_2)}{E^j(k,\bar{g_0},\bar{g_2})}\ \mathcal{R}_{11}$$

$$\frac{E^1(1,g_0,g_1), l \in L}{H^2_{e_2}(1,g_1,g_1,l,g_1,l)}\ \mathcal{R}_{12}$$

Fig. 5 Rules for lazy analysis of concurrent Boolean programs

different approach because it allows us to switch easily between the lazy and eager versions. Both versions are based on the rules shown in Fig. 5.

In the lazy version, the rules are applied in the following order: (i) The H relations are saturated for execution context k; (ii) Then the E relations are computed for k; (iii) then rules $\mathcal{R}_8$ and $\mathcal{R}_9$ are used to initialize the H relations for execution context $k+1$ and the process is repeated. In this way, the first step can be performed using the standard (sequential) reachability algorithm of MOPED. Theorem 2 allows us to find the reachable states directly from the E relations.

The eager version is implemented in a similar fashion, except that it uses a fixed set of E relations that include all possible global state changes. Once the H relations are computed, as described above, then the E relations are reinitialized using rule $\mathcal{R}_{11}$. Next, the following rule, which encodes the `Checker` phase, computes the set of reachable states (assuming that K is the given bound on the number of execution contexts).

$$\frac{E^1(K,\bar{g_0},\bar{g_1}) \qquad E^2(K,\bar{g_1},\bar{g_2}) \qquad g = \texttt{check}(\bar{g_0},\bar{g_2})}{\mathit{Reachable}(g)}\ \texttt{Checker}.$$

Our implementation supports any number of threads. It uses round-robin scheduling with a bound on the number of execution context per thread, as described in Sect. 2.3.

All of our experiments, discussed below, were performed on a 2.4GHz machine with 3.4GB RAM running Linux version 2.6.18-92.1.17.el5.

BlueTooth driver model. First, we report the results for a model of the BlueTooth driver, which has been used in several past studies [5, 21, 27]. The driver model can have multiple threads, where each thread requests the addition or the removal of devices from the system, and checks to see if a user-defined assertion can fail. We used this model to test the scalability of our tool with respect to the number of threads, as well as the number of execution contexts per thread. The results are shown in Fig. 6. The model has 8 shared global variables, at most 7 local variables per procedure, 5 procedures but no recursion, and 37 program statements.

It is interesting to note that the eager analysis is faster than the lazy analysis in some cases (when there are a large number of threads or execution contexts). The running times

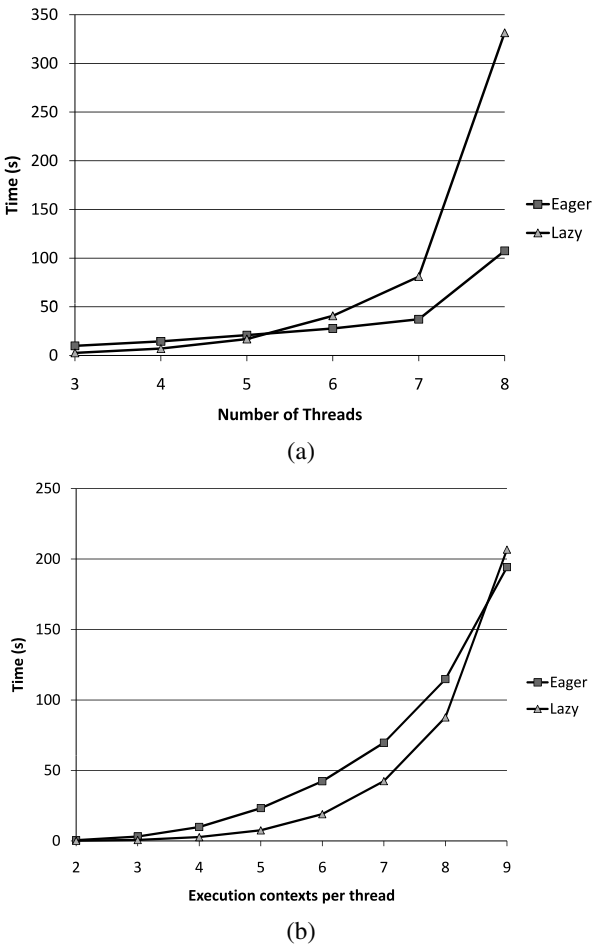


Fig. 6 Experiments with the BlueTooth driver model. Each thread tries to either start or stop the device. (**a**) Running time when the number of execution contexts per thread is fixed at 4. (**b**) Running time when the number of threads is fixed at 3

for symbolic techniques need not be proportional to the number of states explored: even when the eager analysis explores more behaviors than the lazy version, its running time is shorter because it is able to exploit more symmetry in the search space, and the resulting BDDs are small.

The graph in Fig. 6(b) shows the exponential dependence of running time with the number of execution contexts. The graph in Fig. 6(a) shows the expected linear dependence of running time with the number of threads, until the number of threads is 8. We believe that the sharp increase is due to BDDs getting large enough so that operations on them do not entirely fit inside the BDD-cache.

Binary search tree. We also measured the performance of our techniques on a model of a concurrent binary search tree, which was also used in [27]. (Because our model was hand-coded, and the model used in [27] was automatically extracted from Java code, our results are not directly comparable.) This model has a finite heap, and a thread either tries to insert a value, or search for it in the tree. The model has 72 shared global variables, at most 52 local variables per procedure, 15 procedures, and 155 program statements. The model uses recursion.

Threads		Execution contexts per thread				
Inserters	Searchers	2	3	4	5	6
1	1	6.1	21.6	84.5	314.8	1054.8
2	1	11.9	46.8	211.9	832.0	2995.6
2	2	14.1	64.4	298.0	1255.4	4432.1

Fig. 7 Lazy context-bounded analysis of the binary search tree model. The table reports the running time for various configurations in seconds

The eager version of the algorithm timed out on this model for most settings. This may be because the analysis has to consider symbolic operations on the heap, which results in huge BDDs. The results for the lazy version are reported in Fig. 7. They show trends similar to the BlueTooth driver model: a linear increase in running time according to the number of threads, and an exponential increase in running time according to the number of execution contexts per thread.

BEEM benchmark suite. The third set of experiments consisted of common concurrent algorithms, for which finite, non-recursive models were obtained from the BEEM benchmark suite [17]. We hand-translated some of SPIN models into the input language of MOPED. These models do not exploit the full capabilities of our tool because they all have a single procedure. We use these models for a more comprehensive evaluation of our tool. All examples that we picked use a large number of threads. As before, the eager version timed out for most settings, and we report the results for the lazy version.

The benchmark suite also has buggy versions of each of the test examples. The bugs were introduced by perturbing the constants in the correct version by ± 1 or by changing comparison operators (e.g., $>$ to $\geq$, or vice versa). Interestingly, the bugs were found within a budget of 2 or 3 execution contexts per thread. (Note that this may still involve multiple context switches.)

The results are reported in Fig. 8. To put the numbers in perspective, we also give the time required by SPIN to enumerate all reachable states of the program. These are finite-state models, meant for explicit-state model checkers; however, SPIN ran out of memory on three of the eight examples.

The CBA techniques presented in this paper, unlike explicit-state model checkers, do not look for repetition of states: if a state has been reached within k context switches, then it need not be considered again if it shows up after $k + i$ context switches. In general, for recursive programs, this is hard to check because the number of states that can arise after a context switch may be infinite. However, it would still be interesting to explore techniques that can rule out some repetitions.

Predicate-abstraction based Boolean programs. Our fourth set of experiments use the concurrent Boolean programs generated using the predicate abstraction performed by DDVERIFY [28]. We use this set of experiments to validate two hypotheses: first, most bugs manifest themselves in a few context switches; second, our tool, based on CBA, remains competitive with current verification tools when it is given a reasonable bound on the number of context switches.

First, we briefly describe how DDVERIFY operates. When given C source code, DDVERIFY performs predicate abstraction to produce an abstract model of the original program. This model is written out as a concurrent Boolean program and fed to the model checker BOPPO, or in the input language of SMV and fed to SMV. DDVERIFY uses SMV by default because it performs better then BOPPO on concurrent models [28]. If the model checker is able to prove the correctness of all assertions in the model, the entire process

Name	Inst	#gvars	#lvars	#Threads	#EC	Time (s)	SPIN (s)
Anderson N=6,ERROR=0	pos	11	4	6	2	52.46	OOM
Anderson N=6,ERROR=1	neg	11	4	6	2	54.90	OOM
Bakery N=4,MAX=7	pos	17	7	4	2	5.87	28.5
Bakery N=4,MAX=5	neg	17	7	4	2	13.88	44.2
Peterson N=4	pos	25	7	4	3	5.46	3.05
Peterson N=4,ERROR=1	neg	25	7	4	3	25.72	OOM
Msmie N=5,S=10,M=10	pos	23	1	20	2	47.94	31.0
Msmie N=5,S=10,M=10	neg	13	1	13	2	1.29	1.04

Fig. 8 Experiments on finite-state models obtained from the BEEM benchmark suite. The names, along with the given parameter values uniquely identify the program in the test suite. The *columns*, in order, report: the name; buggy (neg) or correct (pos) version; number of shared variables; number of local variables per thread; number of threads; execution context budget per thread; running time of our tool in seconds; and the time needed by SPIN to enumerate the entire state space

succeeds (no bugs). If the model checker returns a counterexample, then it is checked concretely, and if it is spurious, then the abstraction is refined to create a new abstract model and this repeats. DDVERIFY checks for a number of different properties on the source code separately. (The abstract models produced by DDVERIFY have a single procedure and very few local variables. Thus, these experiments do not exploit the full capabilities of CBA as well.)

We now describe the experimental setup. We chose 6 drivers among the ones provided with the distribution of DDVERIFY. For each driver, we chose some properties at random and let DDVERIFY run normally using SMV as its model checker, but we saved the Boolean programs that it produced at each iteration. For each driver and each property, we collected the SMV files and the Boolean programs produced during the last iteration. The experiments were conducted on these files. We gave our tool a budget of 2 threads and 4 execution contexts per thread. A scatter plot of the running times is shown in Fig. 9 and the aggregate times for proving all chosen properties for a given driver are reported in Fig. 10.

Two things should be noted from the results. First, whenever a model was buggy, our tool could find it within the budget given to it. This validates our hypothesis that bugs manifest in few context switches. Second, our tool was much faster than SMV on these benchmarks, with speedups of up to 120X; our tool was slower on only one example. As shown by the dotted line in Fig. 9, the median speedup was about 30X.

8 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 [21] for the case of two

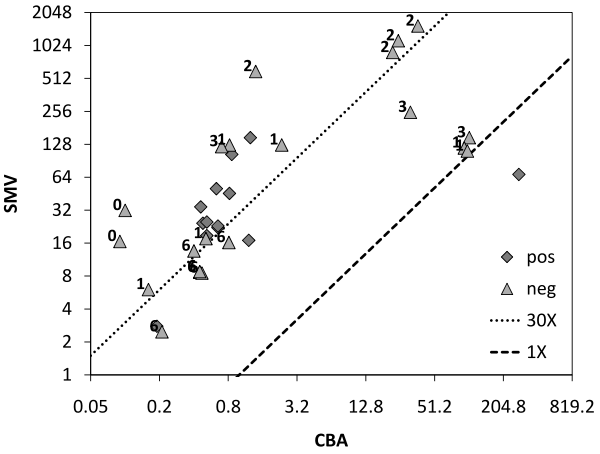


Fig. 9 Scatter plot of the running times of our tool (CBA) against SMV on the files obtained from DDVERIFY. Different dots are used for the cases when the files had a bug (neg) and when they did not have a bug (pos). For the "neg" dots, the number of context switches before a bug was found is shown alongside the dot. The median speedup was about 30X. *Lines* indicating 1X and 30X speedups are also shown as *dashed* and *dotted lines*, respectively

Name	Inst	Time (s)	SMV (s)	Speedup	# CS
applicom	pos	1.3	147.1	117.7	
	neg	33.0	147.1	15.1	[1, 3]
generic_nvram	pos	2.3	88.1	38.3	
gpio	pos	280.6	92.8	0.33	
	neg	106.8	299.6	2.8	[1, 6]
machzwd	pos	0.9	103.4	120.2	
	neg	85.6	4214.7	49.2	[0, 6]
nwbutton	pos	0.19	2.8	14.6	
	neg	0.88	26.1	29.7	[1, 6]
toshiba	pos	3.8	197.1	52.4	
	neg	192.8	243.43	1.3	[1, 6]

Fig. 10 Experiments on concurrent Boolean programs obtained from DDVERIFY. The *columns*, in order, report: the name of the driver; buggy (neg) or correct (pos) version, as determined by SMV; running time of our tool in seconds; the running time of SMV; speedup of our tool against SMV; and the range of the number of context switches after which a bug was found. Each *row* summarizes the time needed for checking multiple properties

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.

A symbolic algorithm for context-bounded analysis was presented recently by Suwimonteerabuth et al. [27]. An earlier algorithm by Qadeer and Rehof [20] required enumeration of all reachable global states at a context switch. Suwimonteerabuth et al. identify places where such an enumeration is not required, essentially by finding different abstract states that the program model cannot distinguish. This enables symbolic computation to some extent. However, in the worst case, the algorithm still requires enumeration of all reachable states.

Analysis of message-passing concurrent systems, as opposed to ones having shared memory, has been considered in [5]. 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 [8, 19]. 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.

Acknowledgements We would like to thank Daniel Kroening, Gerard Basler and Georg Weissenbacher for their help with using DDVERIFY. We would also like to thank Stefan Schwoon and Dejvuth Suwimonteerabuth for their support with MOPED.

Appendix A: Proofs

A.1 Proof of Theorem 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 $p_2, p_4, \ldots, p_{2K-2} \in P$ such that $\langle(1, p_0, p_2, \ldots, p_{2K-2}), c_0\rangle \Rightarrow_{\mathcal{P}_s} \langle(1, p_2, p_4, \ldots, p_{2K}), e_3\ d_K\ c_K\rangle$.

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 $r^s[k, p_1, \ldots, p_{k-1}, p_{k+1}, \ldots, p_K] \in \Delta_s$ be the rule $\langle(k, p_1, \ldots, p_{k-1}, p', p_{k+1}, \ldots, p_K), \gamma\rangle \hookrightarrow \langle(k, p_1, \ldots, p_{k-1}, p', p_{k+1}, \ldots, p_K), u\rangle$. 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 $r_{1\to2}$ stand for the rules that call from the first PDS to the second, and $r_{2\to3}$ 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. 11(a). Then the path shown in Fig. 11(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 $\langle(1, p_0, p_2, \ldots, p_{2K-2}), c_0\rangle$ to $\langle(1, p_2, p_4, \ldots, p_{2K}), e_3\ d_K\ c_K\rangle$ can be broken down as $\sigma = \sigma_A\ r_{1\to2}\ \sigma_B\ r_{2\to3}$. (This is because one must use the rules $r_{1\to2}$ and $r_{2\to3}$, in order, to push e_3 on the stack, after which no rules can fire.) Hence we must have the following (for some states $p_1, p_3, \ldots, p_{2K-1}$):

$$\begin{aligned}\langle(1, p_0, p_2, \ldots, p_{2K-2}), c_0\rangle &\Rightarrow_{\mathcal{P}_s}^{\sigma_A} \langle(K+1, p_1, p_3, \ldots, p_{2K-1}), c_K\rangle\\ &\Rightarrow_{\mathcal{P}_s}^{r_{1\to2}} \langle(1, p_1, p_3, \ldots, p_{2K-1}), d_0\ c_K\rangle\\ &\Rightarrow_{\mathcal{P}_s}^{\sigma_B} \langle(K+1, p_2, p_4, \ldots, p_{2K}), d_K\ c_K\rangle\\ &\Rightarrow_{\mathcal{P}_s}^{r_{2\to3}} \langle(1, p_2, p_4, \ldots, p_{2K}), e_3\ d_K\ c_K\rangle.\end{aligned}$$

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 ith state component, we must have the following:

(a)	(b)
$\langle p_0, c_0, d_0\rangle$	$\langle (1, p_0, p_2, \ldots, p_{2K-2}), c_0\rangle$
$\Rightarrow_1^{\sigma_1}$ $\langle p_1, c_1, d_0\rangle$	$\Rightarrow^{\sigma_1^s[1]}$ $\langle (1, p_1, p_2, p_4, \ldots, p_{2K-2}), c_1\rangle$
$\Rightarrow_2^{\sigma_2}$ $\langle p_2, c_1, d_1\rangle$	$\Rightarrow^{r_{\text{inc}}[1]}$ $\langle (2, p_1, p_2, p_4, \ldots, p_{2K-2}), c_1\rangle$
$\Rightarrow_1^{\sigma_3}$ $\langle p_3, c_2, d_1\rangle$	$\Rightarrow^{\sigma_3^s[2]}$ $\langle (2, p_1, p_3, p_4, \ldots, p_{2K-2}), c_2\rangle$
$\Rightarrow_2^{\sigma_4}$ $\langle p_4, c_2, d_2\rangle$	$\Rightarrow^{r_{\text{inc}}[2]}$ $\langle (3, p_1, p_3, p_4, \ldots, p_{2K-2}), c_2\rangle$
$\ldots$	$\ldots$
$\Rightarrow_1^{\sigma_{2K-1}}$ $\langle p_{2K-1}, c_K, d_{K-1}\rangle$	$\Rightarrow^{r_{\text{inc}}[K-1]}$ $\langle (K, p_1, p_3, p_5, \ldots, p_{2K-3}, p_{2K-2}), c_{K-1}\rangle$
$\Rightarrow_2^{\sigma_{2K}}$ $\langle p_{2K}, c_K, d_K\rangle$	$\Rightarrow^{\sigma_{2K-1}^s[K]}$ $\langle (K, p_1, p_3, p_5, \ldots, p_{2K-3}, p_{2K-1}), c_K\rangle$
	$\Rightarrow^{r_{\text{inc}}[K]}$ $\langle (K+1, p_1, p_3, p_5, \ldots, p_{2K-3}, p_{2K-1}), c_K\rangle$
	$\Rightarrow^{r_{1\to 2}}$ $\langle (1, p_1, p_3, p_5, \ldots, p_{2K-1}), d_0\ c_K\rangle$
	$\Rightarrow^{\sigma_2^s[1]}$ $\langle (1, p_2, p_3, p_5, \ldots, p_{2K-1}), d_1\ c_K\rangle$
	$\Rightarrow^{r_{\text{inc}}[1]}$ $\langle (2, p_2, p_3, p_5, \ldots, p_{2K-1}), d_1\ c_K\rangle$
	$\ldots$
	$\Rightarrow^{r_{\text{inc}}[K-1]}$ $\langle (K, p_2, p_4, p_6, \ldots, p_{2K-2}, p_{2K-1}), d_{K-1}\ c_K\rangle$
	$\Rightarrow^{\sigma_{2K}^s[K]}$ $\langle (K, p_2, p_4, p_6, \ldots, p_{2K-2}, p_{2K}), d_K\ c_K\rangle$
	$\Rightarrow^{r_{\text{inc}}[K]}$ $\langle (K+1, p_2, p_4, p_6, \ldots, p_{2K-2}, p_{2K}), d_K\ c_K\rangle$
	$\Rightarrow^{r_{2\to 3}}$ $\langle (1, p_2, p_4, p_6, \ldots, p_{2K-2}, p_{2K}), e_3\ d_K\ c_K\rangle$

Fig. 11 Simulation of a concurrent PDS run by a single PDS. For clarity, we write $\Rightarrow$ to mean $\Rightarrow_{\mathcal{P}_s}$ in (**b**)

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

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 Theorem 1

A PDS can have infinite number of configurations. Hence, sets of configurations are represented using automata [25]. 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 [25]. We call the algorithm from [25] *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 $(\mathcal{P}_1, \mathcal{P}_2, \ldots, \mathcal{P}_n)$, $|P_s| = K|P|^K$, $|\Delta_s| = K|P|^{K-1}|\Delta|$, $|Proc_s| = |Proc|$, where $\Delta = \bigcup_{i=1}^n \Delta_i$ and $|Proc| = \sum_{i=1}^n |Proc_i|$. To obtain the set of forward reachable configurations from $\langle p, e_1, e_2, \ldots, e_n\rangle$, we will solve *poststar*($\mathcal{A}$) for each $\mathcal{A}$ that represents the singleton set of configurations $\{\langle (1, p, p_2, \ldots, p_K), e_1\rangle\}$, i.e., $|P|^{K-1}$ separate calls to *poststar*. In the result, we can project out all configurations that do not have $(1, p_2, \ldots, p_K, p')$ as their state, for some p'. Directly

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

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

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 $\langle(1, p, p_2, \ldots, p_K), e_1\rangle$, we only want configurations of the form $\langle(1, p_2, \ldots, p_K, p'), u\rangle$. 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 $\langle(k, p'_2, p'_3, \ldots, p'_K, p'), \gamma\rangle$ 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 [25]: (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 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 $poststar_{\Delta_2}(poststar_{\Delta_1}(\mathcal{A}))$ is the same as the running time of $poststar_{\Delta}(\mathcal{A})$, 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 $\mathtt{k}$) 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 $\{\langle\bar{p}_1, c_1\rangle, \langle\bar{p}_2, c_2\rangle, \langle\bar{p}_2, c_3\rangle\}$, then we split it into two automata representing the sets $\{\langle\bar{p}_1, c_1\rangle\}$ and $\{\langle\bar{p}_2, c_2\rangle, \langle\bar{p}_2, c_3\rangle\}$, respectively. Observation (i) shows that this splitting only increases the running time.

Table 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 "$\lvert \rightarrow \rvert$" 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 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 $(i-1)|P||\Delta||Proc|$ transitions, taking time $2(i-1)|P|^2|\Delta||Proc|$. Each result is split into $|P|$ different automata, each with at most $i|P||Proc|$ states. The reader can inductively verify the correctness of the table.

Thus, this requires a total running time of $\mathcal{O}(K|P|^{K+1}|\Delta||Proc|)$. Next, we use the rules in Δ_{call} and repeat the above process for the last K of the sequence $\mathcal{S}$. 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 $(1, p, p_2, \ldots, p_K)$, and after performing the computation of Table 1, we obtain an automaton $\mathcal{A}$ that has the single state $(1, p'_1, \ldots, p'_K)$ 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 $(2, p_2, p'_2, p'_3, \ldots, p'_K)$. Thus, splitting is not necessary, and the time required to process each of the $|P|^{K-1}$ automata obtained from Table 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 Theorem 2

For proving Theorem 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. 12 (where n_0 is the entry point of the thread).

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

$$
\begin{array}{c}
n_0 : (1, g_1, g_2, \ldots, g_k, g_{k+1}, \ldots, g_K, l_0) \\
\Downarrow \\
n_1 : (1, g'_1, g_2, \ldots, g_k, g_{k+1}, \ldots, g_K, l_1) \\
\Downarrow \\
n_1 : (2, g'_1, g_2, \ldots, g_k, g_{k+1}, \ldots, g_K, l_1) \\
\Downarrow \\
n_2 : (2, g'_1, g'_2, \ldots, g_k, g_{k+1}, \ldots, g_K, l_2) \\
\Downarrow \\
\vdots \\
\Downarrow \\
n_{k-1} : (k-1, g'_1, g'_2, \ldots, g'_{k-1}, g_k, g_{k+1}, \ldots, g_K, l_{k-1}) \\
\Downarrow \\
n_k : (k, g'_1, g'_2, \ldots, g'_k, g_{k+1}, \ldots, g_K, l_k)
\end{array}
$$

Fig. 12 An execution in T_1^s

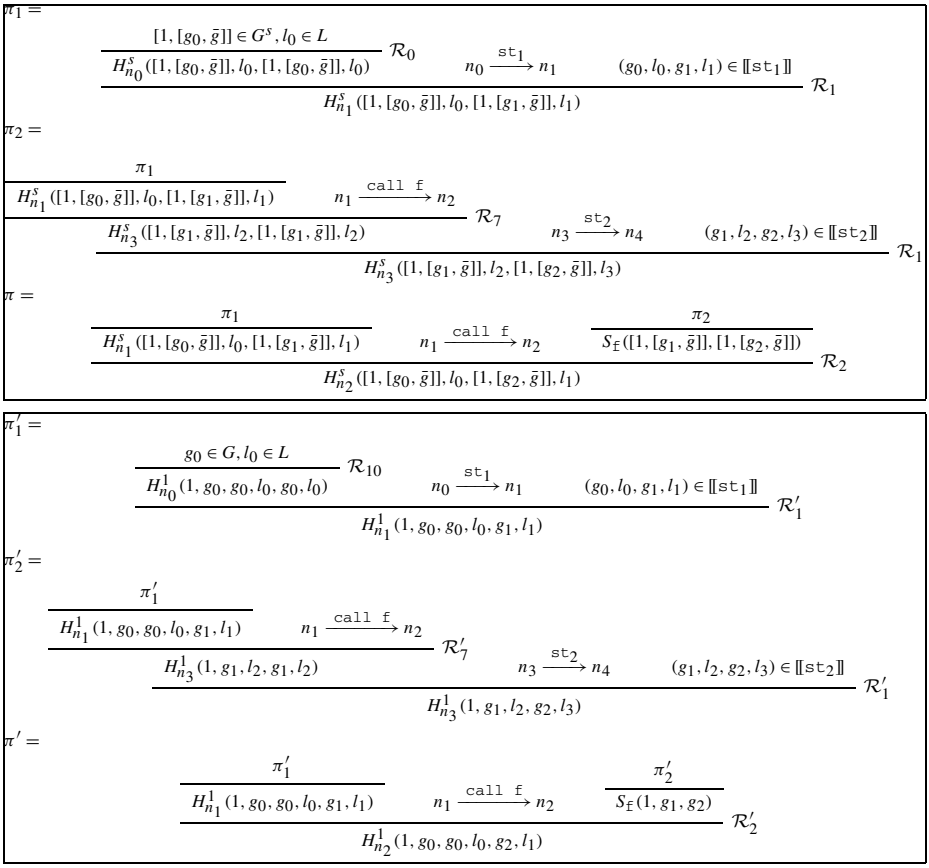


Fig. 13 An example of converting from proof π to proof π'. For brevity, we use `st` to mean a statement in the thread T_1 (and not its translated version in T_1^s)

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 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: $H^s_n([k_1,\bar{g_1}],l_1,[k_2,\bar{g_2}],l_2)$, which semantically means that if the data state at $\text{ep}(n)$ was $([k_1,\bar{g_1}],l_1)$, then the data state at n can be $([k_2,\bar{g_2}],l_2)$; and the summary relation would be $S^s_{\texttt{f}}([k_1,\bar{g_1}],[k_2,\bar{g_2}])$. For a statement `st` in T_1, its translation in T_1^s encodes the transformer: $\{((k,g_1,\ldots,g_k,\ldots,g_K),l,(k,g_1,\ldots,g'_k,\ldots,g_K),l') \mid (g_k,l,g'_k,l') \in \texttt{st}\}$. Additionally, one has a self-loop edge associated with a transformer that increments the value of `k`: $\{([k,\bar{g}],l,[k+1,\bar{g}],l) \mid 1 \le k \le K\}$. 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. 13 for a program path $n_0 \xrightarrow{\texttt{st}_1} n_1 \xrightarrow{\texttt{call f}} n_2$, where the call to `f` takes the path $n_3 \xrightarrow{\texttt{st}_2} n_4$. Let $(g_1,\ldots,g_{k+i})|_k = (g_1,\ldots,g_k)$.

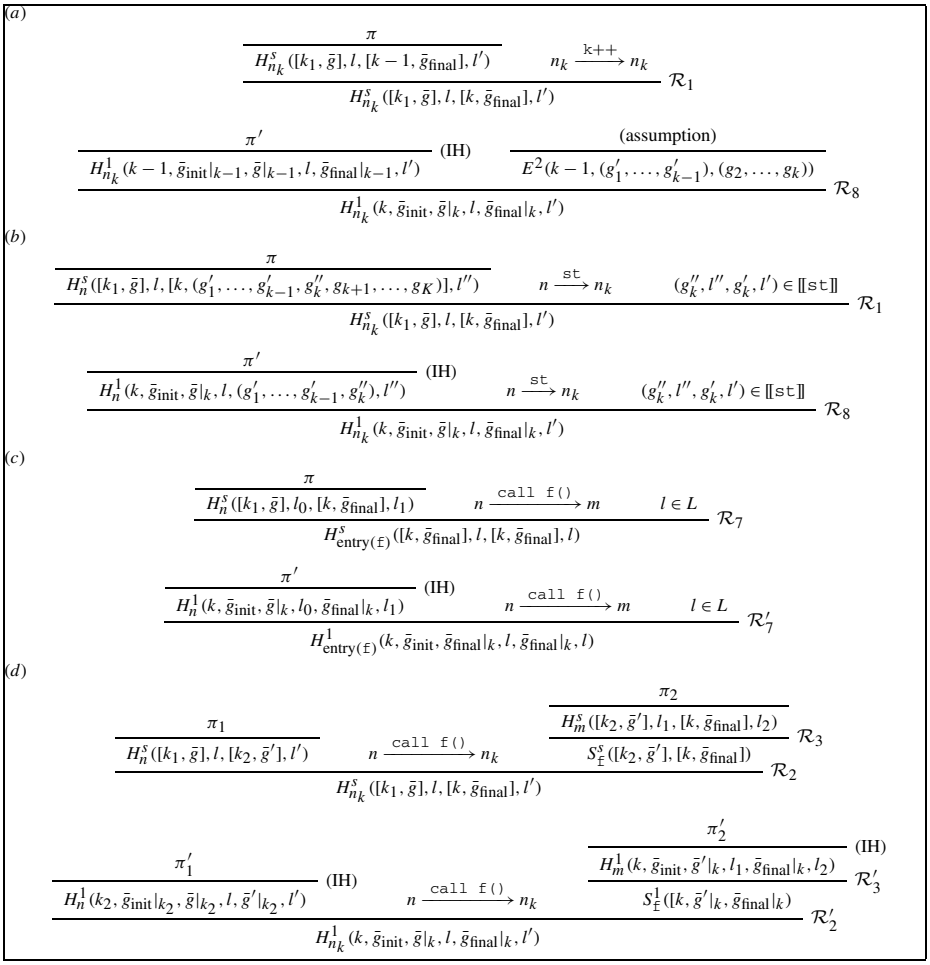


Fig. 14 Simulation of run ρ using rules in Fig. 5. In case (a), $g_k = \texttt{check}(\bar{g}_{\text{init}}|_{k-1}, (g_2, \ldots, g_k))$ and $g'_k = g_k$ (because ρ does not edit these set of variables). In case (d), exitnode(m) holds, $\texttt{f} = \text{proc}(m)$, $k_1 \le k_2 \le k$, the $k_2 + 1$ to k components of $\bar{g}'$ are $(g_{k_2+1}, \ldots, g_k)$ because it arises when $\texttt{k} = k_2$, and the $k_1 + 1$ to k components of $\bar{g}$ are $(g_{k_1+1}, \ldots, g_k)$ for the same reason

The induction hypothesis is as follows: given ρ, as shown in Fig. 12, if there is a proof tree π that derives $H^s_{n_k}([k_1, \bar{g}], l, [k, (g'_1, \ldots, g'_k, g_{k+1}, \ldots, g_K)], l')$ then one can derive $H^1_{n_k}(k, (g_1, \ldots, g_k), \bar{g}|_k, l, (g'_1, \ldots, g'_k), l')$. Note that in this case, the last $(K - k_1)$ components of $\bar{g}$ must be $(g_{k_1+1}, \ldots, g_K)$ because T^s_1 could not have modified them. We have already proved the base case above. Fix $\bar{g}_{\text{init}} = (g_1, \ldots, g_k)$ and $\bar{g}_{\text{final}} = (g'_1, \ldots, g'_k, g_{k+1}, \ldots, g_K)$.

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 $\texttt{k}$. All these cases, and the way to obtain π' are shown in Fig. 14.

One can prove a similar result for T^s_2. Note that $H^1_{n_k}(k, \bar{g}_{\text{init}}, \bar{g}|_k, l, \bar{g}_{\text{final}}|_k, l')$ implies $E^1(k, \bar{g}_{\text{init}}, \bar{g}_{\text{final}}|_k)$. 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.

Going the other way is 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 $E^2(1, g_1', g_2)$, $E^2(2, (g_1', g_2'), (g_2, g_3)), \ldots, E^2(i, (g_1', \ldots, g_i'), (g_2, \ldots, g_{i+1}))$. 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. 14 in the opposite direction to prove the reverse direction of the theorem.

References

1. Ball T, Majumdar R, Millstein T, Rajamani SK (2001) Automatic predicate abstraction of C programs. In: PLDI
2. Ball T, Rajamani S (2000) Bebop: a symbolic model checker for Boolean programs. In: SPIN
3. Berger F, Schwoon S, Suwimonteerabuth D (2005) jMoped. http://www7.in.tum.de/tools/jmoped/
4. Bouajjani A, Fratani S, Qadeer S (2007) Context-bounded analysis of multithreaded programs with dynamic linked structures. In: CAV
5. Chaki S, Clarke EM, Kidd N, Reps TW, Touili T (2006) Verifying concurrent message-passing C programs with recursive calls. In: TACAS
6. Cousot P, Halbwachs N (1978) Automatic discovery of linear restraints among variables of a program. In: POPL
7. Henzinger T, Jhala R, Majumdar R, Sutre G (2002) Lazy abstraction. In: POPL
8. Henzinger TA, Jhala R, Majumdar R (2004) Race checking by context inference. In: PLDI
9. Kiefer S, Schwoon S, Suwimonteerabuth D. Moped. http://www.fmi.uni-stuttgart.de/szs/tools/moped/
10. Knoop J, Steffen B (1992) The interprocedural coincidence theorem. In: CC
11. Lal A, Kidd N, Reps T, Touili T (2007) Abstract error projection. In: SAS
12. Lal A, Touili T, Kidd N, Reps T (2007) Interprocedural analysis of concurrent programs under a context bound. TR-1598, University of Wisconsin, July 2007
13. Lal A, Touili T, Kidd N, Reps T (2008) Interprocedural analysis of concurrent programs under a context bound. In: TACAS
14. Müller-Olm M, Seidl H (2004) Precise interprocedural analysis through linear algebra. In: POPL
15. Murphy B, Lam M (2000) Program analysis with partial transfer functions. In: PEPM
16. Musuvathi M, Qadeer S (2007) Iterative context bounding for systematic testing of multithreaded programs. In: PLDI
17. Pelánek R (2007) BEEM: Benchmarks for explicit model checkers. In: SPIN
18. Qadeer S, Rajamani S (2005) Deciding assertions in programs with references. Technical Report MSR-TR-2005-08, Microsoft Research, Redmond, January 2005
19. Qadeer S, Rajamani SK, Rehof J (2004) Summarizing procedures in concurrent programs. In: POPL
20. Qadeer S, Rehof J (2005) Context-bounded model checking of concurrent software. In: TACAS
21. Qadeer S, Wu D (2004) KISS: keep it simple and sequential. In: PLDI
22. Ramalingam G (2000) Context-sensitive synchronization-sensitive analysis is undecidable. In: TOPLAS
23. Reps T, Horwitz S, Sagiv M (1995) Precise interprocedural dataflow analysis via graph reachability. In: POPL
24. Reps T, Schwoon S, Jha S, Melski D (2005) Weighted pushdown systems and their application to interprocedural dataflow analysis. In: SCP, vol 58
25. Schwoon S (2002) Model-checking pushdown systems. PhD thesis, Technical University of Munich, Munich, Germany, July 2002
26. Sharir M, Pnueli A (1981) Two approaches to interprocedural data flow analysis. In: Program flow analysis: theory and applications. Prentice-Hall, New York
27. Suwimonteerabuth D, Esparza J, Schwoon S (2008) Symbolic context-bounded analysis of multithreaded Java programs. In: SPIN
28. Witkowski T, Blanc N, Kroening D, Weissenbacher G (2007) Model checking concurrent linux device drivers. In: ASE