Interprocedural Context-Unbounded Program Analysis Using Observation Sequences

PEIZUN LIU and THOMAS WAHL, Northeastern University
THOMAS REPS, University of Wisconsin–Madison and GrammaTech Inc.

A classical result by Ramalingam about synchronization-sensitive interprocedural program analysis implies that reachability for concurrent threads running recursive procedures is undecidable. A technique proposed by Qadeer and Rehof, to bound the number of context switches allowed between the threads, leads to an incomplete solution that is, however, believed to catch "most bugs" in practice, as errors tend to occur within few contexts. The question of whether the technique can also prove the absence of bugs at least in some cases has remained largely open.

Toward closing this gap, we introduce in this article the generic verification paradigm of *observation sequences* for resource-parameterized programs. Such a sequence observes how increasing the resource parameter affects the reachability of states satisfying a given property. The goal is to show that increases beyond some "cutoff" parameter value have no impact on the reachability—the sequence has *converged*. This allows us to conclude that the property holds for all parameter values.

We applied this paradigm to the context-*unbounded* program analysis problem, choosing the resource to be the number of permitted thread context switches. The result is a partially correct interprocedural reachability analysis technique for concurrent shared-memory programs. Our technique may not terminate but is able to both refute and prove context-unbounded safety for such programs. We demonstrate the effectiveness and efficiency of the technique using a variety of benchmark programs. The safe instances cannot be proved safe by earlier, context-bounded methods.

CCS Concepts: • **Theory of computation** → **Logic and verification**; **Abstraction**; • **Software and its engineering** → *Formal software verification*;

Additional Key Words and Phrases: Interprocedural analysis, context bound, concurrency, recursion, stack

ACM Reference format:
Peizun Liu, Thomas Wahl, and Thomas Reps. 2020. Interprocedural Context-Unbounded Program Analysis Using Observation Sequences. *ACM Trans. Program. Lang. Syst.* 42, 4, Article 16 (December 2020), 34 pages.
https://doi.org/10.1145/3418583

This article is an extended and improved version of an earlier conference publication [37].
P. Liu and T. Wahl were supported by the U.S. National Science Foundation under grant 1253331 (https://www.nsf.gov/awardsearch/showAward?AWD_ID=1253331). T. Reps was supported by a gift from Rajiv and Ritu Batra, by the Air Force Research Laboratory under DARPA MUSE grant FA8750-14-2-0270 and DARPA STAC grant FA8750-15-C-0082, and by the Office of Naval Research under grants N00014-17-1-2889 and N00014-19-1-2318.
Authors' addresses: P. Liu and T. Wahl, Khoury College of Computer Sciences, Northeastern University, 360 Huntington Avenue, Boston, MA 02115; emails: {liu.pei, t.wahl}@northeastern.edu; T. Reps, Computer Sciences Department, University of Wisconsin–Madison, 1210 West Dayton Street, Madison, WI 53706; email: reps@cs.wisc.edu.

0164-0925/2020/12-ART16 $15.00
https://doi.org/10.1145/3418583

1 INTRODUCTION

Precise reasoning about systems of concurrent threads executing procedures with unbounded call stacks requires an analysis that records the state of the thread before each procedure call to resume execution with the correct stack frame upon return from the call. A synchronization-sensitive analysis, however, tracks the interaction between parallel threads. A landmark result by Ramalingam [47] shows that a precise analysis tracking procedure calls and thread synchronization is impossible algorithmically. This implies that there is no sound and complete reachability method for concurrent threads with shared-memory communication and recursive procedure calls.

In response to this costly (from a program analysis point of view) marriage between recursive procedures and concurrency, extensive research has been conducted on various forms of approximate solutions [9, 14, 23, 29, 33, 34, 41]. These methods sidestep the undecidability by imposing restrictions on the program behavior [9, 29], the synchronization mechanisms [14, 23], the way of switching contexts [33, 34], or the stack depth [41]. Less work exists on the alternative approach of using semi-decision procedures, which may return "unknown" or fail to terminate. For example, Prabhu et al. [43] propose such a procedure to construct a proof of correctness via combining context-bounded analysis (CBA) and k-induction. It generalizes the information obtained from CBA to construct a proof of correctness.

This and other prior approaches exploit, in one form or another, the early work by Qadeer and Rehof [45], who showed that decidability of synchronization-sensitive interprocedural program analysis can be restored by imposing a *context switch bound* k on the analysis: along any path, the scheduler can switch control between threads at most k times. This gives rise to a principled error-detection method: we algorithmically investigate the program for an increasing context bound k until a bug is found. If that does not happen, empirical evidence suggests that the program may in fact be bug-free: concurrency errors tend to occur within a few context switches [40, 46]. Notwithstanding such evidence, it is of course highly desirable to confirm with certainty that the absence of bugs for small k implies their unreachability in general. We call this challenge the *Context-UnBounded Analysis* problem, CUBA for short, for fixed-thread concurrent recursive procedures communicating via shared memory.

Toward solving this challenge, we first propose in this article a generic approach, the verification paradigm of *observation sequences* (OSs). This paradigm applies to programs $\mathcal{P}$ that are parameterized by the amount k available of some program "resource" (broadly construed), such as the number of running threads, the capacity of some data structure, or the number of permitted thread contexts along executions. Given a property C, the strategy to decide whether C holds in $\mathcal{P}$ for all values of parameter k is to analyze $\mathcal{P}$ for increasing values $k = 0, 1, \ldots$. For each value, we compute an *observation* O_k about $\mathcal{P}$, such as a projection of the set of reachable states. The goal then is to demonstrate that the OS $(O_k)_{k=0}^{\infty}$ *converges* when reaching some parameter value k_0, i.e., that increasing k beyond k_0 has no further impact on the "reachable observations" (i.e., the observations about reachable states). If that is the case and the observations found in O_{k_0} do not reveal any property violation, we can conclude that C holds for all k.

By design, OSs are *monotone*, which entails a number of properties useful for convergence detection. For instance, monotone sequences over a finite domain always converge. An example is a sequence that observes in O_k the set of program locations reachable with resource bound k. In general, however, convergence is neither guaranteed nor is it easy to detect if the sequence does converge. In particular, since observation O_k is an abstraction of the system behavior for bound k, the sequence may exhibit *stuttering*: its elements may form a "plateau" (i.e., for some k, $O_k = O_{k+1} \subsetneq O_{k+2}$). The challenge for applying the OS paradigm is often to distinguish sequence stuttering from convergence.

The main technical contribution in this work is to apply the OS paradigm to the CUBA problem. To this end, we consider observations $\mathcal{T}(R_k)$ about the set R_k of states reachable under context bound k in a constant-thread concurrent pushdown system (CPDS). The sequence $(R_k)_{k=0}^{\infty}$ is naturally monotone in k: increasing the context bound cannot decrease the set of reachable states. Given a monotone function $\mathcal{T}$, the sequence of observations $(\mathcal{T}(R_k))_{k=0}^{\infty}$ is then also monotone.

To design a suitable function $\mathcal{T}$, we recall the preceding note that OSs over a finite domain are guaranteed to converge. We can project the sets R_k to a finite domain by mapping each global state to what we call the *visible*-state projection, which includes only the top symbol of each stack. Because the stack alphabet is finite, the domain of this projection is a finite set of tuples. The OS $(\mathcal{T}(R_k))_{k=0}^{\infty}$ of visible states therefore is (monotone and hence) guaranteed to converge. Most reachability properties, including assertions added to a program, are formulated only over visible states and can thus be expressed over the visible-state projection.

But there is no free lunch: because the CUBA problem is undecidable, it is clear that detecting convergence of the visible-state OS must be "hard": it is generally frustrated by the stuttering phenomenon. One of the technical contributions of this work is a technique to distinguish stuttering from convergence. The insight is that once we have reached a plateau (i.e., a succession of equal sequence members $\mathcal{T}(R_k) = \ldots = \mathcal{T}(R_{k+i})$), the *first* new visible state (if any) that shows up later in the sequence must have a special form: it must be triggered by particular stack operations. We call such visible states *generators*. Due to their special form, reachability of new generator states is easier to decide than reachability for arbitrary visible states. The theorem then is that once we reach a plateau *and* we can rule out the reachability of new generator states, the sequence has converged and the CUBA problem is solved.

Before we move on to discussing the contributions of this work, we consider an alternative (that seems to suggest itself) to the approach proposed in this article: given a state set abstraction function such as $\mathcal{T}$, why not interpret the program entirely abstractly, by designing an abstract version of the program's transition relation, known as an *abstract transformer*, and then compute an abstract fixed point of this transformer [17]? If the abstract state set is finite, such a fixed point is guaranteed to exist. It would soundly overapproximate the set of reachable abstract states and thus be able to prove safety properties of the program.

The paradigm proposed in this article does not follow this alternative approach for two reasons. The first is that our goal is to be able to not only prove but also refute safety properties (i.e., detect errors) of the given input programs. Because an abstract fixed point overapproximates the set of reachable abstract states, such a refutation requires detection of spurious errors, potentially followed by a refinement loop. In contrast, our method is exact with respect to the given pushdown program; there is no refinement. The only way it may fail to deliver is by not terminating. Second, we note that an abstract transformer must of course be defined *and implemented* first; mistakes here can jeopardize the soundness of the approach. In contrast, our method can be used on top of an existing systematic testing approach that directly explores the reachable state space of the pushdown system (PDS). This ability, we expect, will increase the chances for the paradigm to be used in new practical contexts.

Toward practical implementation of these techniques to solve the CUBA problem, we face another difficulty: because the sets R_k can be infinite, they must be represented symbolically. The data structure proposed by Bouajjani et al. [11] for this purpose and later used in other works [45, 48] is the pushdown store automaton (PSA). These automata were shown to have worst-case size exponential in k. Consequently, their construction can be expensive in practice, even if we later project each R_k to a finite set such as $\mathcal{T}(R_k)$. We thus have to find alternative ways of maintaining these sets.

To this end, we observe that while the set $\bigcup_{k=1}^{\infty} R_k$ of all reachable global states, an infinite union, is often infinite, the sets R_k of states reachable with context bound k may well be finite. We call this condition *finite-context reachability* (FCR). Because our approach based on OSs is iterative and always maintains only a set of global states reachable under the current context bound k, FCR allows us to represent sets of states using compact data structures for finite sets, such as BDDs or even extensional lists or sets. This facilitates implementing the technique and makes operations such as equivalence checks vastly more efficient than using PSAs.

The question that remains is this: how do we decide whether FCR holds for a given program $\mathcal{P}$? We prove in this article a sufficient condition: if, for a *sequential* PDS, the set of states reachable from *all stacks of size* ≤ 1 is finite, then the set of states reachable from *any* one*state with arbitrary stack size* is finite as well. By induction, then, if this condition holds for all threads' pushdown programs, all sets R_k are finite.

We conclude this article with experiments on concurrent procedural programs, most featuring unbounded recursion, many used in previous related work, so that we can compare the results. We demonstrate on these programs the effectiveness and relative efficiency of our proposed techniques. The main insights from our experiments are the following: our benchmark programs (i) exhibit fairly small context bounds k_0 at which the visible-states OS $(\mathcal{T}(R_k))_{k=0}^{\infty}$ converges, thus allowing us to prove their safety, and in fact in about the same or less time than previous context-*bounded* methods, and (ii) almost all satisfy the FCR condition, which facilitates the analysis.

In summary, we introduce in this article the simple yet elegant paradigm of OSs, for the analysis of resource-parameterized programs. In addition to the CUBA problem, this paradigm has been applied successfully to the analysis of FIFO-communicating distributed systems [39] (where the resource parameter is the FIFO queue bound), and for lifting delay-bounded scheduling [19] to the unbounded case. By their nature, techniques based on OSs find the smallest resource value k for which an error occurs or the sequence converges, as the case may be. We have used this paradigm here to derive a partial technique to decide reachability in concurrent procedural programs with possibly unbounded recursion, an undecidable problem. Although our technique may not terminate, in contrast to context-bounded methods, it can both refute and prove safety properties.

Outline of this article. Section 2 formalizes the program model and the problem we are tackling in this article. Section 3 introduces the (generic) paradigm of OSs and examines a series of its properties (with occasional references to the CUBA problem). Section 4 applies the paradigm of OSs to CUBA, resulting in a partial solution to this problem. Section 5 explores the special case that our CPDS computational model is derived from concurrent Boolean programs, which exhibit (rich) control flow and where local thread states are defined by values of variables. Exploiting such additional information can strengthen our technique. Section 6 presents a sufficient condition for the finiteness of the set of reachable states *within one thread context*. If true, this property allows us to use explicit-state data structures to store such states, facilitating the implementation and rendering it more efficient. Section 7 discusses the implementation of our technique in detail and presents an empirical evaluation. We end the article with a discussion of related work in Section 8 and our outlook into related open problems in Section 9.

2 REACHABILITY ANALYSIS IN THE CPDs

This article is primarily about CPDSs, a lossless encoding of concurrent Boolean programs, which in turn result from predicate abstractions of source code. In this section, we formalize concurrent Boolean programs, CPDSs, and the context-(un)bounded reachability problem.

```
proc    ::= void ident (ident*) { [decl;]* [label: stmt;]* }
decl    ::= decl ident := const
stmt    ::= seqstmt
          | wait (expr)
          | while (expr) { stmt }
          | if (expr) { stmt } else { stmt }
          | ident (expr*)
          | return
          | stmt; stmt

seqstmt ::= skip
          | goto label
          | assert (expr)
          | ident := expr
expr    ::= expr bop expr
          | !expr | (expr)
          | const | ident | ⋆
bop     ::= '&&' | '||' | '=' | '! ='
const   ::= 0 | 1
```

Fig. 1. The syntax of procedures in CBPs. An *ident*ifier is a string and used to name variables and procedures; a *label* is simply a natural number representing the *pc*.

2.1 Concurrent Boolean Programs

A concurrent Boolean program consists of a collection of Boolean procedures running concurrently and communicating via *shared* variables, and a main procedure, which spawns child threads via statements of the form **thread_create(***proc***)**, where *proc* specifies the entry procedure of the spawned thread. Throughout the article, we assume that all work of interest is done by child threads and hence omit the main thread in our analysis. The execution semantics is interleaved and can be formally defined via a translation to CPDSs, which is well studied and omitted here. We do, however, present the syntax of a language for procedures in Figure 1 and briefly summarize the translation at the end of Section 2.2.2.

A procedure *decl*ares local Boolean variables, followed by a list of labeled statements. We assume that each labeled statement is executed atomically. We illustrate the intuition behind some non-standard statements of Boolean procedures. Statement **wait** blocks the execution of the current procedure until the condition represented in *expr* holds (only meaningful in a multi-threaded context and only if *expr* refers to at least one shared variable). Procedure arguments are passed by value. Statement **skip** increments the program counter (*pc*); **goto** *label* sets the *pc* to the given label; **assert** defines assertions for verification. Non-terminal *expr* denotes a Boolean expression over shared and local program variables, constants, and the choice symbol ⋆ ; the latter nondeterministically evaluates to **0** or **1**. Figure 2 illustrates a two-thread Boolean program.

2.2 Concurrent Pushdown Systems

2.2.1 Pushdown Systems. A (sequential) *pushdown system* is a tuple (Q, Σ, Δ, q^I) consisting of the set of *shared states* Q, the *stack alphabet* Σ, the *pushdown program* Δ, and the *initial shared state* $q^I \in Q$. Set Σ does not contain the empty word symbol ε. The set Δ of *actions* is partitioned into $\Delta = \Delta_{\text{pop}} \cup \Delta_{\text{over}} \cup \Delta_{\text{push}}$, consisting of *pop*, *overwrite*, and *push* actions, respectively; we have

$$\begin{array}{lcll} \Delta_{\text{pop}} & \subseteq & (Q \times \Sigma) \times (Q \times \{\varepsilon\}) & \\ \Delta_{\text{over}} & \subseteq & (Q \times \Sigma) \times (Q \times \Sigma) & \cup \quad (Q \times \{\varepsilon\}) \times (Q \times \{\varepsilon\}) \\ \Delta_{\text{push}} & \subseteq & (Q \times \Sigma) \times (Q \times \Sigma \times \Sigma) & \cup \quad (Q \times \{\varepsilon\}) \times (Q \times \Sigma). \end{array}$$

Intuitively, the left-hand sides of the preceding "∪" represent actions on a non-empty stack and the right-hand sides on the empty stack (except for *pop*). We write $\langle q \,|\, \sigma\rangle \to \langle q' \,|\, \sigma'\rangle$ for $(\langle q \,|\, \sigma\rangle, \langle q' \,|\, \sigma'\rangle) \in \Delta$.

Semantics. A *state* of a PDS is an element of $Q \times \Sigma^*$, written in angle brackets: $\langle q \,|\, w\rangle$; state $c^I = \langle q^I \,|\, \varepsilon\rangle$ is *initial*. Actions cause states to change; we extend relation $\to$ to states to capture *steps* of the PDS:

```
1 decl x := 1;

  void foo() {
2   if (⋆)
3     foo();
4   while (x) {}
5   x := 1;
  }

  void bar() {
6   if (⋆)
7     bar();
8   while (!x) {}
9   x := 0; assert(x = 0);
  }
```

$$
\Delta_1 = \{\quad
\begin{array}{llll}
f_{2a}: & \langle q_x \,|\, 2\rangle & \to & \langle q_x \,|\, 3\rangle \\
f_{2b}: & \langle q_x \,|\, 2\rangle & \to & \langle q_x \,|\, 4\rangle \\
f_3: & \langle q_x \,|\, 3\rangle & \to & \langle q_x \,|\, 24\rangle \\
f_{4a}: & \langle q_1 \,|\, 4\rangle & \to & \langle q_1 \,|\, 4\rangle \\
f_{4b}: & \langle q_0 \,|\, 4\rangle & \to & \langle q_0 \,|\, 5\rangle \\
f_5: & \langle q_x \,|\, 5\rangle & \to & \langle q_1 \,|\, \varepsilon\rangle
\end{array}
\quad\}
$$

$$
\Delta_2 = \{\quad
\begin{array}{llll}
b_{6a}: & \langle q_x \,|\, 6\rangle & \to & \langle q_x \,|\, 7\rangle \\
b_{6b}: & \langle q_x \,|\, 6\rangle & \to & \langle q_x \,|\, 8\rangle \\
b_7: & \langle q_x \,|\, 7\rangle & \to & \langle q_x \,|\, 68\rangle \\
b_{8a}: & \langle q_0 \,|\, 8\rangle & \to & \langle q_0 \,|\, 8\rangle \\
b_{8b}: & \langle q_1 \,|\, 8\rangle & \to & \langle q_1 \,|\, 9\rangle \\
b_9: & \langle q_x \,|\, 9\rangle & \to & \langle q_0 \,|\, \varepsilon\rangle
\end{array}
\quad\}
$$

Fig. 2. A CBP (left) consisting of two procedures foo and bar with shared Boolean variable x. Thread 1 executes procedure foo; thread 2 executes procedure bar. We formalize this program as a CPDS (defined in Section 2.2.2) $\mathcal{P}^2 = \{\mathcal{P}_1, \mathcal{P}_2\}$ with $\mathcal{P}_i = (Q, \Sigma_i, \Delta_i, q^I)$ for $i = 1, 2$, $Q = \{q_0, q_1\}$ (where the states q_0 and q_1 correspond to program variable x having the values 0 and 1, respectively), $\Sigma_1 = \{2, 3, 4, 5\}$, $\Sigma_2 = \{6, 7, 8, 9\}$; Δ_1 and Δ_2 are shown on the right. The initial state is $q^I = \langle q_1 \,|\, 2, 6\rangle$. Actions in the figure that mention symbol x exist for $x = 0$ and for $x = 1$.

(a) Given a state $c = \langle q \,|\, \sigma_1..\sigma_z\rangle$ (i.e., with a non-empty stack of size $z \geq 1$ where σ_1 is the "top," we have $c \to c'$, where the successor state c' depends on the action as follows:
- $\langle q \,|\, \sigma_1\rangle \to \langle q' \,|\, \varepsilon\rangle \in \Delta_{\text{pop}}$: If $z = 1$, then $c' = \langle q' \,|\, \varepsilon\rangle$; otherwise, $c' = \langle q' \,|\, \sigma_2..\sigma_z\rangle$. This action **pops** σ_1 off the stack (modeling a terminating procedure).
- $\langle q \,|\, \sigma_1\rangle \to \langle q' \,|\, \sigma'\rangle \in \Delta_{\text{over}}$ ($\sigma' \in \Sigma$): $c' = \langle q' \,|\, \sigma'\sigma_2..\sigma_z\rangle$. This action **overwrites** σ_1 by σ' (modeling the execution of a statement in the currently running procedure).
- $\langle q \,|\, \sigma_1\rangle \to \langle q' \,|\, \rho_0\rho_1\rangle$ ($\rho_0, \rho_1 \in \Sigma$): $c' = \langle q' \,|\, \rho_0\rho_1\sigma_2..\sigma_z\rangle$. This action **overwrites** σ_1 by ρ_1 and **pushes** ρ_0 on the stack (modeling a procedure call).

(b) Given a state $c = \langle q \,|\, \varepsilon\rangle$ (i.e., with the empty stack), the only enabled actions are of the form $\langle q \,|\, \varepsilon\rangle \to \langle q' \,|\, \varepsilon\rangle \in \Delta_{\text{over}}$ and of the form $\langle q \,|\, \varepsilon\rangle \to \langle q' \,|\, \sigma\rangle \in \Delta_{\text{push}}$ ($\sigma \in \Sigma$). The former (an overwrite) changes c to $c' = \langle q' \,|\, \varepsilon\rangle$; the latter (a push) changes c to $c' = \langle q' \,|\, \sigma\rangle$.

No other action is enabled in state c. As can be seen from the semantics, our definition of PDS allows at most two stack symbols to change per step. This restriction is motivated by the intended application to model procedure call stacks. Other PDSs can be inexpensively converted into this normal form, at the mild cost of introducing a few additional states [48, Thm. 3.1].

Also note that the stack is initially empty. When using stacks to model procedure calls, there is typically a main thread that creates worker threads and passes to them the name of the procedure to be executed. In examples, we mostly omit the main thread (it is irrelevant for our purposes) and directly start each stack to contain a single symbol (interpreted as the start frame of the passed procedure). The stack will then normally not be empty until the program terminates, whereas our model is more general and allows stacks to become intermittently empty. Finally, our rules for push and pop actions allow the shared state to change simultaneously with the stack update.

Reachability in PDS. Let $\to^*$ denote the reflexive transitive closure of $\to$. State c of a PDS is *reachable* if $c^I \to^* c$, for the initial state $c^I = \langle q^I \,|\, \varepsilon\rangle$ of the PDS. The reachability problem for a PDS is decidable [20, 48]; the idea is as follows. Given a PDS $\mathcal{P} = (Q, \Sigma, \Delta, q^I)$, the set of reachable

$$
\begin{aligned}
Q = \quad & \{q_0, q_1, q_2\} \\
\Sigma = \quad & \{0, 1, 2\} \\
\Delta = \quad & \{\langle q_0 \,|\, 0\rangle \to \langle q_1 \,|\, 10\rangle, \\
& \ \langle q_1 \,|\, 1\rangle \to \langle q_2 \,|\, 20\rangle, \\
& \ \langle q_2 \,|\, 2\rangle \to \langle q_0 \,|\, 1\rangle, \\
& \ \langle q_0 \,|\, 1\rangle \to \langle q_0 \,|\, \varepsilon\rangle \ \}
\end{aligned}
$$

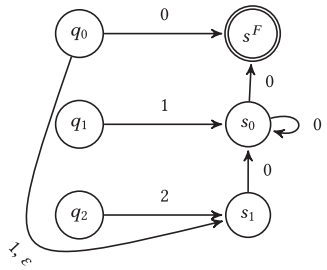

Fig. 3. A PDS $\mathcal{P}$ (left) with initial state $c^I = \langle q_0 \,|\, 0\rangle$ and the PSA $\mathcal{A}$ that captures the reachable states of $\mathcal{P}$ (right), with $S = \{q_0, q_1, q_2, s_0, s_1, s^F\}$, $I = Q$, $F = \{s^F\}$. States s^F, s_0, and s_1 of $\mathcal{A}$ are introduced in Schwoon's construction [48].

states of $\mathcal{P}$ can be represented as a standard finite-state automaton (FSA) $\mathcal{A}$, called *pushdown store automaton* [45, 48], defined as $\mathcal{A} = (S, \Sigma, \delta, I, F)$, where

- S is the finite set of states (it satisfies $Q \subseteq S$),
- Σ is the input alphabet (same as $\mathcal{P}$'s stack alphabet),
- $\delta \subseteq S \times \Sigma^{+\varepsilon} \times S$ is the transition relation,
- $I \subseteq S$ is the set of initial states (it satisfies $I \subseteq Q$), and
- $F \subseteq S$ is the set of accepting states (it satisfies $F \cap Q = \emptyset$).

Instead of accepting words, the PSA *accepts PDS state* $\langle q \,|\, w\rangle$ if, starting from PSA state q (note: $Q \subseteq S$), reading word w from left to right leads to a PSA state in F. We write $L(\mathcal{A})$ for the set of accepted PDS states. Schwoon [48] presents a polynomial-time construction that yields, for a given PDS $\mathcal{P}$, a PSA $\mathcal{A}$ with the following property.

Theorem 1 ([48]). *A PDS state is reachable in $\mathcal{P}$ exactly if it is accepted by $\mathcal{A}$.*

An example of a PDS and its corresponding store automaton is given in Figure 3.

2.2.2 Concurrent Pushdown Systems. A *concurrent pushdown system* is a fixed-thread, interleaving combination of sequential PDSs. Formally, given $n \in \mathbb{N}$, a CPDS $\mathcal{P}^n$ is a collection of n PDSs $\mathcal{P}_i = (Q, \Sigma_i, \Delta_i, q^I)$, $1 \le i \le n$. The set Q of shared states is the same for all $\mathcal{P}_i$, as is the initial shared state q^I. The PDS $\mathcal{P}_i$ have individual stack alphabets and pushdown programs. A *state* of a CPDS is an element of $Q \times \Sigma_1^* \times \cdots \times \Sigma_n^*$, written in angle brackets form $\langle q \,|\, w_1, \ldots, w_n\rangle$; this indicates that q is the shared state, and thread $i \in \{1, \ldots, n\}$ has stack contents $w_i \in \Sigma_i^*$. State $\langle q^I \,|\, \varepsilon, \ldots, \varepsilon\rangle$ is *initial*. An example of a CPDS is shown in Figure 2.

Given a state $s = \langle q \,|\, w_1, \ldots, w_n\rangle$, we refer to $\langle q \,|\, w_i\rangle$ as thread i's *thread state* and to w_i as its *local state*. In addition, of interest is frequently the *thread-visible state*, which comprises the part of s that is visible to a thread. For example, the executability of a thread's actions depends only on its visible state. Formally, let $\mathcal{T} : \Sigma^* \to \Sigma \cup \{\varepsilon\}$ be the function

$$
\mathcal{T}(w) = \begin{cases} \sigma_1 & \text{if } w = \sigma_1..\sigma_z \\ \varepsilon & \text{otherwise } (w = \varepsilon) \end{cases}, \tag{1}
$$

which extracts the top symbol, if any, from a stack. We extend $\mathcal{T}$ to act on a thread state via $\mathcal{T}\langle q \,|\, w\rangle = \langle q \,|\, \mathcal{T}(w)\rangle$, on state s via $\mathcal{T}(s) = \langle q \,|\, \mathcal{T}(w_1), \ldots, \mathcal{T}(w_n)\rangle$, and on a set S of states in the standard way via $\mathcal{T}(S) = \{\mathcal{T}(s) : s \in S\}$. We refer to $\mathcal{T}(s)$ as s's *visible state (projection)*. Note that for any set S, $\mathcal{T}(S)$ is finite.

The visible-states notion is rich enough to permit the expression of many reachability properties. In particular, it applies to all **assert** statements in programs (as described in Figure 1), which are

formulated only over program variables. Assertions are widely used in programs written in C/C++ or Java to formalize safety properties. In contrast, "heavy" properties that require the inspection of the stack history cannot be expressed over visible states, such as "the size of the stack is more than 100" or "the number of interrupt handler calls in the stack is less than 5" [2]. In this work, we consider properties expressible over visible states.

A *step* of the CPDS $\mathcal{P}^n$ is an action performed by one of the $\mathcal{P}_i$, which changes $\mathcal{P}^n$'s current state $s = \langle q \,|\, w_1, \ldots, w_n \rangle$, as follows. First, a number $i \in \{1, \ldots, n\}$ and an enabled action from Δ_i are non-deterministically chosen (if no such action exists, for any i, the CPDS cannot make a step: its current state has no successor). Second, the action is applied to state $\langle q \,|\, w_i \rangle$ of PDS $\mathcal{P}_i$. This results in a new state $\langle q' \,|\, w_i' \rangle$ of $\mathcal{P}_i$. Third, the new CPDS state s' is obtained from s by replacing q by q' and w_i by w_i'. We say the CPDS makes a step from s to s' *triggered* by thread i.

The preceding step semantics describes a binary relation $\rightarrow$ on the set of states. Let again $\rightarrow^*$ be the reflexive transitive closure. State s of CPDS $\mathcal{P}^n$ is *reachable* if $\langle q^I \,|\, \varepsilon, \ldots, \varepsilon \rangle \rightarrow^* s$. We denote by R the (often infinite) set of states reachable in $\mathcal{P}^n$. The reachability problem for CPDSs is undecidable [47].

2.2.3 From Concurrent Boolean Programs to CPDS. A concurrent Boolean program written according to the grammar in Figure 1 can be encoded as a CPDS [35, 45, 48, 49]; we briefly sketch the translation BP $\rightarrow$ PDS in this section. Because a CBP is a combination of sequential BPs, the translation CBP $\rightarrow$ CPDS can be derived naturally. A complete example is given in Figure 2.

Let sequential Boolean program $\mathbb{P}$ be defined over sets of global variables V_s and local variables V_l, respectively. Let V_{pc} be the set of program-counter values. We translate $\mathbb{P}$ into a PDS $\mathcal{P} = (Q, \Sigma, \Delta, q^I)$ as follows: $Q = \{0,1\}^{|V_s|}$ and $\Sigma = V_{pc} \times \{0,1\}^{|V_l|}$. Individual statements are translated into actions Δ. In particular, labeled statements, except procedure calls and **return**s, translate into overwrite actions Δ_{over}. Procedure calls translate into push actions Δ_{push}; **return**s translate into pop actions Δ_{pop}. Modeling the processing of procedure calls via stacks will be introduced in Section 5.1. The initial thread state c^I of the PDS is defined by the variable initializations according to Figure 1; in particular, c^I satisfies $pc = 0$—that is, execution starts in the first program location.

The properties of interest, written as **assert** statements, are encoded as a set of visible (thread) states. Each assertion is typically translated into a set of states.

Example 2. Consider a Boolean program $\mathbb{P}$ with a shared variable s, a local variable l and $|V_{pc}| = 9$, and a statement **assert** $(l = 0)$ labeled by 7. We convert the assertion-checking problem into a reachability problem by translating the negated assertion into a set of thread-visible states. To attain this goal, the values of global variable s are encoded as state $q \in \{0,1\}$. Similarly, the values of local-variable pair $(pc, l) = (7, 0)$ are encoded as the single stack-alphabet symbol $\sigma = 7$. Therefore, this assertion can be encoded as visible thread states $\{\langle 0 \,|\, 7 \rangle, \langle 1 \,|\, 7 \rangle\}$. Reaching either one of them implies a property violation.

We remark that the translation from a BP to a PDS is lossless, and therefore so is the translation from a CBP to a CPDS. The verification results of the properties obtained for a CBP and its translated CPDS are thus equivalent. In practice, the preceding translation can cause a blow-up, which can be curbed using on-the-fly techniques (this issue is orthogonal to the technique presented here).

2.3 Context-(Un)Bounded Reachability

Consider a *path* generated by a CPDS—for instance, a sequence of states pairwise related by $\rightarrow$. Each step along the path is triggered by exactly one thread. A *context* is a sequence of consecutive steps along a path that are all triggered by the same thread. For $k \in \mathbb{N}$, we say state s is

reachable with context bound k (or *within k contexts*, for short) if there exists a path from $\langle q^I \mid \varepsilon, \ldots, \varepsilon\rangle$ to s that consists of at most k contexts.[1] Context-bounded reachability analysis—the problem of determining, given a CPDS $\mathcal{P}^n$, a state s, and a number k, whether s is reachable in $\mathcal{P}^n$ within k contexts—is practically interesting in part because it is decidable [45]; the decision procedure uses a PSA to symbolically represent the (potentially) infinite number of reachable states.

In contrast, in this work we are concerned with the undecidable Context-UnBounded (reachability) Analysis problem, CUBA, for CPDSs. In the sequel, resource bound k will therefore always play the role of the number of contexts. We denote by R_k the set of states reachable within k contexts. Despite the bound, set R_k can still be infinite: k does not restrict the number of steps a single thread can trigger within a context.

3 ANALYZING PROGRAMS USING OSS

In this section, we introduce the key technical concepts behind the simple and elegant paradigm of OSs. With the intent to use this paradigm to tackle context-unbounded reachability in mind, we accompany this introduction with examples relevant to the CUBA problem and discuss specific circumstances of the paradigm's application to CUBA at the end.

The paradigm considers programs $\mathcal{P}$ that are parameterized by the amount k available of some *resource*. Resource bounds may be physical or logical, and they may be imposed only for analysis purposes (such as in this article). We have successfully used the paradigm with resources such as the capacity of some data structure, such as a FIFO queue bound [39], and the delay bound of a deterministic scheduler [19]: the latter bound specifies how often a scheduler may skip the process to be executed next to generate other execution schedulers without resorting to full non-determinism.

Let C be a property of interest, which we wish to establish in $\mathcal{P}$ for all values of k. The generic plan is to analyze $\mathcal{P}$ for increasing values $k = 0, 1, \ldots$ and to compute, for each k, an *observation* O_k about $\mathcal{P}$, such as what states are reachable under bound k.

Definition 3. An observation sequence is a sequence $(O_k)_{k=0}^{\infty}$ with the following properties:

- Monotonicity: For all k, $O_k \subseteq O_{k+1}$.
- Computability: For all k, O_k is computable and finitely representable.
- Expressibility: Property C is expressible over $(O_k)_{k=0}^{\infty}$, which means that for all k, O_k contains enough information to decide whether C holds for $\mathcal{P}$ under bound k, and this question is algorithmically decidable.

Sequence $(O_k)_{k=0}^{\infty}$ has *domain* D if $\cup_{k=0}^{\infty} O_k \subseteq D$.

Regarding the computability requirement, note that bounding the resource does not guarantee finiteness of the resulting state space—there can be other sources of infinity, such as unbounded recursion in the case of the CUBA problem. It is therefore not clear that the sets O_k can even be finitely represented, let alone computed. If they can, symbolic techniques (e.g., based on automata) may be required, but they can make certain reasoning steps involved in the paradigm more expensive. For the CUBA case, we discuss this problem in detail in Section 6, where we give a criterion that guarantees that *for every k*, the observation sets O_k (which are sets of reachable states in this case) are finite. This guarantees computability even without symbolic data structures.

Example 4. Consider the two-thread CPDS in Figure 4 (left). If k denotes the maximum number of thread contexts allowed per execution path, then let R_k be the set of global states reachable up

[1] An alternative definition uses the notion of a context *switch* bound [45]. We chose the equivalent but simpler formulation via context bounds. The two are related in the obvious way.

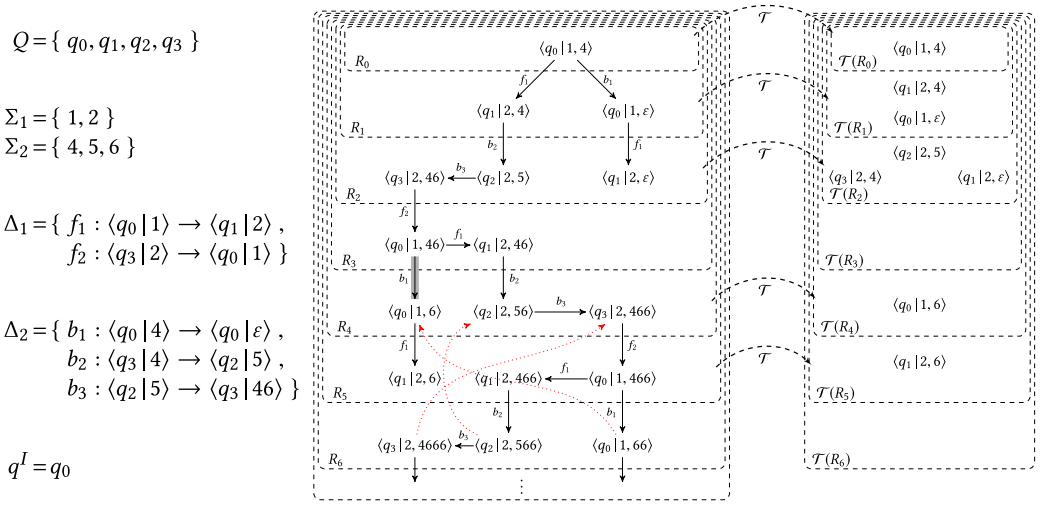

Fig. 4. A two-thread CPDS $\mathcal{P}^2$ (left) and its *reachability table* (right). We have $\mathcal{P}^2 = \{\mathcal{P}_1, \mathcal{P}_2\}$ with $\mathcal{P}_i = (Q, \Sigma_i, \Delta_i, q^I)$ for $i = 1, 2$; the initial state is $\langle q_0 \,|\, 1, 4 \rangle$. For $k = 1, \ldots, 6$, each entry labeled R_k on the right actually shows the set $R_k \setminus R_{k-1}$ of reachable states that are *new* for bound k; similarly for $\mathcal{T}(R_k)$. As illustrated by the red dotted arrows (which are not steps of the CPDS), the set $R_6 \setminus R_5$ equals $R_4 \setminus R_3$ except that in all states of the former set, the stacks of $\mathcal{P}_2$ contain an additional symbol 6 at the bottom. As is easy to see, this pattern repeats every two rounds. As a result, set R_k strictly grows with increasing values of k (i.e., sequence $(R_k)_{k=0}^{\infty}$ does not converge).

Table 1. Basic Terminology Concerning an OS $(O_k)_{k=0}^{\infty}$

Terminology: $(O_k)_{k=0}^{\infty}$ ...	**Definition**	**Explanation**
... *plateaus at* k_0	$O_{k_0} = O_{k_0+1}$	Pauses *or* stops growing
... *stutters at* k_0	$O_{k_0} = O_{k_0+1} \wedge \exists_{k>k_0}\, O_k \subsetneq O_{k+1}$	Pauses but does not stop growing
... *collapses at* k_0	$\forall_{k \geq k_0}\, O_{k_0} = O_k$	Stops growing
... *converges*	$\exists_{k_0}\, (O_k)_{k=0}^{\infty}$ collapses at k_0	Collapses at some k_0
... *is stutter-free*	$\forall_{k_0}\, \neg((O_k)_{k=0}^{\infty}$ stutters at $k_0)$	Does not stutter at any k_0

to bound k, and let $\mathcal{T}(R_k)$ be the set of visible states (see Section 2.2.2) reachable up to bound k. Both $(R_k)_{k=0}^{\infty}$ and $(\mathcal{T}(R_k))_{k=0}^{\infty}$ are OSs: they are monotone by construction and computable by Qadeer and Rehof [45]. Any reachability property is expressible over $(R_k)_{k=0}^{\infty}$, whereas $(\mathcal{T}(R_k))_{k=0}^{\infty}$ is more restricted but still permits context-bounded properties stating shared-state reachability or mutually exclusive local-state reachability.

Table 1 defines the key concepts that are frequently used in this article. In addition, an OS *diverges* if it does not converge. This and all terms defined in Table 1 are with respect to an input program over which the OS is defined.

Observation sequences: basic properties. The phenomena of plateaus, stuttering, collapse, and convergence of OSs interact in several interesting ways, summarized in this section.

Property 5. *An OS over a finite domain converges.*

Proof. Let $K = \{k \in \mathbb{N} : O_k \subsetneq O_{k+1}\}$ be the set of indices where the sequence increases. As the domain of $(O_k)_{k=0}^{\infty}$ is finite, set K is finite. Therefore, let $k_0 := \max K + 1$. For every $k \geq k_0$, we have $k \notin K$ and hence, by monotonicity, $O_k = O_{k+1}$. The sequence thus collapses at k_0 and converges. □

Convergence alone does not imply that the limit of sequence $(O_k)_{k=0}^{\infty}$ can be computed, due to stuttering. In the absence of stuttering, however, reaching a plateau is tantamount to convergence.

Property 6. *If OS $(O_k)_{k=0}^{\infty}$ does not stutter at k_0 and plateaus at k_0, then it collapses at k_0.*

Proof. Recall that "plateaus at k_0" means $O_{k_0} = O_{k_0+1}$, and "$(O_k)_{k=0}^{\infty}$ does not stutter at k_0" means $O_{k_0} = O_{k_0+1} \Rightarrow \forall k \geq k_0 \; O_k = O_{k+1}$ is valid. Together, we have $\forall k \geq k_0 \; O_k = O_{k+1}$, from which the collapse at k_0 immediately follows. □

Example 7. We revisit the example in Figure 4. Sequence $(\mathcal{T}(R_k))_{k=0}^{\infty}$ has a finite domain and thus converges. As the right-hand side of the figure shows, it stutters at $k = 2$: $\mathcal{T}(R_2) = \mathcal{T}(R_3) \neq \mathcal{T}(R_4)$; then it plateaus again at $k = 5$. The goal in this work is to show that it in fact collapses at $k = 5$. In contrast, sequence $(R_k)_{k=0}^{\infty}$ is stutter-free (shown in Section 4.1) but diverges (suggested by Figure 4; see also Example 27).

Consider now the tentative Scheme 1 for verifying a property C using an OS $(O_k)_{k=0}^{\infty}$. The scheme attempts the obvious: it increases k (with O_0 suitably initialized) until the sequence *seems to* have converged, checking for errors on the way. We often refer to the iterations of the main loop in Line 1 as *rounds* of Scheme 1.

SCHEME 1: Verifying C for unbounded k using some OS $(O_k)_{k=0}^{\infty}$

Input: A program $\mathcal{P}$, property C
Output: Whether C holds for all resource bounds k; if not, a value of k for which it is violated

```
1: for k := 1 to ∞ do
2:     compute the set O_k of observations under resource bound k
3:     if O_k violates C then
4:         return "error reachable with resource bound k"
5:     if O_{k−1} = O_k then
6:         return "C holds for any resource bound"
```

Whether this scheme is implementable, and what it is good for, depends on features of the OS:

(a) Since, for all k, O_k is computable and finitely representable (Definition 3), Line 2 is implementable.
(b) Since C is expressible over $(O_k)_{k=0}^{\infty}$ (Definition 3), Line 3 is implementable.
(c) If equality in the sequence domain is decidable, then the test in Line 5 is implementable. (This condition is non-trivial if the domain of $(O_k)_{k=0}^{\infty}$ is infinite.)
(d) If C is violated (i.e., for some k), then Scheme 1 terminates (in Line 4).
(e) If $(O_k)_{k=0}^{\infty}$ converges, then Scheme 1 terminates as well, since then eventually $O_{k-1} = O_k$.
(f) If $(O_k)_{k=0}^{\infty}$ is stutter-free, then the output in Line 6 is correct.

We summarize the special case that preconditions (c), (e), and (f) are met as follows.

Property 8. *If OS $(O_k)_{k=0}^{\infty}$ converges and is stutter-free, and equality in the domain of $(O_k)_{k=0}^{\infty}$ is decidable, then any property expressible over $(O_k)_{k=0}^{\infty}$ is decidable.*

OSs for CUBA. Consider now sequences over parameter k denoting a context bound for concurrent threads running recursive procedures over which the CUBA problem is expressible. We take a systematic look at the interaction between stuttering and convergence for such an OS.

Stutter-free:	Always converges: Yes	Always converges: No
Yes	×	$(R_k)_{k=0}^{\infty}$
No	$(\mathcal{T}(R_k))_{k=0}^{\infty}$	–

The table shows the four combinations possible between these two features. The apparently most powerful OSs do not stutter and are guaranteed to converge. By Property 8 and the undecidability of the CUBA problem, such an OS *does not exist* ("×" in the table). A stutter-free OS that is not guaranteed to converge, such as the sequence $(R_k)_{k=0}^{\infty}$ in Example 4, gives rise to a possibly non-terminating but partially correct algorithm derived from Scheme 1; this case is discussed in Section 4.1. A stuttering OS that always converges, such as the sequence $(\mathcal{T}(R_k))_{k=0}^{\infty}$ also used in Example 4, is algorithmically more difficult to use; this case is discussed in detail in Section 4.2. Finally, OSs that may not converge *and* suffer from stuttering are least useful and not considered in this article.

4 CUBA using OSs

This section presents two instances of applying the OS paradigm to the CUBA problem, namely using sequences $(R_k)_{k=0}^{\infty}$ and $(\mathcal{T}(R_k))_{k=0}^{\infty}$ mentioned at the end of Section 3. As we will see, sequence $(R_k)_{k=0}^{\infty}$ can solve the CUBA problem only in a limited number of cases, but it serves as a good instance to illustrate the technique, and potential challenges, which will then be tackled by using $(\mathcal{T}(R_k))_{k=0}^{\infty}$ instead.

4.1 CUBA Using Global-State Reachability

To illustrate the use of OSs for context-unbounded analysis, we begin with the simple instance of sequence $(R_k)_{k=0}^{\infty}$, which maps the context parameter k to the full set R_k of states reachable under that context bound. $(R_k)_{k=0}^{\infty}$ does not suffer from plateaus that "fake convergence," as is straightforward to show next.

Lemma 9. *$(R_k)_{k=0}^{\infty}$ is stutter-free: for all k_0, if $R_{k_0} = R_{k_0+1}$, then for all $k \geq k_0$, $R_k = R_{k+1}$.*

Proof. Given $R_{k_0} = R_{k_0+1}$, we show the claim about k by induction. It holds for $k = k_0$. We assume $R_k = R_{k+1}$ and show $R_{k+1} = R_{k+2}$. Appealing to monotonicity, we only show that $R_{k+2} \subseteq R_{k+1}$.

To this end, let $t \in R_{k+2}$, so there exists a path p to t that uses at most $k+2$ contexts. Let s be the state along this path *right before the final context switch*. State s splits p into two sub-paths: $p = p_1 \circ p_2$.

By construction, $s \in R_{k+1}$. By the induction hypothesis ($R_k = R_{k+1}$), there exists a path p' to s that uses at most k contexts. Because p' ends in s and p_2 starts in s, sequence $p'' := p' \circ p_2$ is a well-formed path. Moreover, p'' ends in t because p_2 does. Finally, p'' uses only $k+1$ contexts: k contexts along p', plus one more along p_2. As a result, $t \in R_{k+1}$. □

By the discussion from Section 3, Scheme 1 instantiated with the stutter-free OS $(R_k)_{k=0}^{\infty}$, denoted by *Scheme 1* (R_k), is a partially correct algorithm to decide reachability in CPDSs for unbounded numbers of thread contexts.

Example 10. We revisit the two-thread program shown in Figure 2. It is taken, with slight adaptations, from Prabhu et al. [43]; the approach presented there fails to terminate on that example. The example is small but non-trivial in that both threads have stacks, and each can grow without

bound, even without any context switch. *Scheme 1* (R_k) does not terminate at $k = 2$: $R_1 \subsetneq R_2$. Indeed, consider $c = \langle q_0 \,|\, 4, \varepsilon\rangle$: thread foo is spinning in the **while** loop, and bar has reached the end of its program. This state is reachable with two contexts (belongs to R_2), as witnessed by the path

$$\langle q_1 \,|\, 2, 6\rangle \xrightarrow{f_{2b}} \langle q_1 \,|\, 4, 6\rangle \xrightarrow{b_{6b}} \langle q_1 \,|\, 4, 8\rangle \xrightarrow{b_{8b}} \langle q_1 \,|\, 4, 9\rangle \xrightarrow{b_9} \langle q_0 \,|\, 4, \varepsilon\rangle \,.$$

State c is obviously not reachable with one context (does not belong to R_1), because both threads must leave their initial state to get to c. However, *Scheme 1* (R_k) does terminate for $k = 3$: $R_2 = R_3$.

In practice, the utility of *Scheme 1* (R_k) is limited by two factors. The first is that the efficiency of implementing it suffers from the complexity of computing and representing the sets R_k: once computed, they are used in Line 5 in a comparison. If we use PSAs A_k to represent the R_k, the comparison requires an automaton-equivalence check: $R_{k-1} = R_k$ amounts to checking $R_k \subseteq R_{k-1}$, which can be done via language containment: $L(A_k) \cap \overline{L(A_{k-1})} = \emptyset$, where $L(\cdot)$ denotes an automaton's language, and $\overline{\cdot}$ denotes language complement. Efficient complementation requires deterministic automata. Because the NFAs A_k already have worst-case size exponential in k, the determinization can potentially yield a doubly exponential running time. These estimates suggest that, using PSAs, *Scheme 1* (R_k) cannot be expected to scale to interesting PDSs. We address this problem in Section 6.

The second, and more fundamental, limitation is that $(R_k)_{k=0}^{\infty}$ cannot eventually hit a plateau for every input program: the CUBA problem's undecidability implies that there must be (many) instances of divergence. We address this problem in the rest of this section.

4.2 CUBA Using Visible-State Reachability

Because we cannot have both stutter-freeness and a convergence guarantee for an OS that solves the full CUBA problem, we investigate in this section a sequence with features diagonally opposite to those of $(R_k)_{k=0}^{\infty}$: guaranteed convergence but potential stuttering. By Property 5, convergence is guaranteed by a finite domain. To this end, let $(\mathcal{T}(R_k))_{k=0}^{\infty}$ be the sequence that maps the context parameter k to the *finite* set $\mathcal{T}(R_k)$ of *visible* states reachable under that context bound (see Section 2.2.2 and Example 4). Guaranteeing convergence, this sequence must exhibit stuttering behavior for some programs. We give an example of this phenomenon.

Example 11. We revisit the example in Figure 4. The sequence $(\mathcal{T}(R_k))_{k=0}^{\infty}$ plateaus at $k = 2$: $\mathcal{T}(R_2) = \mathcal{T}(R_3)$. In the next step, we find out that this plateau is merely due to stuttering: $\mathcal{T}(R_3) \subsetneq \mathcal{T}(R_4)$. Another plateau emerges at $k = 5$. The table shows no reachable visible states beyond $\mathcal{T}(R_5)$.

In this section, we design a technique that easily proves that $(\mathcal{T}(R_k))_{k=0}^{\infty}$ collapses at $k = 5$ for the program in Figure 4. The problem we have to solve is to distinguish plateaus that merely signal stuttering from those that signal sequence convergence. We will accomplish this by replacing the test $\mathcal{T}(R_{k-1}) = \mathcal{T}(R_k)$ in Line 5 of Scheme 1 (which would lead to incorrect answers) by a stronger one that rules out stuttering.

4.2.1 Convergence Detection Using Generators. The idea for designing a sufficient condition for the absence of stuttering at k is as follows. Assume, by contraposition, that $\mathcal{T}(R_{k-1}) = \mathcal{T}(R_k) \subsetneq \mathcal{T}(R)$. It is clear that once any $g \in \mathcal{T}(R) \setminus \mathcal{T}(R_k)$ has been encountered (for some larger k), its discovery may generate many more heretofore unseen states that follow in the wake of g. But what about the *first* time a new state in $\mathcal{T}(R) \setminus \mathcal{T}(R_k)$ is encountered after a plateau? We argue that such a state is of a special form. If we can rule out the reachability of any new states of this form, for any larger k, then the same holds for all of those *first* new encountered states and hence for *any* new states: the sequence has converged.

Before we put this intuition to the test, we formalize the idea. We slightly generalize the concept of generators to any set $\mathcal{G}$ of visible states with the following "pragmatic" property: if, upon encountering a plateau, all reachable visible states in $\mathcal{G}$ have been reached, the sequence converges.

Definition 12. A set $\mathcal{G}$ of visible states is a generator set if the following condition holds for every k: if $\mathcal{T}(R_{k-1}) = \mathcal{T}(R_k)$ and $\mathcal{G} \cap \mathcal{T}(R) \subseteq \mathcal{T}(R_k)$, then $\mathcal{T}(R_k) = \mathcal{T}(R)$.

It is immediate from the definition that the notion of being a generator set is *upward closed*: any superset of a generator set is also a generator set. Because our goal is to prove $\mathcal{T}(R_k) = \mathcal{T}(R)$, the right-hand side of the implication in Definition 12, there is a natural desire to make the left-hand side "weak," hence to keep generator sets small.

Definition 12 suggests the following strategy for convergence detection using generators:

(a) Statically define a set $\mathcal{G}$ of visible states, and prove that it is a generator set.
(b) During the execution of Scheme 1, if the sequence plateaus at $k-1$, check whether all reachable generators (the elements of $\mathcal{G} \cap \mathcal{T}(R)$) have been reached. If so, terminate; $\mathcal{T}(R_k)$ is the set of reachable visible states.

So far, the discussion in Section 4.2 has been very generic. The preceding step (a) of course depends on the application and is the topic of Section 4.2.2. Step (b) seems preposterous: it involves the set $\mathcal{T}(R)$ of reachable visible states whose determination is the very goal of this section—are we caught in a cyclic argument? We address this step in Section 4.2.3.

4.2.2 A Generator Set for CUBA. Given an application domain for the OS paradigm, finding a set $\mathcal{G}$ with the strong guarantees suggested by Definition 12 requires good intuition about that domain. For the case of CPDSs, we define $\mathcal{G}$ to be the visible states $\langle q \,|\, \sigma_1, \ldots, \sigma_n \rangle$ in which at least one thread-visible state $\langle q \,|\, \sigma_i \rangle$ may have emerged as the result of a **pop** action, in the following sense:

$$\begin{aligned} \mathcal{G} = \{\ \langle q \,|\, \sigma_1, \ldots, \sigma_n \rangle :\ & \text{there exists } i \text{ s.t.} \\ & - \ \langle q \,|\, \varepsilon \rangle \text{ is the target of a } \textit{pop} \text{ edge in } \Delta_i \textbf{ and} \\ & - \ (\sigma_i = \varepsilon \ \textbf{ or } \ \langle ? \,|\, ? \sigma_i \rangle \text{ is the target of a } \textit{push} \text{ edge in } \Delta_i\ )\}, \end{aligned} \tag{2}$$

where ? stands for arbitrary shared states or stack symbols, respectively. Given i as in Equation (2), visible thread states $\langle q \,|\, \sigma_i \rangle$ are those that can emerge from a pop: the shared state is the target of a pop, **and** the resulting stack is either empty **or** has a top symbol that was overwritten as part of an earlier push. These are the symbols ρ_1 in a push action $(q, \sigma) \to (q', \rho_0 \rho_1)$.

Example 13. We revisit the example in Figure 4. State $\langle q_0 \,|\, 1, 6 \rangle$ is a generator: $\langle q_0 \,|\, \varepsilon \rangle$ is the target of pop edge $\langle q_0 \,|\, 4 \rangle \to \langle q_0 \,|\, \varepsilon \rangle$, and $\langle ? \,|\, ? 6 \rangle$ is the target of push edge $\langle q_2 \,|\, 5 \rangle \to \langle q_3 \,|\, 46 \rangle$, both in Δ_2.

Note that $\mathcal{G}$ is defined purely syntactically via the pushdown programs. As a consequence, stack symbols σ_j for threads $j \neq i$ can be arbitrary and in particular unreachable—Equation (2) does not restrict them in any way. This problem is solved as part of the preceding step (b), where we aim to project set $\mathcal{G}$ to its reachable fragment; see Section 4.2.3.

Theorem 14. *Set $\mathcal{G}$ defined in Equation (2) is a generator set.*

Proof. The intuition is that for a pop action, the visible state after the action depends on more than the current visible state—namely, it also depends on the stack element right below the top. In contrast, for pushes and overwrites, the next visible state can be determined solely from the current visible state and the action, independently of the stack history. Intuitively, these properties mean the following: if all visible states reachable after a pop have been reached, the uncertainty inherent in pop actions goes away; consequently, reaching a plateau implies convergence in this case.

Now we have the formal argument. The condition in Definition 12 has the propositional form $X \wedge Y \Rightarrow Z$, which is equivalent to $X \wedge \neg Z \Rightarrow \neg Y$; we prove the latter form. Therefore, let k be such that $\mathcal{T}(R_{k-1}) = \mathcal{T}(R_k)$; furthermore, assume that $\mathcal{T}(R_k) \neq \mathcal{T}(R)$ (i.e., $\mathcal{T}(R) \setminus \mathcal{T}(R_k) \neq \emptyset$); we prove $\mathcal{G} \cap \mathcal{T}(R) \not\subseteq \mathcal{T}(R_k)$ (i.e., $\mathcal{G} \cap (\mathcal{T}(R) \setminus \mathcal{T}(R_k)) \neq \emptyset$).

Sequence $(\mathcal{T}(R_k))_{k=0}^{\infty}$ stutters at $k-1$; due to guaranteed convergence, the plateau eventually comes to an end (but could be long). Let K mark the "last index of equality" during the stuttering:

$$K = \max\{k' \in \mathbb{N} : k' \geq k \wedge \mathcal{T}(R_{k'-1}) = \mathcal{T}(R_{k'})\}.$$

So we have $\mathcal{T}(R_{k-1}) = \mathcal{T}(R_k) = \ldots = \mathcal{T}(R_K) \subsetneq \mathcal{T}(R_{K+1})$. Consider a path p to some state in R_{K+1} whose visible-state projection $\mathcal{T}$ belongs to $\mathcal{T}(R_{K+1}) \setminus \mathcal{T}(R_K)$. Along p, let now t be the *first* state such that $\mathcal{T}(t) \notin \mathcal{T}(R_K)$; state t must necessarily be reached inside context $K+1$. The choice of t means that for the state s preceding t along p, $\mathcal{T}(s) \in \mathcal{T}(R_K) = \mathcal{T}(R_{K-1})$. Hence, there exists a state $s' \in R_{K-1}$ such that $\mathcal{T}(s') = \mathcal{T}(s)$. We visualize this situation as follows:

$$p: \quad \cdots \circ \underbrace{\rightarrow \cdots \rightarrow}_{K-1} \circ \underbrace{\rightarrow \cdots \rightarrow}_{K} \circ \underbrace{\overset{i}{\rightarrow} \cdots \overset{i}{\rightarrow} \overset{s}{\cdot} \overset{i:\,a}{\rightarrow} \overset{t}{\cdot} \rightarrow \cdots}_{K+1}$$

In this representation, $\circ$ denotes context switches, and $\rightarrow$ denotes steps (global transitions). Index i is the identity of the thread executing in context $K+1$, and a is the final action executed by i as it reaches t.

We now show that $\mathcal{T}(t) \in \mathcal{G}$, which proves $\mathcal{G} \cap (\mathcal{T}(R) \setminus \mathcal{T}(R_k)) \neq \emptyset$. To this end, we distinguish action types for a: if a is a *push* or an *overwrite*, then firing a from $\mathcal{T}(s')$ instead of $\mathcal{T}(s)$ yields the same successor visible state, namely $\mathcal{T}(t)$ (the successor does not depend on the stack history in these cases). This conclusion contradicts $\mathcal{T}(t) \notin \mathcal{T}(R_K)$: we have $\mathcal{T}(s') \in \mathcal{T}(R_{K-1})$, and we need at most one context switch to fire a from $\mathcal{T}(s')$.

Thus, a must be a *pop* action. Consequently, for the thread-visible state $\langle q \,|\, \sigma_i \rangle$ of thread i in state t, we have $a = \cdots \rightarrow (q, \varepsilon) \in \Delta_i$, for thread i's pushdown program, and after action a the stack of i is either empty ($\sigma_i = \varepsilon$), or symbol σ_i was overwritten as part of the action that, some time ago, pushed the symbol just popped, so there is a push edge of the form $\cdots \rightarrow (?, ?\sigma_i)$, as required by Equation (2). As a result, $\mathcal{T}(t) \in \mathcal{G}$. □

4.2.3 Overapproximating the Reachable Generators. Looking at task (b) following Definition 12, how do we prove that all reachable generators—the elements of $\mathcal{G} \cap \mathcal{T}(R)$—have been reached for *any* future round? And is not the computation of $\mathcal{T}(R)$ exactly the problem we set out to solve to begin with, namely the reachability of visible states for arbitrary context bounds? The resolution of the paradox is an interpolation argument: we do not need to compute $\mathcal{G} \cap \mathcal{T}(R)$ precisely—any overapproximation of it that is contained in $\mathcal{T}(R_k)$ is sufficient to prove $\mathcal{G} \cap \mathcal{T}(R) \subseteq \mathcal{T}(R_k)$, the crucial condition of Definition 12. An overapproximation of $\mathcal{G} \cap \mathcal{T}(R)$ can in turn be obtained from an overapproximation Z of $\mathcal{T}(R)$; we then have $\mathcal{G} \cap Z \supseteq \mathcal{G} \cap \mathcal{T}(R)$.

How do we efficiently compute a tight estimate $Z \supseteq \mathcal{T}(R)$? We are overapproximating a set whose exact computation requires a context- and synchronization-sensitive analysis, which is undecidable. However, if we drop *either one* of these sensitivities, the problem becomes decidable and easier in practice. We choose here a *context-insensitive* overapproximation, as follows.

Our rough idea is that we cut off the stack at size 1. For each push action, we ignore the stack contents underneath the newly pushed element. For each pop action, we do not know what the emerging element is, but we do know that it is either ε (the stack has become empty) or a symbol that was overwritten as part of an earlier push. To cover the latter case, we inspect all push actions in the program and collect the set E of symbols written right underneath the newly pushed symbol. These are the candidates for elements emerging after a pop. We do not care whether it is the right one—our analysis is context-insensitive.

Cutting off the stack at size 1 this way results in a finite-state system that can simply be explored exhaustively; its reachable states form the set Z. Formally, given $\mathcal{P} = (Q, \Sigma, \Delta, q^I)$, let $\Sigma^{+\varepsilon} = \Sigma \cup \{\varepsilon\}$. We construct $\mathcal{M} = (Q \times \Sigma^{+\varepsilon}, T)$ as done in Algorithm 2. Lines 2 and 3 collect the set E as defined earlier. The loop beginning in Line 5 constructs the set $T \subseteq (Q \times \Sigma^{+\varepsilon}) \times (Q \times \Sigma^{+\varepsilon})$ of transitions; we write $\langle q \,|\, \sigma\rangle \mapsto_T \langle q' \,|\, \sigma'\rangle$ for $(\langle q \,|\, \sigma\rangle, \langle q' \,|\, \sigma'\rangle) \in T$. Each action gives rise to a transition in $\mathcal{M}$ (Line 6). Pushes are implemented by dropping the symbol underneath the newly pushed symbol, which is accomplished by the $\mathcal{T}$ function. In addition, for pops we context-insensitively overapproximate the emerging symbol, which is done via candidate set E, as explained previously.

ALGORITHM 2: Construction of finite-state system $\mathcal{M}$ whose reachability set defines Z

Input: A sequential PDS $\mathcal{P} = (Q, \Sigma, \Delta, q^I)$
Output: A sequential finite-state system $\mathcal{M} = (Q \times \Sigma^{+\varepsilon}, T)$

```
1:  E = ∅                                                    ▷ set of "emerging" symbols
2:  for each push action (q, σ) → (q′, ρ0ρ1) ∈ Δ do
3:      E = E ∪ {ρ1}
4:  T = ∅
5:  for each action (q, σ) → (q′, σ′) ∈ Δ do
6:      T = T ∪ {(q, σ) ↦ (q′, 𝒯(σ′))}
7:      if σ ≠ ε ∧ σ′ = ε then                               ▷ a pop action
8:          for each ρ ∈ E do
9:              T = T ∪ {(q, σ) ↦ (q′, ρ)}
10: return (Q × Σ^{+ε}, T)
```

Given a CPDS $\mathcal{P}^n = \mathcal{P}_1 \| \ldots \| \mathcal{P}_n$, we begin by building a multi-threaded finite-state program $\mathcal{M}^n = \mathcal{M}_1 \| \ldots \| \mathcal{M}_n$ with $\mathcal{M}_i = (Q \times \Sigma_i^{+\varepsilon}, T_i)$, $1 \le i \le n$, where $\mathcal{M}_i$ is obtained via Algorithm 2 on $\mathcal{P}_i$. A *state* of $\mathcal{M}^n$ is an element of $Q \times \Sigma_1^{+\varepsilon} \times \cdots \times \Sigma_n^{+\varepsilon}$; a *transition* of $\mathcal{M}^n$, written in the form $\langle q \,|\, \sigma_1, \ldots, \sigma_n\rangle \mapsto \langle q' \,|\, \sigma_1', \ldots, \sigma_n'\rangle$, is defined exactly if there exists $i \in \{1, \ldots, n\}$ such that $\langle q \,|\, \sigma_i\rangle \mapsto_{T_i} \langle q' \,|\, \sigma_i'\rangle$ and for all $j \ne i, \sigma_j = \sigma_j'$. Like $\mathcal{P}^n$, $\mathcal{M}^n$ has an interleaving execution model: each transition affects the local state of at most one thread. With an initial state $t^I := \langle q^I \,|\, \varepsilon, \ldots, \varepsilon\rangle$ of $\mathcal{M}^n$, Z is defined as the set of reachable states of $\mathcal{M}^n$.

Lemma 15. $Z \supseteq \mathcal{T}(R)$.

Proof. Let $t \in R$, and hence $\bar{t} := \mathcal{T}(t) \in \mathcal{T}(R)$; we show $\bar{t} \in Z$. Let Π be a path in $\mathcal{P}^n$ to t, and let $\bar{\Pi}$ be the sequence of visible states obtained by applying $\mathcal{T}$ to each state along Π in sequence:

$$\begin{array}{lccccccccc}
\Pi :: & t^I = \langle q^I \,|\, \varepsilon, \ldots, \varepsilon\rangle & \xrightarrow{a_1} & \ldots & \xrightarrow{a_{l-1}} & t_{l-1} & \xrightarrow{a_l} & \ldots & \xrightarrow{a_m} & t_m = t \\
\downarrow \mathcal{T} & \downarrow \mathcal{T} & & & & \downarrow \mathcal{T} & & & & \downarrow \mathcal{T} \\
\bar{\Pi} :: & \bar{t}^I = \langle q^I \,|\, \varepsilon, \ldots, \varepsilon\rangle & \overset{e_1}{\longmapsto} & \ldots & \overset{e_{l-1}}{\longmapsto} & \bar{t}_{l-1} & \overset{e_l}{\longmapsto} & \ldots & \overset{e_m}{\longmapsto} & \bar{t}_m = \bar{t}.
\end{array}$$

We first show that $\bar{\Pi}$ is a legal path in $\mathcal{M}^n$. By the construction of T in Algorithm 2, mapping any step $t_l \to t_{l+1}$ along Π via $\mathcal{T}$ to an edge $\bar{t}_l \mapsto \bar{t}_{l+1}$ along $\bar{\Pi}$ results in a valid transition in $\mathcal{M}_i$, where i is the index of the thread triggering the step in Π: all stack symbols but the i^{th} are unchanged, because $\mathcal{P}^n$ and $\mathcal{M}^n$ share the same strictly interleaving model.

Because $\bar{\Pi}$ is a legal path in $\mathcal{M}^n$ and ends in $\bar{t}$, visible state $\bar{t}$ is in $\mathcal{M}^n$'s reachability set Z. □

Example 16. Figure 5 shows the two-thread finite-state program $\mathcal{M}^2$ and set Z for the program in Figure 4.

$$
\begin{aligned}
T_1 \;=\; \{ & \\
& e_1 : \langle q_0 \,|\, 1\rangle \;\mapsto\; \langle q_1 \,|\, 2\rangle \\
& e_2 : \langle q_3 \,|\, 2\rangle \;\mapsto\; \langle q_0 \,|\, 1\rangle \;\} \\
\\
T_2 \;=\; \{ & \\
& f_1 : \langle q_0 \,|\, 4\rangle \;\mapsto\; \langle q_0 \,|\, \varepsilon\rangle \\
& f_2 : \langle q_0 \,|\, 4\rangle \;\mapsto\; \langle q_0 \,|\, 6\rangle \\
& f_3 : \langle q_1 \,|\, 4\rangle \;\mapsto\; \langle q_2 \,|\, 5\rangle \\
& f_4 : \langle q_2 \,|\, 5\rangle \;\mapsto\; \langle q_3 \,|\, 4\rangle \;\}
\end{aligned}
$$

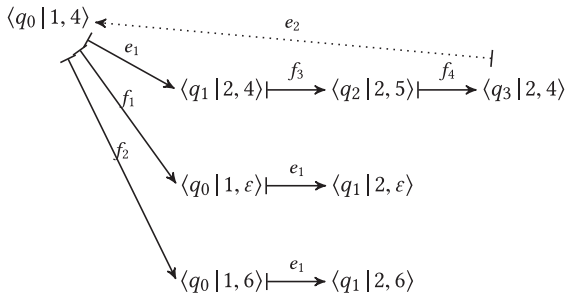

Fig. 5. An example showing the computation of Z for the CPDS in Figure 4. $\mathcal{M}^2 = (\mathcal{M}_1, \mathcal{M}_2)$. We start $\mathcal{M}^2$ in $\langle q_0 \,|\, 1, 4\rangle$. Left: The transitions in T_1 and T_2. Right: The reachability tree of $\mathcal{M}^2$ whose states form the set Z.

Increasing the precision of Z. The preceding solution to approximate the set $\mathcal{T}(R)$ from above can easily be generalized in a way that may give rise to a more precise approximation: instead of cutting off the stack at size 1, we can build a *b-bounded* PDS—a finite-state system that resembles a PDS except that the stack has a capacity limit of b. It differs from a PDS in the following modifications for push and pop actions, which are designed to enforce the overapproximation:

Push actions: First perform the action under the standard semantics. If, as a result, the stack size is $b + 1$, then drop the bottom stack symbol.

Pop actions: First perform the action under the standard semantics. If, as a result, the stack size is $b - 1$, then create $|E|$ *additional* successors by non-deterministically adding an element of E to the *bottom* of the stack. Thus, if a stack has size b, then a pop gives rise to $|E| + 1$ successors.

It is easy to see that for the reachability set Z_b of the b-bounded PDS, the projection $\mathcal{T}(Z_b)$ overapproximates $\mathcal{T}(R)$, for any $b \geq 1$. Interestingly, for $b \to \infty$, the sequence of sets $\mathcal{T}(Z_b)$ forms a *descending* chain of overapproximations of $\mathcal{T}(R)$ that is guaranteed to converge, due to finiteness. However, the limit may still exceed the reachable visible states of $\mathcal{P}$ (i.e., $\mathcal{T}(R)$ is a lower bound of the sequence, but it may not be the greatest):

$$Z \;=\; Z_1 \;=\; \mathcal{T}(Z_1) \;\supseteq\; \mathcal{T}(Z_2) \;\supseteq\; \ldots \;\supseteq\; \lim_{b\to\infty} \mathcal{T}(Z_b) \;\overset{?}{\supsetneq}\; \mathcal{T}(R). \tag{3}$$

In particular, in each of the b-bounded systems, a spurious bottom-of-stack symbol can eventually percolate to the top, resulting in a pair $\langle q \,|\, \sigma\rangle$ that does not exist as a reachable visible state of the PDS. However, because this pair is contained in each set $\mathcal{T}(Z_b)$, it is also contained in $\lim_{b\to\infty} \mathcal{T}(Z_b)$.

4.2.4 CUBA via Convergence Detection. We summarize the process of verifying property C using $(\mathcal{T}(R_k))_{k=0}^{\infty}$ in Algorithm 3 (we assume R_{-1} and R_0 are suitably initialized). The algorithm differs from Scheme 1 mostly in the convergence test in Line 6: when reaching a *new* plateau at $k - 1$ (since the $\mathcal{T}$ sets are finite, we check for set equality efficiently via their cardinalities), it launches the generator test. If that fails, we will not perform the test again until reaching the next *new* plateau, if any: the sets $\mathcal{T}(R_k)$ do not change inside a plateau. This property also makes the algorithm tight: it stops at the minimal context bound k where $(\mathcal{T}(R_k))_{k=0}^{\infty}$ converges.

The algorithm is partially correct (it does not lie) by the properties of generator sets. It may not terminate if C holds: if $\mathcal{G} \cap Z$ includes unreachable generators, this set will never be contained in $\mathcal{T}(R_k)$, causing the algorithm to run forever.

Example 17. We revisit the example in Figure 4. Applying Definition 12, we obtain $\mathcal{G} = \{\langle q_0 \mid 1, \varepsilon\rangle, \langle q_0 \mid 1, 6\rangle, \langle q_0 \mid 2, \varepsilon\rangle, \langle q_0 \mid 2, 6\rangle\}$. With set Z as computed in Example 16, we obtain $\mathcal{G} \cap Z = \{\langle q_0 \mid 1, \varepsilon\rangle, \langle q_0 \mid 1, 6\rangle\}$. We now start Algorithm 3. The first plateau is reached at $k = 2$, but $\mathcal{G} \cap Z \setminus \mathcal{T}(R_2) = \{\langle q_0 \mid 1, 6\rangle\} \neq \emptyset$. Because visible state $\langle q_0 \mid 1, 6\rangle$ may be reached in the future, we must continue. The next plateau is reached at $k = 5$. Now we indeed have $\mathcal{G} \cap Z \subseteq \mathcal{T}(R_5)$ (visible state $\langle q_0 \mid 1, 6\rangle$ was reached in round 4), so the algorithm terminates; we have $\mathcal{T}(R) = \mathcal{T}(R_5)$.

ALGORITHM 3: Verifying C for unbounded k using $(\mathcal{T}(R_k))_{k=0}^{\infty}$ and convergence detection

Input: A CPDS $\mathcal{P}$, property C
Output: Whether C holds for all context bounds k; if not, a value of k for which it is violated

```
1: compute the generator set G and the overapproximate visible-state reachability set Z
2: for k := 1 to ∞ do
3:     compute the set R_k of reachable states of P under context bound k
4:     if T(R_k) violates C then
5:         return "error reachable with context bound k"
6:     if |T(R_{k-2})| < |T(R_{k-1})| = |T(R_k)| and G ∩ Z ⊆ T(R_k) then
7:         return "C holds for any context bound"
```

To summarize, Algorithm 3 gives us a sound method to decide context-unbounded reachability in multi-threaded stack programs. Although the OS $(\mathcal{T}(R_k))_{k=0}^{\infty}$ is guaranteed to converge, we had to modify Scheme 1 to account for the possibility of stuttering.

5 CUBA FOR CONCURRENT BOOLEAN PROGRAMS

The approach to context-unbounded reachability analysis presented in Section 4 is designed for CPDSs and is thus applicable to any formalism that losslessly translates into such systems, including concurrent Boolean programs. The aforementioned translation CBP → CPDS encodes the values of the *pc* and all local variables of the procedure currently executed by a thread into a single stack-alphabet symbol.

Treating each alphabet symbol as a structure-less state, however, ignores information about individual local variables and the *pc*. The goal of this section is to demonstrate how such information can be taken advantage of in interprocedural control-flow analysis to make certain steps of CUBA more precise, how it improves the chances for convergence, and how it leads, in the best case, to a fragment of CBPs for which reachability analysis is decidable.

5.1 Reachable-State Approximation via Call-Return Matches

Recall the computation of set Z used in Algorithm 3 to detect convergence: it consists of the reachable states of a transition system $\mathcal{M}$ that overapproximates the stack of each PDS by just its top. Pushes simply overwrite the top. Pop actions incur loss of information; to remain sound, they are simulated by non-deterministically choosing all possible *emerging* stack symbols, such as those that the stack program ever places on the stack right before pushing another symbol on top of it. The determination of what emerging symbols are is made statically.

The precision of the overapproximation Z of $\mathcal{T}(R_k)$ is critical: unreachable generator states accidentally included in Z will *always* cause Algorithm 3 to fail: $\mathcal{G} \cap Z \subseteq \mathcal{T}(R_k)$ is then false for every k. As an example, Figure 6 shows a program for which this algorithm fails to detect convergence of $(\mathcal{T}(R_k))_{k=0}^{\infty}$ because set Z contains an unreachable generator, namely $\langle q_0 \mid 1, 7\rangle \in \mathcal{G} \cap Z$. The CPDS in Figure 6 was obtained, however, from a concurrent Boolean program. Does this program provide information that can be used to build a better program $\mathcal{M}$, giving rise to fewer unreachable generators in $\mathcal{G} \cap Z$ and possibly leading to convergence?

```
   decl s: enum {q0, q1, q2, q3, q4};
0  s := 0;

   void foo() {
1    wait(s=0); s := 1;
2    wait(s=3);
     if (*) { s := 0; goto 1; }
     else   { s := 2; goto 2; }
   }

   void bar1() {
3    wait(s=0 || s=1 || s=2);
     if (s=0) { return; }
     else { if (s=1) s := 2;
            else goto 6; }
4    wait(s=2); s := 3; bar1();
5    wait(s=0); return;
6    wait(s=2); s := 4; bar2();
7    wait(s=1); return;
   }

   void bar2() {
8    wait(s=1); return;
   }
```

Initial state: $\langle q_0 \mid 1, 3\rangle$

$$
\begin{aligned}
Q &= \{q_0, q_1, q_2, q_3, q_4\}\\
\Sigma_1 &= \{1, 2\}\\
\Delta_1 &= f_1 : \langle q_0 \mid 1\rangle \rightarrow \langle q_1 \mid 2\rangle\\
&\quad\ f_{2a} : \langle q_3 \mid 2\rangle \rightarrow \langle q_0 \mid 1\rangle\\
&\quad\ f_{2b} : \langle q_3 \mid 2\rangle \rightarrow \langle q_2 \mid 2\rangle\\
\Sigma_2 &= \{3, \ldots, 8\}\\
\Delta_2 &= b_{3a} : \langle q_0 \mid 3\rangle \rightarrow \langle q_0 \mid \varepsilon\rangle\\
&\quad\ b_{3b} : \langle q_1 \mid 3\rangle \rightarrow \langle q_2 \mid 4\rangle\\
&\quad\ b_{3c} : \langle q_2 \mid 3\rangle \rightarrow \langle q_2 \mid 6\rangle\\
&\quad\ b_4 : \langle q_2 \mid 4\rangle \rightarrow \langle q_3 \mid 35\rangle\\
&\quad\ b_5 : \langle q_0 \mid 5\rangle \rightarrow \langle q_0 \mid \varepsilon\rangle\\
&\quad\ b_6 : \langle q_2 \mid 6\rangle \rightarrow \langle q_4 \mid 87\rangle\\
&\quad\ b_7 : \langle q_1 \mid 7\rangle \rightarrow \langle q_1 \mid \varepsilon\rangle\\
&\quad\ b_8 : \langle q_4 \mid 8\rangle \rightarrow \langle q_1 \mid \varepsilon\rangle
\end{aligned}
$$

$$
\begin{aligned}
T_1 &= e_1 : \langle q_0 \mid 1\rangle \mapsto \langle q_1 \mid 2\rangle\\
&\quad\ e_2 : \langle q_3 \mid 2\rangle \mapsto \langle q_0 \mid 1\rangle\\
&\quad\ e_3 : \langle q_3 \mid 2\rangle \mapsto \langle q_2 \mid 2\rangle
\end{aligned}
$$

$$
\begin{aligned}
T_2 &= f_1 : \langle q_0 \mid 3\rangle \mapsto \langle q_0 \mid \varepsilon\rangle\\
&\quad\ f_2 : \langle q_0 \mid 3\rangle \mapsto \langle q_0 \mid 5\rangle\\
&\quad\ \boxed{f_3 : \langle q_0 \mid 3\rangle \mapsto \langle q_0 \mid 7\rangle}\\
&\quad\ f_4 : \langle q_1 \mid 3\rangle \mapsto \langle q_2 \mid 4\rangle\\
&\quad\ f_5 : \langle q_2 \mid 3\rangle \mapsto \langle q_2 \mid 6\rangle\\
&\quad\ f_6 : \langle q_2 \mid 4\rangle \mapsto \langle q_3 \mid 3\rangle\\
&\quad\ f_7 : \langle q_0 \mid 5\rangle \mapsto \langle q_0 \mid \varepsilon\rangle\\
&\quad\ f_8 : \langle q_0 \mid 5\rangle \mapsto \langle q_0 \mid 5\rangle\\
&\quad\ \boxed{f_9 : \langle q_0 \mid 5\rangle \mapsto \langle q_0 \mid 7\rangle}\\
&\quad\ f_{10} : \langle q_2 \mid 6\rangle \mapsto \langle q_4 \mid 8\rangle\\
&\quad\ f_{11} : \langle q_1 \mid 7\rangle \mapsto \langle q_1 \mid \varepsilon\rangle\\
&\quad\ f_{12} : \langle q_1 \mid 7\rangle \mapsto \langle q_1 \mid 5\rangle\\
&\quad\ \boxed{f_{13} : \langle q_1 \mid 7\rangle \mapsto \langle q_1 \mid 7\rangle}\\
&\quad\ f_{14} : \langle q_4 \mid 8\rangle \mapsto \langle q_1 \mid \varepsilon\rangle\\
&\quad\ \boxed{f_{15} : \langle q_4 \mid 8\rangle \mapsto \langle q_1 \mid 5\rangle}\\
&\quad\ f_{16} : \langle q_4 \mid 8\rangle \mapsto \langle q_1 \mid 7\rangle
\end{aligned}
$$

$$
\begin{aligned}
\mathcal{G} \cap Z = \{&\langle q_0 \mid 1, \varepsilon\rangle,\ \langle q_1 \mid 2, \varepsilon\rangle\\
&\langle q_0 \mid 1, 5\rangle,\ \langle q_1 \mid 2, 5\rangle\\
&\langle q_1 \mid 2, 7\rangle,\ \boxed{\langle q_0 \mid 1, 7\rangle}\}
\end{aligned}
$$

Fig. 6. A two-threaded program consisting of three procedures; threads 1 and 2 execute procedures foo and bar1, respectively (bar1 in turn calls bar2). The program declares a shared variable s of a five-valued *enum* type, which, for readability, we use to abbreviate an encoding using three Boolean variables (to abide by the grammar of Section 2.1). There are no local variables (other than the *pc*s). The formalization of this program as a CPDS is given in the middle in the figure. The transition sets of the finite-state programs $\mathcal{M}_{1,2}$ for overapproximating generators Z, as well as $\mathcal{G} \cap Z$, are given on the right.

In a CPDS obtained from a CBP, stacks model the processing of procedure calls as follows. Given a procedure call with a caller f and a callee g (i.e., a specific call to g inside the code of f) the *return location* (the *pc* of f right after the call to g) is stored in the current top stack frame, along with the values of all local variables. Then, a new stack frame that contains a fresh instance of g is created and pushed onto the top of the stack. When (and if) g terminates, this frame is destroyed, and the frame corresponding to the caller f emerges at the top of the stack. Procedure f continues its execution, with the *pc* pointing to the return location and all local variable values restored.

Inspired by earlier work on *nested word automata* [3], in this section we introduce an approach to computing Z based on information about procedure call and return pairs: we overapproximate pop actions by extracting the procedure g we are returning from and by only allowing emerging states whose *pc* points to a location after a call to g. For example, in Figure 6, a return from a call to bar2 only permits $pc = 7$. If this control-flow information were ignored, the corresponding pop would be modeled by $pc \in \{5, 7\}$.

Formalization. We define some notation. Given a stack symbol σ, we denote its two components, the program counter and the local-variable value tuple, by $\sigma.pc$ and $\sigma.l$, respectively. We call σ a *returning state for (some procedure)* f if $\sigma.pc$ points to a return statement from f and a *post-call state for* f if there is a (possibly interprocedural) control-flow edge from a call to f to location $\sigma.pc$. Let $\Sigma^r \subseteq \Sigma$ and $\Sigma^p \subseteq \Sigma$ be the sets of returning and post-call states for arbitrary procedures, respectively.

Definition 18. The *call-return relation* $\Omega \subseteq \Sigma^r \times \Sigma^p$ consists of all pairs (ρ, π) such that ρ is a returning state and π is a post-call state *for the same procedure.*

Ω relates each returning state ρ exactly to those post-call states whose *pc* points to a location reached right after a call to the procedure we are returning from in ρ. The relation is left-total, under two easy-to-enforce assumptions: (i) each procedure defined in the program is called at least once, and (ii) procedure calls in the program do not happen at the very end of the program but are always followed by a statement (which can be **skip**). The relation treats local-variable values non-deterministically.

Our approach now computes, for a given program, its call-return relation according to Definition 18 from the program text. We then use this information in the definition of structure $\mathcal{M}$, whose reachable states give rise to set Z, by allowing pop actions on returning states ρ to result in successor states π only when $(\rho, \pi) \in \Omega$.

Formally, we translate the given program $\mathbb{P}$ into a PDS $\mathcal{P} = (Q, \Sigma, \Delta, q^I)$ as before, but this time we compute the call-return relation Ω along the way. We then construct $\mathcal{M} = (Q \times \Sigma^{+\varepsilon}, T)$ as done in Algorithm 4. This algorithm is identical to Algorithm 2, except it does not compute a set E and instead takes emerging state candidates from the call-return relation Ω.

ALGORITHM 4: Build $\mathcal{M}$ used for computing Z, exploiting call-return information

Input: A sequential PDS $\mathcal{P} = (Q, \Sigma, \Delta, q^I)$ and a call-return relation Ω
Output: A sequential finite-state system $\mathcal{M} = (Q \times \Sigma^{+\varepsilon}, T)$

```
1: T = ∅
2: for each action (q, w) → (q', w') ∈ Δ do          ▷ w ∈ Σ^{+ε}, w' ∈ Σ^{≤2}
3:     T = T ∪ {(q, w) ↦ (q', T(w'))}
4:     if w ≠ ε ∧ w' = ε then                        ▷ a pop action
5:         for each (w, π) ∈ Ω do                    ▷ w ∈ Σ^r by def. of pop
6:             T = T ∪ {(q, w) ↦ (q', π)}
7: return M = (Q × Σ^{+ε}, T)
```

Given a concurrent Boolean program $\mathbb{P}^n = \mathbb{P}_1 \| \ldots \| \mathbb{P}_n$, we translate each $\mathbb{P}_i$ into a PDS $\mathcal{P}_i$ with individual call-return relations, using which we build $\mathcal{M}_i = (Q \times \Sigma_i^{+\varepsilon}, T_i)$, $1 \leq i \leq n$ via Algorithm 4. These compose to a finite-state concurrent system $\mathcal{M}^n = \mathcal{M}_1 \| \ldots \| \mathcal{M}_n$; as before, we define Z to be the set of reachable states of $\mathcal{M}^n$.

Lemma 19. *$\mathcal{T}(R) \subseteq Z$ for the reachability set Z of program $\mathcal{M}^n$ defined via Algorithm 4.*

Proof. By a simple adjustment to the proof of Lemma 15. Given $\mathcal{T}(t) \in \mathcal{T}(R)$, since the path Π to t is formed according to the semantic rules for programs with procedure calls and returns, any step $(q, w) \mapsto (q', \pi)$ along $\mathcal{T}(\Pi)$ (the visible-state projection of Π) that corresponds to a pop action remains a valid step in structure $\mathcal{M}^n$ obtained via Algorithm 4: the *pc* of π points to a post-call site of the procedure we are returning from in state w. All pop actions corresponding to procedure call returns have this form. As a result, $\mathcal{T}(t)$ is reachable in $\mathcal{M}^n$ and hence contained in Z (Algorithm 4). □

Example 20. For the program in Figure 6, the call-return relation is $\Omega = \{(3, 5), (5, 5), (7, 5), (8, 7)\}$. In set T_2, therefore, the boxed transitions can be excluded. For example, local state 3 permits only 5 as its successor. In particular, (unreachable) generator state $\langle q_0 \,|\, 1, 7\rangle$ does not appear in Z and hence not in $\mathcal{G} \cap Z$. As a result, $(\mathcal{T}(R_k))_{k=0}^{\infty}$ converges. Recall that the same sequence diverges for this program without using call-return information (see the discussion at the beginning of Section 5.1).

```
0 decl s := 0;

  void foo() {
1     decl l1, l2;
2     l1 := 0; l2 := !s;
3     bar(l2);
4     l2 := !l1; s := 0;
5     if (s != l2) foo();
6     return;
  }
```

```
  void foo() {
1     decl l1, l2;
2     l1 := 0; l2 := *;
3     bar(l2);
4     l2 := !l1; skip;
5     if (*) foo();
6     return;
  }
```

Fig. 7. Left: Parts of a multi-threaded program, with shared variable s and procedures foo and bar; the code for the latter is irrelevant for this example and omitted. Right: The program after soundly abstracting away s from foo. The shaded part highlights changes. The set of reachable local-variable valuations on the left is a subset of the corresponding set on the right.

5.2 Reachable-State Approximation via Local-State Aware Call-Return Matches

The definition of Ω, which relates states pointing to **return** statements to corresponding post-call states, can be refined to further increase the precision of the reachability set Z. In Section 5.1, Ω ignores the values of local variables, which—for the sake of soundness—means to treat them non-deterministically. An explicit-state implementation would have to enumerate all such valuations.

Although in general we cannot statically determine the precise set of possible values of local variables at the time of a procedure call, we may be able to soundly approximate this set, which would reduce the set of emerging states to be considered in Line 5 of Algorithm 4.

Given a sequential Boolean program, the possible valuations of the local variables at a call site depend on the values of the shared variables, which are modifiable by other PDSs. We can obtain an overapproximation of the set of such valuations by abstracting away shared variables in a sound way as follows:

- all *write* accesses (left-hand side of assignments) to shared variables are replaced by **skip**;
- all *read* accesses (right-hand side of assignments, uses in conditionals) are replaced by non-determinism.

(More precise abstractions are possible.) We can now perform reachability analysis on this modified sequential program (ignoring the shared variables) and obtain an overapproximation $R_L \subseteq \Sigma$ of the set of reachable tuples in $V_{pc} \times \{0,1\}^{|V_l|}$.

Example 21. Consider the program on the left in Figure 7. Using the overapproximation shown on the right, we can estimate the set of local-variable valuations reachable at Line 4, after the call to procedure bar. We obtain the following: $R_L|_{pc=4} = \{(4,0,0),(4,0,1)\}$: variable l1 has the value 0. This is an improvement compared to treating both l1 and l2 non-deterministically: when returning from the call to bar, we overapproximate the pop operation by non-deterministically selecting among the two valuations $\{(4,0,0),(4,0,1)\}$ instead of among four.

We use the set R_L to refine the call-return relation as follows.

Definition 22. The *local-state aware call-return relation* $\Omega_L \subseteq \Sigma^r \times \Sigma^p$ is given by

$$\Omega_L = \Omega \cap (\Sigma^r \times R_L) = \{(\rho,\pi) \in \Omega : \pi \in R_L\}. \tag{4}$$

In other words, we restrict the post-call states in the relation to be members of the reachability set R_L. Like Ω, Ω_L is left-total (under the same trivial assumptions). Algorithm 4, the finite-state

concurrent system $\mathcal{M}^n$, and its reachability set Z are defined as before, except they are now based on the (tighter) definition Ω_L instead of Ω.

Tightening Ω_L via live-variable analysis. The local-state aware call-return relation Ω_L is still not most precise: when returning from a procedure call, we only need to reinstate the values of local variables that are *live* at the post-call location: those whose value, along some continuation of the computation, is used but not overwritten before the first use. (Note that the use may be in an assertion.) We can assign *dead* local variables an arbitrary fixed value upon return from the procedure call.

To implement this idea, we first compute relation Ω_L and then restrict the post-call states in it such that each dead variable v has a unique reachable value $r(v)$. In other words, for each such v, the mapping r picks one fixed value, 0 or 1, among the values that v takes in all states in R_L.

Example 23. We revisit Example 21. Local variable l1 is live at Line 4, and its value is known to be 0. The value of l2 is unknown (the analysis using the approximate program on the right in Figure 7 does not reveal it), but this variable is dead: its value is overwritten (Line 4) before its first use (Line 5). We therefore do not need to treat it non-deterministically and instead assign it a value of $r(\mathsf{l2}) = 0$ when simulating the return from the call to bar. As a result, we are now left with a single local-variable valuation at location $pc = 4$: $(4, 0, 0)$.

We use these ideas and function r to refine the local-state aware call-return relation as follows.

Definition 24. The *live local-state aware call-return relation* $\Omega_{LL} \subseteq \Sigma^r \times \Sigma^p$ is given by

$$\Omega_{LL} = \{(\rho, \pi) \in \Omega_L : \forall v : v \text{ dead variable: } \pi.v = r(v)\}. \tag{5}$$

5.3 Decidable Fragments of CUBA for Concurrent Boolean Programs

Recall that the call-return relation Ω is left-total: it assigns to each returning state at least one matching post-call state. What if Ω is a left-total *function*—for instance, for each returning state, the post-call state can in fact be uniquely determined statically? Intuitively, this means that there is no need to keep the frame of the caller on the stack: upon encountering the **return** from the call, we can just replace the stack frame of the callee (which has terminated) by the unique post-call frame of the caller.

As an example, we revisit Figure 6. We focus on thread 2 (thread 1 has no recursion): it consists of two procedures, bar1 and bar2, each of which is called recursively in exactly one respective location (bar1 in Line 4 and bar2 in Line 6). In addition, both procedures have no local variables (other than the *pc*). This means that returning from either procedure, we know the exact local state $\sigma \in \Sigma$ (consisting only of the *pc*) from which we need to continue the execution of the caller. We can entirely eliminate the call stack.

As is turns out, the reachability set Z of $\mathcal{M}^n$ in this case not just overapproximates but in fact *equals* the set $\mathcal{T}(R)$ of reachable visible states. In other words, to compute this set, we do not need the OS approach, such as via Algorithm 3. Instead, after computing the finite-state program $\mathcal{M}_i$ using the individual call-return relations of the given Boolean programs, if they are *all* functional, we return the set of reachable states of the finite-state system $\mathcal{M}^n = \mathcal{M}_1 \| \ldots \| \mathcal{M}_n$ as the set of reachable visible states of the CPDS.

Theorem 25. *Reachability in concurrent Boolean programs for which the call-return relations Ω of all threads are functional is decidable.*

Proof. We prove the theorem by arguing that the procedure of computing Z in Section 4.2.3 is a decision procedure for the reachability of deterministic-return programs. We have already known

that the procedure terminates with a set $Z \supseteq \mathcal{T}(R)$. So we only need to prove $Z \subseteq \mathcal{T}(R)$ (i.e., for any $t \in Z$, $t \in \mathcal{T}(R)$).

Let $\bar{p}$ be a path in $\mathcal{M}^n = (\mathcal{M}_1, \ldots, \mathcal{M}_n)$, where $\mathcal{M}_i$ is constructed via Algorithm 4 for $1 \le i \le n$. Then, we can construct a corresponding path p in $\mathcal{P}^n$ such that

$$\begin{array}{lccccccccccc}
\bar{p} :: & \bar{t}^I = \langle q^I \,|\, \varepsilon, \ldots, \varepsilon \rangle & \overset{e_1}{\longmapsto} & \ldots & \overset{e_{l-1}}{\longmapsto} & \bar{t}_{l-1} & \overset{e_l}{\longmapsto} & \bar{t}_l & \overset{e_l}{\longmapsto} & \ldots & \overset{e_m}{\longmapsto} & \bar{t}_m = \bar{t} \\
\uparrow \mathcal{T} & \uparrow \mathcal{T} & & & & \uparrow \mathcal{T} & & \uparrow \mathcal{T} & & & & \uparrow \mathcal{T} \\
p :: & t^I = \langle q^I \,|\, \varepsilon, \ldots, \varepsilon \rangle & \overset{a_1}{\longrightarrow} & \ldots & \overset{a_{l-1}}{\longrightarrow} & t_{l-1} & \overset{a_l}{\longrightarrow} & t_l & \overset{e_l}{\longmapsto} & \ldots & \overset{a_m}{\longrightarrow} & t_m = t.
\end{array}$$

Path p is constructed inductively. With $\bar{t}^I$, obviously we have t^I such that $\bar{t}^I = \mathcal{T}(t^I)$. Suppose we have already constructed a path $t^I \to \ldots \to t_{l-1}$ satisfying the preceding condition. With the next transition $\bar{t}_{l-1} \mapsto \bar{t}_l$ triggered by e_l in $\mathcal{M}^n$, we have a corresponding action a_l from which e_l is derived in Algorithm 4. So we construct t_l as follows: if a_l is an **overwrite** or a **push** action, then by its semantics and the definition of $\mathcal{T}$, we get t_l by simply firing a_l in $\mathcal{P}^n$ from t_{l-1}; if a_l is a **pop**, then we can still uniquely determine t_l because $\mathcal{P}^n$ is deterministic-return, which implies, for any **pop**, the emerging stack symbol is uniquely defined after its execution. □

The theorem also holds when Ω is replaced by Ω_L or Ω_{LL}. Being tighter, these relations in fact have a higher chance of being functional than Ω.

Boolean programs with functional call-return relations. We identify some classes of Boolean programs for which Ω_{LL}, the tightest of our call-return relations, is functional. Generalizing the example in Figure 6, this is the case for programs such that

(1) each procedure has only one call site l (so that the *pc* of the post-call state can be determined statically), and
(2) each local variable is dead at location l, *or* its value can be determined statically (e.g., as done for Ω_L, without multi-threaded reachability analysis).

We can characterize such procedures as having *linear control flow*: the call graph follows production rules of a linear context-free grammar; the recursive calls correspond to non-terminal symbols, of which such grammars have at most one on the right-hand side of each rule.

A special case is a *right*-linear grammar: one where, for each rule, the right-hand side non-terminal symbol (if any) is the right-most symbol of the rule [22, Section 9.1]. Such grammars describe the (regular) control flow of *tail-recursive* procedures, where the post-call location is the end of the procedure. This special case is relevant because the preceding condition (2) holds trivially: all local variables are dead at the end of a procedure.

Corollary 26. *Reachability in tail-recursive concurrent Boolean programs is decidable.*

Each tail-recursive procedure can be translated into a **while** program, without a stack, making each thread finite-state. The composition of the threads is also finite-state; hence, the reachability problem for the concurrent system is decidable, in agreement with the preceding corollary. The value of Theorem 25 is that it generalizes the concept of tail recursion while maintaining decidability, namely to programs where each procedure satisfies preceding properties (1) and (2). After a call site, the program is free, among other things, to manipulate shared variables or to redefine local variables before their first use.

6 FINITE-CONTEXT REACHABILITY

Both *Scheme 1* (R_k) and Algorithm 3 are sound—but generally non-terminating—reachability analyzers for multi-threaded stack programs. There are, however, two differences: first, sequence

$(\mathcal{T}(R_k))_{k=0}^{\infty}$ projects from the reachable state space to a smaller set and can therefore converge faster, as we will demonstrate in Section 7. Second, because the sets $\mathcal{T}(R_k)$ are finite, we can always store them using cheaper data structures than the PSAs required for R_k, such as extensional containers or BDDs. In particular, no automaton-equivalence check is required for fixed-point detection. A blemish is that because we compute $\mathcal{T}(R_k)$ via projection from R_k, it seems that we still need to compute R_k in *automaton* form first. Or do we?

It turns out that, in many cases, we in fact do not. The insight is that although the set $R = \cup_{k=1}^{\infty} R_k$ of all reachable global states tends to be infinite in "truly pushdown" programs, it may be that *within every single context* only a finite number of new states are reachable. By induction, therefore, all sets R_k are finite; a condition we call *finite-context reachability*. For the class of CPDSs satisfying this condition, each round of our reachability algorithms *Scheme 1* (R_k) and Algorithm 3 operates over finite sets and thus can be implemented entirely without automata.

Example 27. Consider the CPDS in Figure 4. The maximum size of $\mathcal{P}_2$'s call stack after k contexts is roughly $k/2$, rendering the sets R_k finite; this CPDS satisfies FCR. In contrast, the set R of all reachable global states is infinite: the stack grows without bound.

If all sets R_k are finite, we can represent them using efficient and succinct finite-state data structures rather than automata. Subsequent uses of R_k then become feasible or even very efficient. For example, the convergence test $R_{k-1} = R_k$ in *Scheme 1* (R_k) can now be done in constant time by checking the sets' cardinalities. It is therefore worth investigating how we can decide the FCR condition from the given pushdown program.

Because each set R_k is *regular*, it is the language of some FSA. Finiteness of the language of an FSA is easily decidable (by checking whether every path from an initial state to an accepting state is simple). Hence, if we have the FSA that recognizes R_k, we can decide whether R_k is finite. So what is the problem? The problem is that we cannot even obtain the initial-states set of this FSA without solving the CUBA problem: that set is precisely R_{k-1}.

The solution is to work with approximations that are sufficient (but perhaps unnecessary) for finiteness. Our plan is as follows:

Step 1: We first show that if the set of states reachable in a sequential PDS **from all stacks of size ≤ 1** is finite—a condition that can be effectively checked, as we will discuss—then the set of states reachable from any *one* state with arbitrary stack size is finite.

Step 2: We use induction on k to show that if the preceding condition holds for all threads' PDSs, all sets R_k are finite.

To realize this plan, we need a small amount of additional notation. Let $\mathcal{P} = (Q, \Sigma, \Delta, ?)$ be a PDS (the initial shared state is irrelevant here). For a set of states $S \subseteq Q \times \Sigma^*$, let $R(S)$ denote the set of states reachable in $\mathcal{P}$ starting from S (so $S \subseteq R(S) \subseteq Q \times \Sigma^*$). For a state $s \in Q \times \Sigma^*$, we also write $R(s)$ to abbreviate $R(\{s\})$. Finally, we denote by $|s|$ the size of s's stack.

Lemma 28. *If $R(Q \times \Sigma^{+\varepsilon})$ is finite, then for every $s \in Q \times \Sigma^*$, $R(s)$ is finite.*

Proof. Assume $R(Q \times \Sigma^{+\varepsilon})$ is finite; we prove the claim on the right-hand side by strong induction on $|s|$. For $|s| \leq 1$, $s \in Q \times \Sigma^{+\varepsilon}$, so $R(s)$ is finite because $R(Q \times \Sigma^{+\varepsilon})$ is finite. For an arbitrary $s \in Q \times \Sigma^*$ with $|s| \geq 2$, we consider the reachability tree T spanned by the program from state s. We show that T is finite, which proves that $R(s)$ is finite. To this end, we use a classic result from graph theory known as *König's lemma* [27]:

A connected, infinite, and locally finite graph has an infinite simple path.[2]

The goal is to apply this lemma to T. Tree T is not connected, but as a reachability tree, it is "one-way connected" in the sense that all of its nodes are reachable from s. The 1927 work by König contains a stronger version of the lemma in which "connected" is replaced by "one-way connected." Furthermore, T is locally finite because the pushdown program Δ is finite. Given these two properties, the contrapositive of König's lemma tells us this: if T has only finite simple paths, then T is finite, as desired. What remains to show is therefore that any simple path p is finite. We show, equivalently, that p visits only finitely many distinct states. We distinguish two cases:

(a) If there exists a state t along p with $|t| < |s|$, then by the induction hypothesis, the suffix of p from t can only reach finitely many distinct states. Because the prefix of p up to t is finite, the whole path p, starting from s, also reaches only finitely many distinct states.

(b) Otherwise, the stack size along p *never* decreases below $|s|$. The bottom frame of s's stack thus never moves to the top (because $|s| \geq 2$, the bottom and top frames are different). The bottom frame thus has no impact on the execution of p but is "carried around" in all states along p. Formally, let p' be the sequence of states obtained from p after removing the bottom frame in every state; p' is a valid path in $\mathcal{P}$. Moreover, p''s initial state s' satisfies $|s'| = |s| - 1$; thus, by the induction hypothesis, p' can only reach finitely many distinct states. Call the set of these states $R_{p'}$. The reachable states along p, then, are obtained as

$$R_p = \{\langle q \,|\, w\sigma\rangle : \langle q \,|\, w\rangle \in R_{p'}\},$$

where σ is the bottom stack symbol of s. $R_{p'}$ is a finite set, and hence so is R_p. □

Lemma 28 implies that if from stacks of size 1 only finitely many states are reachable, then for any set $S \subseteq Q \times \Sigma^*$, $R(S)$ is can be infinite only in the trivial case when S itself is infinite.

We now move on to Step 2 of the plan mentioned earlier to decide FCR, which leads to our main result in this section.

Theorem 29. *If for all $i \in \{1, \ldots, n\}$ the reachability set $R(Q \times \Sigma_i^{+\varepsilon})$ of the i^{th} PDS starting from any stack of size ≤ 1 is finite, then for every k, R_k is finite.*

Proof. By induction on k. The condition is true for $k = 0$, because R_0 is the set of initial states, which is a singleton by the definition of PDSs. Now assume R_k is finite; its elements are states of the form $\langle q \,|\, w_1, \ldots, w_n\rangle$. R_{k+1} is obtained by going through these finitely many states and firing each of the finitely many threads on its corresponding thread state until completion (i.e., as the following finite union):

$$R_{k+1} = \bigcup_{\substack{\langle q \,|\, w_1, \ldots, w_n\rangle \in R_k, \\ i \in \{1, \ldots, n\}}} \{\textit{global states reachable by thread } i \textit{ executing from } \langle q \,|\, w_i\rangle\}.$$

To show that R_{k+1} is finite, it therefore suffices to show that for every thread state $\langle q \,|\, w_i\rangle$ occurring in R_k and every i, thread i can only reach finitely many states from $\langle q \,|\, w_i\rangle$. This property follows from Lemma 28, because $R(Q \times \Sigma_i^{+\varepsilon})$ is finite and $\langle q \,|\, w_i\rangle \in Q \times \Sigma_i^*$. □

What remains to be discussed is how we decide whether, for all $i \in \{1, \ldots, n\}$, $R(Q \times \Sigma_i^{+\varepsilon})$ is finite. This property can be checked exactly: for each i, we build the PSA $\mathcal{A}_i$ for the PDS of thread i but with the (finite) initial states set $Q \times \Sigma_i^{+\varepsilon}$. We then check whether every path in $\mathcal{A}_i$ from an initial state to an accepting state is simple. In fact, this test is equivalent to checking the absence of

[2]A directed graph is *locally finite* if every node has only finitely many successors. A path is *simple* if it is loop-free and therefore visits any node at most once.

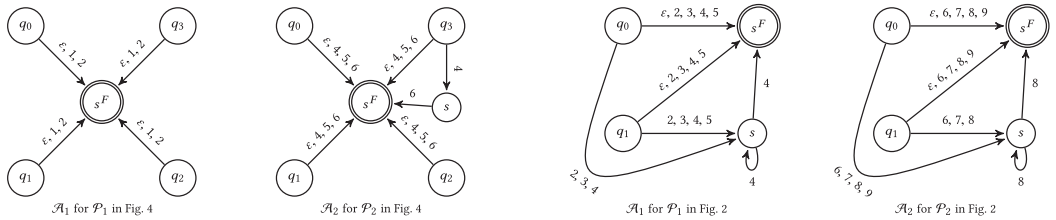

Fig. 8. Determining the FCR condition for the two CPDS in Figures 4 and 2 via Theorem 29. For $i \in \{1, 2\}$, PSA $\mathcal{A}_i$'s initial states are the shared states of $\mathcal{P}_i$; its accepting state is s^F. $\mathcal{A}_i$ accepts $R(Q \times \Sigma_i^{+\varepsilon})$. The absence of loops in the two automata for Figure 4 implies their languages, and hence $R(Q \times \Sigma_i^{+\varepsilon})$ are finite.

loops in the graph structure of $\mathcal{A}_i$: by the PSA construction [48], any loops in $\mathcal{A}_i$ are connected to the initial and final states sets. If the $\mathcal{A}_i$ have no loops, their languages and hence all sets $R(Q \times \Sigma_i^{+\varepsilon})$ are finite.[3] As a result, the system satisfies FCR.

As an example, for the two-thread program presented in Figure 4, we first determine that $\mathcal{P}_1$ has no function calls, so its stack size is constant. For $\mathcal{P}_2$, we build the PSA for the program and start-state set $Q \times \Sigma_i^{+\varepsilon}$ for $i = 1, 2$. The resulting graphs are loop-free, confirming that the CPDS satisfies FCR. Note that the stack size across contexts is unbounded. In contrast, the two-thread program presented in Example 10 does not satisfy FCR because there exist self-loops in both threads, as shown in the two right-most graphs of Figure 8.

7 CUBA IN PRACTICE

In this section, we first discuss the implementation of the proposed techniques in Section 7.1 and then present an empirical evaluation in Section 7.2. The evaluation is conducted using two types of experiments over a collection of benchmark programs. The goal is to evaluate both the effectiveness and efficiency of our techniques.

7.1 Implementation

We implemented the techniques described here in a verifier called Cuba, which offers three approaches:

1. Scheme 1 with explicit-state encoding (which requires FCR), denoted by *Scheme 1* (R_k),
2. Algorithm 3 with explicit-state encoding (which also requires FCR), denoted by Algorithm 3 ($\mathcal{T}(R_k)$), and
3. Algorithm 3 with state sets encoded using PSAs S_k (S = "Symbolic"), denoted by Algorithm 3 ($\mathcal{T}(S_k)$).

The overall procedure that follows shows how these three approaches are organized in Cuba. Given a CPDS $\mathcal{P}^n$ and a property C, it first determines whether FCR holds. If so, Cuba forks two computational threads: one runs visible-state reachability, and the other runs global-state reachability. It returns the answer of whichever terminates first, if any. Otherwise, Cuba runs visible-state reachability with PSA.

Algorithm 3 ($\mathcal{T}(S_k)$) is identical to Algorithm 3 ($\mathcal{T}(R_k)$), except it uses automata to represent infinite sets of states. It requires the computation of the set $\mathcal{T}(S_k)$ of reachable visible states (which is always finite and represented explicitly) from S_k. The latter is a finite set of symbolic states, which take the form $\tau = \langle q \,|\, \mathcal{A}_1, \ldots, \mathcal{A}_n \rangle$, where $q \in Q$ is a shared state and the $\mathcal{A}_i$ are PSAs. The

[3] A similar test is proposed by Bouajjani et al. [11], there in the context of *Büchi* PDSs, where the reachability of infinitely many states from some "green" set indicates acceptance.

Input: A CPDS $\mathcal{P}^n$ and a property C
1: **if** $\mathcal{P}^n$ satisfies FCR **then**
2: Algorithm 3. ($\mathcal{T}(R_k)$) || Scheme 1 (R_k) ▷ two threads
3: **else**
4: Algorithm 3. ($\mathcal{T}(S_k)$)

semantics of symbolic states is given by the concretization function γ, which maps τ to a set of concrete states:

$$\gamma(\tau) = \{\langle q \,|\, w_1, \ldots, w_n\rangle \in Q \times \Sigma_1^* \times \cdots \times \Sigma_n^* : \forall i \, \langle q \,|\, w_i\rangle \in L(\mathcal{A}_i)\}. \tag{6}$$

Function γ extends to sets S_k of symbolic states pointwise.

Function $\mathcal{T}$ applies to a symbolic state as follows:

$$\mathcal{T}(\langle q \,|\, \mathcal{A}_1, \ldots, \mathcal{A}_n\rangle) \; = \{q\} \times \mathcal{T}(\mathcal{A}_1) \times \cdots \times \mathcal{T}(\mathcal{A}_n), \tag{7}$$

where $\mathcal{T}(\mathcal{A}_i) \; = \; \{\mathcal{T}(w) \in \Sigma_i^{+\varepsilon} : \langle q \,|\, w\rangle \in L(\mathcal{A}_i)\}$, $1 \le i \le n$. Function $\mathcal{T}$ extends to sets of symbolic states pointwise. Set $\mathcal{T}(S_k)$ is thus a finite union of subsets of the finite set $Q \times \Sigma_1^{+\varepsilon} \times \cdots \times \Sigma_n^{+\varepsilon}$. To compute it, we enumerate the symbolic states $\langle q \,|\, \mathcal{A}_1, \ldots, \mathcal{A}_n\rangle$ in S_k and apply formula Equation (7). It remains to be described how we determine $\mathcal{T}(\mathcal{A}_i)$, which is the task of Algorithm 5.

ALGORITHM 5: $\mathcal{T}(\mathcal{A}_i)$

Input: A PSA $\mathcal{A}_i = (S_i, \Sigma_i, \delta_i, \{q\}, \{s^F\})$
Output: $\{\mathcal{T}(w) \in \Sigma_i^{+\varepsilon} \,|\langle q \,|\, w\rangle \in L(\mathcal{A}_i)\}$
1: $E := \emptyset$
2: **for each** pair (σ, q') s.t. $(q, \sigma, q') \in \delta_i$ **do** ▷ label σ is a symbol or ε
3: **if** there is a path from q' to s^F **then**
4: $E := E \cup \{\sigma\}$
5: **return** E

The **for** loop starting in Line 2 iterates over successors q' of q: any stack symbol σ appearing at the top of a stack of an accepted state (or ε if the stack is empty) also appears as a label of an edge leaving state q in $\mathcal{A}_i$. Symbol σ can be extended to an accepted word on the stack exactly if there is a path from the successor q' to s^F, as checked by Algorithm 5.

Cuba is implemented in C++. It takes a CPDS as input. A simple CPDS parser implements the functionality of verifying the FCR condition. Generators are computed up front. In the implementation of Algorithm 3 ($\mathcal{T}(R_k)$), we maintain two additional pieces of information in the data structure of reached global states: (i) the id of the thread that reaches the state and (ii) the context value k in which the state is reached. Set R_k is organized as a vector V of antichains, where each antichain $V[q]$ contains incomparable states with shared state q. Set $\mathcal{T}(Rk_k)$ is stored in the same way, but each antichain contains visible states. In contrast, Algorithm 3 ($\mathcal{T}(S_k)$) is implemented symbolically, based on PSAs. Both algorithms explore the state space in a breadth-first order. Both the source code and executables are available online [38].

7.2 Empirical Evaluation

Our experiments were designed to explore the following questions:

Q1: Is Cuba effective? Do the OSs (either $(R_k)_{k=0}^{\infty}$ or $(\mathcal{T}(R_k))_{k=0}^{\infty}$) eventually converge/can this be detected? If so, with small context bounds?

Q2: Is FCR effective in practice? In other words, is it applicable, and if so, will it benefit the computation of $(\mathcal{T}(R_k))_{k=0}^{\infty}$?

Q3: Is Cuba competitive against existing tools?

Experimental setup. For the evaluation, we collected a number of non-trivial concurrent Boolean programs, converted from C or Java programs using predicate abstraction either by us or done as part of previous work. Most of the programs are recursive. In total, the Boolean programs comprise roughly 2,600 lines of code, ranging from 36 to 362 lines. They include up to eight threads, featuring five shared and four local variables on average. We translated the concurrent Boolean programs into CPDSs as described in Section 2.2.3.

For each benchmark program, we consider a safety property like a race condition, specified via an assertion in the original program, or a visible state in the corresponding CPDS, respectively. We emphasize that both verification and falsification results obtained by Cuba are exact with respect to the input Boolean programs; there is no approximation (on top of that incurred by predicate abstraction).

The programs are organized into three suites:

- *01–03*: Three different versions of a Windows NT Bluetooth driver [16, 46, 49]. The driver has two types of threads: *stoppers*, which call a stopping procedure to halt the driver, and *adders*, which call a procedure to perform I/O. Note that the original version of the driver is not recursive; however, we use a recursive procedure to model the counter used in the program, as also done in previous work [16]. The authors report assertion violations caused by race conditions in versions 1 and 2 [46, 49]; in contrast, no violation occurs in version 3.
- *04–05*: Program 4 implements a binary search tree supporting concurrent manipulations [28]. Two types of threads are considered: an *inserter* inserts nodes into a tree, whereas a *searcher* searches for a node with a given value. Program 5 is an artificial benchmark converted from an online parallel file crawler that allows multiple users to recursively access files in a given directory.
- *06–14*: A set of examples taken from earlier publications: 6 and 9 from Prabhu et al. [43], 7 from Chaki et al. [16], 8 from Schwoon [48], and the parameterized programs 10–14 from the SLAM toolkit [6, 7, 26]. Because only very few programs from this toolkit are multi-threaded and recursive, we include some non-recursive but interesting programs (12–14). Program 14 is the largest in our collection: it contains two threads, with 11 procedures in each thread.

We conducted two types of experiments: (i) we performed state-reachability analysis with on-the-fly assertion checking on each benchmark to empirically answer Q1 and Q2. For unsafe examples, in addition to the context bound that revealed the error, we also report bounds on convergence for all reachable states (which happens later); (ii) we compared the performance of Cuba to that of JMoped [48, 49] to answer Q3. JMoped performs CBA; we pass to it the context bound at which Cuba terminates.

All experiments were performed on a 2.3-GHz Intel Xeon machine with 64 GB of memory, running 64-bit Linux. The timeout was set to 1 hour and the memory limit to 4 GB. All benchmarks are available online [38].

Evaluating Cuba *(experiments of type (i)).* The results are shown in Table 2. Column *FCR* shows that FCR holds in many of our examples. The various $k_{\max}$ columns show the effectiveness of CUBA using global-state or visible-state reachability. We first observe that Algorithm 3 terminates on all examples but one, where it runs out of memory. (This example features eight threads and requires a state set representation using PSAs, making the memory usage blow up.) Second, we observe that in most cases, k_{max} is small, often far less than 10. This result is good news because the resource

Table 2. Results

ID/Program	Prog. Features			$(R_k)_{k=0}^{\infty}$	$(\mathcal{T}(R_k))_{k=0}^{\infty}$		
	Thread	*FCR?*	*Safe?*	k_{max}	k_{max}	*Time*	*Mem*
01/Bluetooth-1	1 + 1	•	✗	≥7	6 (4)	0.26	18.14
	1 + 2	•	✗	≥7	6 (3)	2.32	136.26
	2 + 1	•	✗	≥8	7 (4)	12.76	347.74
02/Bluetooth-2	1 + 1	•	✗	≥7	6 (4)	0.53	23.43
	1 + 2	•	✗	≥7	6 (3)	4.39	196.73
	2 + 1	•	✗	≥8	7 (4)	14.21	387.23
03/Bluetooth-3	1 + 1	•	✓	≥7	6	0.47	22.15
	1 + 2	•	✓	≥7	6	4.71	180.11
	2 + 1	•	✓	≥8	7	14.46	375.42
04/BST-Insert	1 + 1	•	✓	2	2	1.17	24.53
	2 + 1	•	✓	3	3	15.84	140.93
	2 + 2	•	✓	≥5	4	45.21	355.74
05/FileCrawler	$1^{\bullet}$ + 2	•	✓	6	6	0.03	5.35
06/K-Induction	1 + 1	○	✓	≥4	3	0.23	3.78
07/Proc-2	2 + $2^{\bullet}$	○	✓	≥4	3	0.52	18.04
08/Stefan-1	2	○	✓	≥3	2	1.01	2.81
	4	○	✓	≥5	4	16.36	1185.62
	8	○	–	≥8	≥8	–	OOM
09/Dekker	$2^{\bullet}$	•	✓	≥6	6	0.21	13.42
10/Quick-Sort	2	•	✓	≥4	4	11.22	120.42
11/Inc-Counter	2	•	✓	≥5	5	20.62	153.63
12/P-NullCheck	$2^{\bullet}$	•	✓	≥4	4	14.95	86.29
13/P-TwoFiles	$2^{\bullet}$	•	✗	≥5	5 (4)	4.20	61.65
14/P-SlicTalk	$2^{\bullet}$	•	✓	≥5	5	2584.03	1500.65

Thread = number of threads; $n + m^{\bullet}$ means there are two templates instantiated by n and m threads, respectively. Superscript $^{\bullet}$ indicates the thread is *non*-recursive. *FCR?* = •: FCR holds; *Safe?* = ✓: assertion holds; k_{max}: point of collapse of $(R_k)_{k=0}^{\infty}$ / $(\mathcal{T}(R_k))_{k=0}^{\infty}$, ≥ indicates the method was interrupted as the other reached a conclusion. *Time* = runtime (seconds); *Mem* = memory usage (MB). Parenthesized number = context bound revealing the bug.

cost increases quickly with increasing values of k. One reason is that we compute $(\mathcal{T}(R_k))_{k=0}^{\infty}$ precisely by projection from R_k; the latter's cost is exponential in k.

Comparing visible-state to global-state reachability methods, we observe that $(\mathcal{T}(R_k))_{k=0}^{\infty}$ generally collapses before $(R_k)_{k=0}^{\infty}$ (it is easy to see that it cannot collapse later). The only way $(R_k)_{k=0}^{\infty}$ can be more effective than $(\mathcal{T}(R_k))_{k=0}^{\infty}$ is when Algorithm 3 does not terminate but *Scheme 1* (R_k) does; however, there is no such case in our benchmarks.

Program 9 is recursion-free (the only one among our benchmarks). Interestingly, although the CUBA fragment over recursion-free programs is decidable, Algorithm 3 may still not terminate: we may still not be able to tell stuttering from convergence. However, *Scheme 1* (R_k) is guaranteed to terminate, but convergence will not be detected until the stack reaches its maximum depth.

Tool comparison (experiments of type (ii)). Both Cuba and JMoped detected the bugs in Bluetooth suites 1 and 2 (expected), and they did not identify any errors in Bluetooth-3. The running times are comparable, as seen in Figure 9. The key difference is that with about the same or fewer resources, Cuba was able to *prove the correctness* of Bluetooth-3 and BST-Insert, which provides a significant increase in assurance. As for the implementation, JMoped is built atop the Qadeer/Rehof algorithm [45] and PSAs. Our results show that an explicit-state approach (provided that FCR holds) is competitive and far easier to implement. Although we failed to get Getafix to run, the execution time it reports on the Bluetooth suite [35] is comparable to that of Cuba. However, as with JMoped, Getafix does not prove correctness.

8 RELATED WORK

Context-sensitive reachability analysis for concurrent recursive programs is undecidable [47], even with only two threads and finite-domain variables. To cope with undecidability, various remedies have been proposed. One is to limit the computational model so that analysis without a context bound becomes decidable. Restrictions of the threads' synchronization capabilities include no communication between threads [14], communication via a finite number of nested locks [23, 24], communication following a transactional policy [44], or communication via specific concurrent queuing systems [30]. In contrast, our solution applies to the general case of (finite-state)

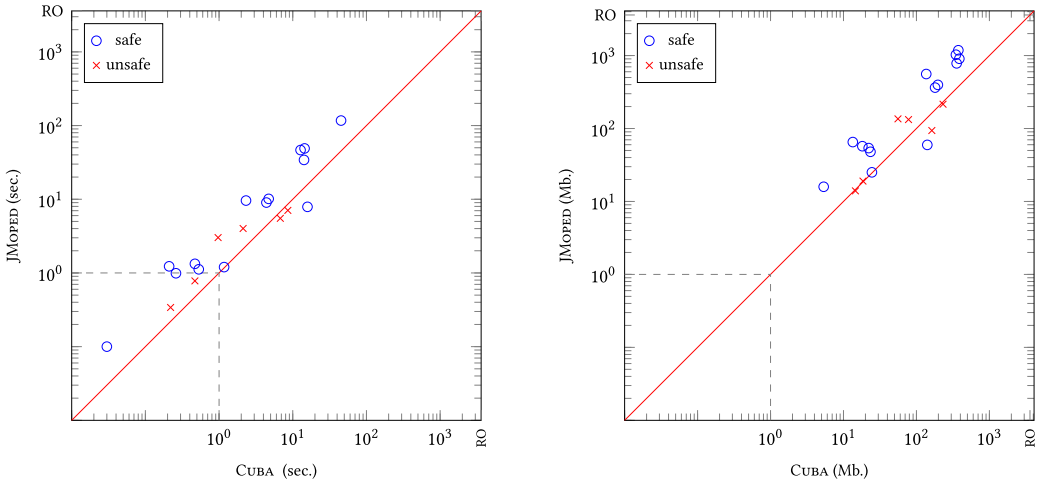

Fig. 9. Comparing performance of Cuba to JMoped on runtime (left) and memory usage (right). Unsafe: resources used until bug found; safe: until convergence. We ran the comparison only on benchmark suites 1–5 and 9 because none of the existing tools, to our knowledge, can properly translate the remaining programs to JMoped format.

shared-memory concurrency with arbitrary recursion depth. To make progress, we have forgone completeness of the analysis.

An alternative semi-decision procedure is presented by Chaki et al. [16], who target recursive concurrent message passing programs. They tackle the undecidability using a CEGAR scheme. In particular, they reduce the reachability queries of said programs to the undecidable problem of checking the disjointness of two context-free languages L_1 and L_2. To bypass the undecidability, the CEGAR scheme (1) computes the overapproximation A_1 and A_2 of L_1 and L_2; (2) if $A_1 \cap A_2 = \emptyset$, reports unreachability; and (3) otherwise checks for spuriousness and refines A_1 and A_2 if necessary.

Another remedy to the undecidability is to perform CBA, which underapproximates the set of reachable states by bounding the number of contexts. Originating in the work of Qadeer and Wu [46], CBA has emerged as a practical automatic formal analysis technique for shared-memory concurrent software [12, 13, 30]. The upside is that CBA reduces concurrent-software analysis to a decidable problem [45]. The downside is that a bug that requires more than that bound to manifest will slip through. Our approach is an attempt to eliminate such uncertainty by leveraging information learned across CBA runs with increasing bounds.

Atig et al. [5] suggested a stratified context-bounding method useful for programs with dynamic thread creation. The idea is to bound the number of contexts for each individual thread, but not the number of contexts for *all* threads, or the stack operations a thread can perform within a context. In contrast, our work aims at the orthogonal scenario of a fixed number of threads, with an arbitrary number of context switches between. La Torre et al. [32] proposed a partially correct but incomplete strategy that tries to prove all reachable states of a parameterized program are already reached under a k-round-robin schedule. Our approach does not enforce such a scheduling to programs.

Inspired by CBA, several fine-grained bounded analyses are proposed and corresponding decidability results are derived. Approaches include *phase-bounded* analysis [9, 29], where in each phase all pop actions are required to belong to a dedicated thread; *scope-bounded* analysis [33, 34], where the number of scopes (contexts between a procedure call and its return) is bounded; and a method

due to other works [4, 15], which assumes an ordering of the stacks and stipulates that a pop action is applied to the first non-empty stack. Similar to CBA, however, all of them underapproximate.

Prabhu et al. [43] proposed a semi-decision procedure to construct a proof of correctness via combining CBA and k-induction. As argued by those authors [43], k-induction requires considering paths of length k from arbitrary initial states, which makes the separation from error states non-trivial. Our approach is simpler, because it considers as parameter k the context bound itself and looks for converging sets of reachable states. A potential advantage of the work in Prabhu et al. [43] is that it investigates inductiveness of an invariant rather than the whole set of reachable states. If the goal is to prove a safety property that happens to be inductive while the set of reachable states does not converge, then our method will fail while that of Prabhu et al. [43] may succeed.

Reducing concurrent programs to sequential programs and then subjecting them to sequential verifiers is also an active research area [10, 21]. The KISS verifier [46] pioneered this approach via proposing a source-to-source translation of concurrent to sequential programs that underapproximates the behaviors of the original program. It limits the context switches to 2. Lal and Reps [36] and La Torre et al. [31] extended the bound from 2 to any fixed k. Emmi et al. [19] introduced delay-bounded scheduling and expanded on the capabilities of the exploration of Lal and Reps [36] with its ability to analyze programs with dynamic thread creation. A more general sequentialization framework is proposed by Bouijjani et al. [10]. Recent work [41, 42] on sequentialization proposed a translation that allows programs with unbounded contexts. However, it does not allow unbounded recursion. In summary, sequentialization is often more efficient than the Qadeer/Rehof algorithm [45]; several implementations exist. However, it assumes bounded context switching or bounded recursion and hence retains the limitations of CBA.

Verification of recursive programs is well understood, and tools have been designed and developed. However, most of them focus on sequential programs. JMoped [49] and Getafix [35] can handle concurrent recursive programs, but under context bounds. JMoped implements a BDD-based symbolic version of the Qadeer/Rehof algorithm [45]. Cuba also uses a variant of this algorithm for programs that do not satisfy FCR (only for $(\mathcal{T}(R_k))_{k=0}^{\infty}$). However, compared with our tool, the main difference is still that JMoped is built atop CBA and thus is mainly a bug-finding tool. Similarly, Getafix is a verifier based on fixed-point calculus and CBA.

The technique in this article can be viewed as solving a parameterized verification problem by determining *cutoffs*. These are bounds on the parameter(s) that provably suffice to draw conclusions about the unbounded-size program family. In almost all prior works that we are aware of, the cutoff parameter is the number of threads or processes [1, 8, 18]. OSs offer a unified approach for arbitrary (discrete) parameters. In addition, cutoffs are typically determined statically, often leaving them too large for practical verification. In contrast, our approach is akin to earlier *dynamic* strategies [1, 25]. The work of Kaiser et al. [25] aims at detecting convergence of sequences over thread-count parameters for solving the *decidable* problem of (essentially) local-state reachability in communicating finite-state machines, purely for efficiency.

9 CONCLUSION

In this article, we introduced the generic paradigm of OSs, which is intended to be used to analyze various kinds of resource-parameterized programs for the validity of a given (*safety*; more to follow) property for any parameter value. We applied the paradigm to the context-unbounded analysis (CUBA) problem for concurrent finite-state recursive procedures, for which reachability is undecidable. The paradigm resulted in a sound but incomplete method that can both refute and prove safety properties but may not terminate. Our results support the following conclusions: (i) for our benchmark programs, we were able to prove context-unbounded safety in about the same

time as, or less time than, previous methods used for CBA, and (ii) almost all of these programs exhibit small context bounds not only for error reporting but also, via sequence convergence, for proving correctness.

One reason for the practicality of our implementation is that the number of states reachable within one thread context—for instance, without passing control to another thread—is often finite, even if the overall number of reachable states is infinite. We have developed a sufficient condition for this FCR property.

Our approach to detecting convergence can be seen as an alternative to techniques based on abstract fixed point computation [17], which typically involve an abstract predicate transformer. Here, we instead compute reachable states of the pushdown program without further abstraction and then detect, using the novel concept of generators, when a set of suitable observations about these states converges as the resource parameter increases. We have laid out in Section 1 our motivation for proposing this approach.

The question of future work can split into work related to the CUBA problem and, more broadly, work related to the OS paradigm. As for the former, our computational model is (in principle) complete for fixed-thread concurrent programs with finite-state recursive threads. Dynamically created threads are common in practice but were excluded in this work because the focus was on eliminating the context-switch bound proposed earlier. Permitting a dynamic number of threads introduces another *resource* that likely needs to undergo OS treatment, thus requiring a convergence analysis for a multi-resource setting. How to establish convergence in the presence of multiple resources is an interesting and foundational question all by itself and is beyond the scope of this work.

Regarding broader future work on the OS paradigm, one question is whether the paradigm is really a paradigm. In other words, does it apply to a wide array of scenarios? A fresh instance of the technique has been successfully applied to systems of FIFO-communicating state machines, with the queue-length bound as the resource parameter [39], but we feel this only scratches the surface. In terms of properties, we have only applied the paradigm to safety-property verification so far. Liveness properties are related to reachable-cycle detection; how to approach this using our paradigm is an open question.

Finally, an interesting theoretical question is whether the FCR problem is decidable: we have only given a sufficient condition in this work.

REFERENCES

[1] Parosh Abdulla, Frédéric Haziza, and Lukáš Holík. 2016. Parameterized verification through view abstraction. *Int. J. Softw. Tools Technol. Transf.* 18, 5 (Oct. 2016), 495–516. DOI: https://doi.org/10.1007/s10009-015-0406-x

[2] Rajeev Alur and P. Madhusudan. 2004. Visibly pushdown languages. In *Proceedings of the 36th Annual ACM Symposium on Theory of Computing (STOC'04)*. ACM, New York, NY, 202–211. http://doi.acm.org/10.1145/1007352.1007390

[3] Rajeev Alur and P. Madhusudan. 2009. Adding nesting structure to words. *J. ACM* 56, 3 (May 2009), Article 16, 43 pages. http://doi.acm.org/10.1145/1516512.1516518

[4] Mohamed Faouzi Atig, Benedikt Bollig, and Peter Habermehl. 2008. Emptiness of multi-pushdown automata is 2ETIME-complete. In *Proceedings of the 12th International Conference on Developments in Language Theory (DLT'08)*. 121–133.

[5] Mohamed Faouzi Atig, Ahmed Bouajjani, and Shaz Qadeer. 2009. Context-bounded analysis for concurrent programs with dynamic creation of threads. In *Proceedings of the 15th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'09)*. 107–123. DOI: https://doi.org/10.1007/978-3-642-00768-2_11

[6] Thomas Ball and Sriram K. Rajamani. 2000. Bebop: A symbolic model checker for Boolean programs. In *Proceedings of the International SPIN Workshop on Model Checking of Software (SPIN'00)*. 113–130.

[7] Dirk Beyer. 2019. Sv-benchmarks. Retrieved October 16, 2020 from https://github.com/sosy-lab/sv-benchmarks/tree/master/clauses/BOOL.

[8] Jesse D. Bingham. 2005. A new approach to upward-closed set backward reachability analysis. *Electr. Notes Theor. Comput. Sci.* 138, 3 (2005), 37–48. DOI: https://doi.org/10.1016/j.entcs.2005.01.045

[9] Benedikt Bollig, Dietrich Kuske, and Roy Mennicke. 2017. The complexity of model checking multi-stack systems. *Theor. Comp. Sys.* 60, 4 (May 2017), 695–736. DOI: https://doi.org/10.1007/s00224-016-9700-6

[10] Ahmed Bouajjani, Michael Emmi, and Gennaro Parlato. 2011. On sequentializing concurrent programs. In *Proceedings of the 18th International Conference on Static Analysis (SAS'11)*. 129–145. http://dl.acm.org/citation.cfm?id=2041552.2041565.

[11] Ahmed Bouajjani, Javier Esparza, and Oded Maler. 1997. Reachability analysis of pushdown automata: Application to model-checking. In *Proceedings of the 8th International Conference on Concurrency Theory (CONCUR'97)*. 135–150. http://dl.acm.org/citation.cfm?id=646732.701281.

[12] Ahmed Bouajjani, Javier Esparza, Stefan Schwoon, and Jan Strejček. 2005. Reachability analysis of multithreaded software with asynchronous communication. In *Proceedings of the 25th International Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS'05)*. 348–359. DOI: https://doi.org/10.1007/11590156_28

[13] Ahmed Bouajjani, Séverine Fratani, and Shaz Qadeer. 2007. Context-bounded analysis of multithreaded programs with dynamic linked structures. In *Proceedings of the 19th International Conference on Computer-Aided Verification (CAV'07)*. 207–220. http://dl.acm.org/citation.cfm?id=1770351.1770383.

[14] Ahmed Bouajjani, Markus Müller-Olm, and Tayssir Touili. 2005. Regular symbolic analysis of dynamic networks of pushdown systems. In *Proceedings of the 16th International Conference on Concurrency Theory (CONCUR'05)*. 473–487. DOI: https://doi.org/10.1007/11539452_36

[15] Luca Breveglieri, Alessandra Cherubini, Claudio Citrini, and Stefano Crespi Reghizzi. 1996. Multi-pushdown languages and grammars. *Int. J. Found. Comput. Sci.* 07, 03 (1996), 253–291. https://doi.org/10.1142/S0129054196000191

[16] S. Chaki, E. Clarke, N. Kidd, T. Reps, and T. Touili. 2006. Verifying concurrent message-passing C programs with recursive calls. In *Proceedings of the 12th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'06)*. 334–349. DOI: https://doi.org/10.1007/11691372_22

[17] Patrick Cousot and Radhia Cousot. 1977. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL'77)*. 238–252.

[18] E. Allen Emerson and Vineet Kahlon. 2000. Reducing model checking of the many to the few. In *Proceedings of the 17th International Conference on Automated Deduction (CADE-17)*. 236–254. http://dl.acm.org/citation.cfm?id=648236.753642.

[19] Michael Emmi, Shaz Qadeer, and Zvonimir Rakamarić. 2011. Delay-bounded scheduling. In *Proceedings of the 38th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL'11)*. ACM, New York, NY, 411–422. DOI: https://doi.org/10.1145/1926385.1926432

[20] Alain Finkel, Bernard Willems, and Pierre Wolper. 1997. A direct symbolic approach to model checking pushdown systems. *Elec. Not. Theor. Comput. Sci.* 9 (1997), 27–37. DOI: https://doi.org/10.1016/S1571-0661(05)80426-8

[21] Naghmeh Ghafari, Alan J. Hu, and Zvonimir Rakamarić. 2010. Context-bounded translations for concurrent software: An empirical evaluation. In *Proceedings of the 17th International SPIN Conference on Model Checking Software (SPIN'10)*. 227–244. http://dl.acm.org/citation.cfm?id=1928137.1928160.

[22] John Hopcroft and Jeffrey Ullman. 1979. *Introduction to Automata Theory, Languages and Computation*. Addison-Wesley.

[23] Vineet Kahlon and Aarti Gupta. 2007. On the analysis of interacting pushdown systems. In *Proceedings of the 34th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL'07)*. ACM, New York, NY, 303–314. DOI: https://doi.org/10.1145/1190216.1190262

[24] Vineet Kahlon, Franjo Ivančić, and Aarti Gupta. 2005. Reasoning about threads communicating via locks. In *Proceedings of the 17th International Conference on Computer-Aided Verification (CAV'05)*. 505–518. DOI: https://doi.org/10.1007/11513988_49

[25] Alexander Kaiser, Daniel Kroening, and Thomas Wahl. 2010. Dynamic cutoff detection in parameterized concurrent programs. In *Proceedings of the 22nd International Conference on Computer-Aided Verification (CAV'10)*. 645–659. DOI: https://doi.org/10.1007/978-3-642-14295-6_55

[26] Anvesh Komuravelli, Arie Gurfinkel, and Sagar Chaki. 2014. SMT-based model checking for recursive programs. In *Proceedings of the 16th International Conference on Computer-Aided Verification (CAV'14)*. 17–34.

[27] Dénes König. 1927. *Über eine Schlussweise aus dem Endlichen ins Unendliche* (in German). *Acta Sci. Math. (Szeged)* 3, 2–3, 121–130.

[28] H. T. Kung and Philip L. Lehman. 1980. Concurrent manipulation of binary search trees. *ACM Trans. Database Syst.* 5, 3 (Sept. 1980), 354–382. DOI: https://doi.org/10.1145/320613.320619

[29] Salvatore La Torre, Parthasarathy Madhusudan, and Gennaro Parlato. 2007. A robust class of context-sensitive languages. In *Proceedings of the 22nd Annual IEEE Symposium on Logic in Computer Science (LICS'07)*. IEEE, Los Alamitos, CA, 161–170. DOI: https://doi.org/10.1109/LICS.2007.9

[30] Salvatore La Torre, P. Madhusudan, and Gennaro Parlato. 2008. Context-bounded analysis of concurrent queue systems. In *Proceedings of the 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'08)*. 299–314. http://dl.acm.org/citation.cfm?id=1792734.1792762.

[31] Salvatore La Torre, P. Madhusudan, and Gennaro Parlato. 2009. Reducing context-bounded concurrent reachability to sequential reachability. In *Proceedings of the 21st International Conference on Computer-Aided Verification (CAV'09)*. 477–492. DOI: https://doi.org/10.1007/978-3-642-02658-4_36

[32] Salvatore La Torre, P. Madhusudan, and Gennaro Parlato. 2010. Model-checking parameterized concurrent programs using linear interfaces. In *Proceedings of the 22nd International Conference on Computer-Aided Verification (CAV'10)*. 629–644. DOI: https://doi.org/10.1007/978-3-642-14295-6_54

[33] Salvatore La Torre and Margherita Napoli. 2011. Reachability of multistack pushdown systems with scope-bounded matching relations. In *Proceedings of the 22nd International Conference on Concurrency Theory (CONCUR'11)*. 203–218. http://dl.acm.org/citation.cfm?id=2040235.2040253.

[34] Salvatore La Torre, Margherita Napoli, and Gennaro Parlato. 2016. Scope-bounded pushdown languages. *Int. J. Found. Comput. Sci.* 27, 02 (2016), 215–233. DOI: https://doi.org/10.1142/S0129054116400074

[35] Salvatore La Torre, Madhusudan Parthasarathy, and Gennaro Parlato. 2009. Analyzing recursive programs using a fixed-point calculus. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'09)*. ACM, New York, NY, 211–222. DOI: https://doi.org/10.1145/1542476.1542500

[36] Akash Lal and Thomas Reps. 2009. Reducing concurrent analysis under a context bound to sequential analysis. *Form. Methods Syst. Des.* 35, 1 (Aug. 2009), 73–97. DOI: https://doi.org/10.1007/s10703-009-0078-9

[37] Peizun Liu and Thomas Wahl. 2018. CUBA: Interprocedural context-unbounded analysis of concurrent programs. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'18)*. 105–119.

[38] Peizun Liu and Thomas Wahl. 2018. Resource-Aware Program Analysis Using Observation Sequences: CUBA. Received December 1, 2019 from https://lpzun.github.io/rapa/apps/cuba/.

[39] Peizun Liu, Thomas Wahl, and Akash Lal. 2019. Verifying asynchronous event-driven programs using partial abstract transformers. In *Proceedings of the 31st International Conference on Computer-Aided Verification (CAV'19)*. 386–404.

[40] Shan Lu, Soyeon Park, Eunsoo Seo, and Yuanyuan Zhou. 2008. Learning from mistakes: A comprehensive study on real world concurrency bug characteristics. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XIII)*. ACM, New York, NY, 329–339. DOI: https://doi.org/10.1145/1346281.1346323

[41] Truc L. Nguyen, Bernd Fischer, Salvatore La Torre, and Gennaro Parlato. 2015. Unbounded lazy-CSeq: A lazy sequentialization tool for C programs with unbounded context switches. In *Proceedings of the 21st International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'15)*, Vol. 9035. 461–463. DOI: https://doi.org/10.1007/978-3-662-46681-0_45

[42] Truc L. Nguyen, Bernd Fischer, Salvatore La Torre, and Gennaro Parlato. 2016. Lazy sequentialization for the safety verification of unbounded concurrent programs. In *Proceedings of the 14th International Symposium on Automated Technology for Verification and Analysis (ATVA'16)*. 174–191. DOI: https://doi.org/10.1007/978-3-319-46520-3_12

[43] Prathmesh Prabhu, Thomas Reps, Akash Lal, and Nicholas Kidd. 2011. *Verifying Concurrent Programs via Bounded Context-Switching and Induction*. Technical Report TR-1701. Computer Sciences Department, University of Wisconsin, Madison, WI.

[44] Shaz Qadeer, Sriram K. Rajamani, and Jakob Rehof. 2004. Summarizing procedures in concurrent programs. In *Proceedings of the 31st ACM SIGPLAN Symposium on Principles of Programming Languages (POPL'04)*. ACM, New York, NY, 245–255. DOI: https://doi.org/10.1145/964001.964022

[45] Shaz Qadeer and Jakob Rehof. 2005. Context-bounded model checking of concurrent software. In *Proceedings of the 11th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'05)*. 93–107. DOI: https://doi.org/10.1007/978-3-540-31980-1_7

[46] Shaz Qadeer and Dinghao Wu. 2004. KISS: Keep it simple and sequential. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'04)*. ACM, New York, NY, 14–24. DOI: https://doi.org/10.1145/996841.996845

[47] G. Ramalingam. 2000. Context-sensitive synchronization-sensitive analysis is undecidable. *ACM Trans. Program. Lang. Syst.* 22, 2 (March 2000), 416–430. DOI: https://doi.org/10.1145/349214.349241

[48] Stefan Schwoon. 2000. *Model-Checking Pushdown Systems*. Ph.D. Dissertation. Lehrstuhl für Informatik VII der Technischen Universität München.

[49] Dejvuth Suwimonteerabuth, Javier Esparza, and Stefan Schwoon. 2008. Symbolic context-bounded analysis of multithreaded Java programs. In *Proceedings of the 15th International SPIN Workshop on Model Checking Software (SPIN'08)*. 270–287. DOI: https://doi.org/10.1007/978-3-540-85114-1_19

Received July 2019; revised April 2020; accepted August 2020