Context-Bounded Analysis of Concurrent Programs

Pascal Baumann
Max Planck Institute for Software Systems (MPI-SWS), Germany

Moses Ganardi
Max Planck Institute for Software Systems (MPI-SWS), Germany

Rupak Majumdar
Max Planck Institute for Software Systems (MPI-SWS), Germany

Ramanathan S. Thinniyam
Max Planck Institute for Software Systems (MPI-SWS), Germany

Georg Zetzsche
Max Planck Institute for Software Systems (MPI-SWS), Germany

Abstract

Context-bounded analysis of concurrent programs is a technique to compute a sequence of under-approximations of all behaviors of the program. For a fixed bound k, a context bounded analysis considers only those runs in which a single process is interrupted at most k times. As k grows, we capture more and more behaviors of the program. Practically, context-bounding has been very effective as a bug-finding tool: many bugs can be found even with small bounds. Theoretically, context-bounded analysis is decidable for a large number of programming models for which verification problems are undecidable. In this paper, we survey some recent work in context-bounded analysis of multithreaded programs.

In particular, we show a general decidability result. We study context-bounded reachability in a language-theoretic setup. We fix a class of languages (satisfying some mild conditions) from which each thread is chosen. We show context-bounded safety and termination verification problems are decidable iff emptiness is decidable for the underlying class of languages and context-bounded boundedness is decidable iff finiteness is decidable for the underlying class.

2012 ACM Subject Classification Theory of computation → Concurrency; Software and its engineering → Software verification

Keywords and phrases Context-bounded analysis, Multi-threaded programs, Decidability

Digital Object Identifier 10.4230/LIPIcs.ICALP.2023.3

Category Invited Talk

Funding This research was partially funded by the DFG project 389792660 TRR 248–CPEC and by the European Union (ERC, FINABIS, 101077902). Views and opinions expressed are however those of the author(s) only and do not necessarily reflect those of the European Union or the European Research Council Executive Agency. Neither the European Union nor the granting authority can be held responsible for them.

1 Introduction

Algorithmic verification of shared-state multithreaded programs is one of the main motivations for research in theoretical computer science. The general problem is undecidable, even when the class of programs is restricted in different ways. Thus, one direction of research has focused on finding decidable models that *over-approximate* the problem and another on finding *under-approximations*. An over-approximate model captures more behaviors than the original program; thus, if we find that the over-approximation has no bad behaviors, we can be certain that neither does the original program. An under-approximation, conversely, captures fewer behaviors. In this case, if we find a bad behavior in the approximation, we know that the bad behavior is also possible in the original program.

We consider a particular type of under-approximation: *context bounding*. Context-bounding is a technique to construct a parameterized sequence of under-approximations [46, 37]. For a fixed parameter k, a k-context-bounded analysis considers only those behaviors of the program in which an individual thread is interrupted by the scheduler at most k times. As k increases, more and more behaviors of the original program fall into the purview of the analysis. In the limit, all behaviors are covered.

Context-bounding has become a popular technique because of two reasons. For a wide class of programming models and verification questions, context-bounded analyses become decidable, even though the unrestricted problems are undecidable. Moreover, in practice, context-bounded analysis has had success as a bug finding tool, since many bugs in practical instances can be discovered even with small values of k [46, 44, 36, 34].

We focus on decidability questions. In order to avoid "trivial" encodings of Turing machines, we restrict programs to be *finite data*—that is, we assume each program variable to take on finitely many values. Even with this restriction, depending on the model of programs, decidability can be non-immediate because the state space of a program can be infinite in other respects, such as the stack of an individual thread or the number of pending threads.

Properties of concurrent programs For the moment, we focus on three decision problems: context-bounded *reachability* ("is there a k-bounded execution that reaches a specific global state?"), context-bounded *termination* ("all all k-bounded executions terminating?"), and context-bounded *boundedness* ("is there a bound on the number of pending threads along every k-bounded execution?"). We shall come back to other problems later.

Context-bounded analysis is a family of problems, depending on the model of concurrent programs as well as on the correctness properties considered. Qadeer and Rehof's original paper [46], that introduced context-bounding, stipulated that there is a fixed number of *recursive* threads that read or write shared variables but these threads do not spawn further threads. They showed that the reachability problem is NP-complete. Note that even with two threads, the reachability problem for finite-data programs is already undecidable: for example, we can encode the intersection non-emptiness problem for pushdown automata. On the other hand, if threads are not recursive, then the reachability problem is decidable without any context bounding restrictions, even if threads can spawn further threads: this can be shown by a reduction to the coverability problem for vector addition systems with states (VASS). Subsequently, Atig, Bouajjani, and Qadeer [11] extended decidability for context-bounded reachability when threads can spawn further threads. They showed an upper bound of 2EXPSPACE and a matching lower bound was shown by Baumann et al. [14]. Similar techniques show the same complexity for termination and boundedness.

A special case: Asynchronous programs The special case of $k = 0$ of context-bounded analysis is important enough to have its own name: *asynchronous programs*. In an asynchronous program, threads are executed atomically to completion (that is, never interrupted by the scheduler). Many software systems based on cooperative scheduling implement this model. Sen and Viswanathan [47] studied the model and showed reachability is decidable by reducing to a well-structured transition system. Ganty and Majumdar [28] showed that reachability, termination, and boundedness are all EXPSPACE-complete, by again reducing to coverability problems for VASS.

Majumdar, Thinniyam, and Zetzsche [40] proved decidability results for asynchronous programs in a general language-theoretic setting. They fix a class of languages $\mathcal{C}$, and consider asynchronous programs in which each individual thread is a language from the class $\mathcal{C}$ over the alphabet of thread names as well as a transformer over the global states. That is, each thread is a language (from $\mathcal{C}$) of words of the form dwd', where d and d' are global states and w is a sequence of thread names. The intent is that an atomic execution of the thread takes the global state from d to d' and also spawns new instances of all the threads in w.

They show that for all classes $\mathcal{C}$ satisfying a mild language-theoretic assumption (the class $\mathcal{C}$ is a *full trio*), safety and termination are decidable if and only if the underlying language class $\mathcal{C}$ has a decidable emptiness problem. Similarly, boundedness is decidable if and only if finiteness is decidable for $\mathcal{C}$. As a consequence, they get decidability results for asynchronous programs over context-free languages, higher-order recursion schemes, as well as other language classes studied in infinite-state verification.

Contribution Our starting point is the general approach of Majumdar, Thinniyam, and Zetzsche [40]. We show their general decidability results can be extended to context-bounded analysis (any $k \geq 0$). We define concurrent programs over a language class $\mathcal{C}$ and show analogous decidability results: (i) context-bounded reachability and context-bounded termination for programs are decidable if and only if $\mathcal{C}$ has a decidable emptiness problem, and (ii) context-bounded boundedness is decidable if and only if $\mathcal{C}$ has a decidable finiteness problem. As a consequence, we get a uniform proof for decidability for these problems for programs over context-free languages and for programs over higher-order recursion schemes.

The key argument in both settings is that of *downclosures* of languages under the subword ordering. Safety, termination, and boundedness are preserved if we "lose" some spawned threads, as long as the sequence of global state changes (and there are at most k of them for the fixed context bound k) is maintained. Since downclosures (even when maintaining a bounded number of distinguished letters) are always regular languages, this implies: If our concurrent program satisfies one of the above properties, then each thread can be over-approximated by a regular language so that the property is still satisfied. The decision procedure for reachability then runs two semi-decision procedures: one enumerates executions (to check for reachability) and the other enumerates regular languages and checks that (1) the thread languages are contained in the regular languages and (2) uses known decidability results for context-bounded reachability with regular thread languages.

The decision procedure does not, in particular, need to *construct* an explicit description of the downclosure. In fact, it even shows decidability for language classes for which downclosures cannot be constructed. On the flip side, we do not get complexity bounds.

Other properties What about other properties? Ganty and Majumdar showed fair termination for context-free asynchronous programs is decidable (by reduction to Petri net reachability) [28]. Majumdar, Thinniyam, and Zetzsche generalized the result to show that fair termination is equivalent to configuration reachability in the general setting [40]. On the other hand, decidability of fair termination implies the decidability of checking the "equal letters problem": deciding if a language in $\mathcal{C}$ has an equal number of as and bs. Thus, fair termination is undecidable for indexed languages. The undecidability is inherited by context-bounded fair termination. On the other hand, somewhat surprisingly, fair termination is decidable for context-bounded runs of context-free multithreaded programs [15].

2 Preliminaries

An *alphabet* is a finite non-empty set of *symbols*. For an alphabet Σ, we write Σ^* for the set of finite sequences of symbols (also called *words*) over Σ. A set $L \subseteq \Sigma^*$ of words is a *language*. By $\mathsf{pref}(L) = \{u \in \Sigma^* \mid \exists v \in \Sigma^* \colon uv \in L\}$ we denote the set of prefixes of words in L.

The *subword* order $\sqsubseteq$ on Σ^* is defined as follows: for $u, v \in \Sigma^*$ we have $u \sqsubseteq v$ if and only if u can be obtained from v by deleting some of v's letters. For example, $abba \sqsubseteq bababa$, but $abba \not\sqsubseteq baaba$. The *downclosure* (or *downward closure*) $\downarrow w$ of a word $w \in \Sigma^*$ with respect to the subword order is defined as $\downarrow w := \{w' \in \Sigma^* \mid w' \sqsubseteq w\}$. The downclosure $\downarrow L$ of a language $L \subseteq \Sigma^*$ is given by $\downarrow L := \{w' \in \Sigma^* \mid \exists w \in L \colon w' \sqsubseteq w\}$. An important fact is that the subword ordering $\sqsubseteq$ is a *well-quasi ordering* (Higman's lemma). A consequence is that the downclosure $\downarrow L$ of *any* language L is a regular language [32]. However, a representation for the downclosure of a language may not be effectively constructible.

The projection of a word $w \in \Sigma^*$ onto some alphabet $\Gamma \subseteq \Sigma$, written $\mathrm{Proj}_\Gamma(w)$, is the word obtained by erasing from w each symbol which does not belong to Γ. For a language L, define $\mathrm{Proj}_\Gamma(L) = \{\mathrm{Proj}_\Gamma(w) \mid w \in L\}$. We write $|w|_\Gamma$ for the number of occurrences of letters $a \in \Gamma$ in w, and similarly $|w|_a$ if $\Gamma = \{a\}$.

A *multiset* $\mathbf{m} \colon X \to \mathbb{N}$ over a set X maps each symbol of X to a natural number. The size $|\mathbf{m}|$ of a multiset $\mathbf{m}$ is given by $|\mathbf{m}| = \sum_{x \in X} \mathbf{m}(x)$. The set of all multisets over X is denoted $\mathbb{M}[X]$. We identify subsets of X with multisets in $\mathbb{M}[X]$ where each element is mapped to 0 or 1. We write $\mathbf{m} = [\![a, a, c]\!]$ for the multiset $\mathbf{m} \in \mathbb{M}[\{a, b, c, d\}]$ such that $\mathbf{m}(a) = 2$, $\mathbf{m}(b) = \mathbf{m}(d) = 0$, and $\mathbf{m}(c) = 1$. The Parikh image $\mathsf{Parikh}(w) \in \mathbb{M}[\Sigma]$ of a word $w \in \Sigma^*$ is the multiset such that for each letter $a \in \Sigma$ we have $\mathsf{Parikh}(w)(a) = |w|_a$.

Given two multisets $\mathbf{m}, \mathbf{m'} \in \mathbb{M}[X]$ we define $\mathbf{m} \oplus \mathbf{m'} \in \mathbb{M}[X]$ to be the multiset such that for all $a \in X$, we have $(\mathbf{m} \oplus \mathbf{m'})(a) = \mathbf{m}(a) + \mathbf{m'}(a)$. If $\mathbf{m}(a) \geq \mathbf{m'}(a)$ for all $a \in X$, we also define $\mathbf{m'} \ominus \mathbf{m} \in \mathbb{M}[X]$: for all $a \in X$, we have $(\mathbf{m} \ominus \mathbf{m'})(a) = \mathbf{m}(a) - \mathbf{m'}(a)$. For $X \subseteq Y$ we regard $\mathbf{m} \in \mathbb{M}[X]$ as a multiset in $\mathbb{M}[Y]$ where undefined values are mapped to 0.

Language Classes and Full Trios A *language class* is a collection of languages, together with some finite representation. Examples are the regular languages (e.g. represented by finite automata) or the context-free languages (e.g. represented by pushdown automata). A relatively weak and reasonable assumption on a language class is that it is a *full trio*, that is, it is closed under *rational transductions*. Equivalently, a language class is a full trio if it is closed under each of the following operations: taking intersection with a regular language, taking homomorphic images, and taking inverse homomorphic images [16].

We assume that all full trios $\mathcal{C}$ considered in this paper are *effective*: Given a language L from $\mathcal{C}$, a regular language R, and a homomorphism h, we can compute a representation of the languages $L \cap R$, $h(L)$, and $h^{-1}(L)$ in $\mathcal{C}$.

Many classes of languages studied in formal language theory form effective full trios. These include the regular and the context-free languages [33], the indexed languages [2, 25], the languages of higher-order pushdown automata [42], higher-order recursion schemes [31, 24, 40], Petri nets [29, 35], and lossy channel systems. However, the class of deterministic context-free languages is not a full trio: this class is not closed under rational transductions.

3 A Language-Theoretic Model of Concurrent Programs

Intuitively, a concurrent program consists of a shared global state and a finite number of thread names. Instances of thread names are called threads. A configuration of such a program consists of the current value of the global state and a multiset of partially-executed threads. A non-deterministic scheduler picks a partially-executed thread and runs it for some number of steps. An executing thread can change the global state. It can also spawn new threads—these can be picked and executed by the scheduler (in any order) in the future. When a scheduler swaps a running thread for another one, we say that there is a context switch. In our formal model, we keep the global state explicit and we model the execution behavior of threads as languages. The language of a thread captures the new threads it can spawn, as well as the effect of the execution on the global state.

3.1 Model

Let $\mathcal{C}$ be an (effective) full trio. A *concurrent program* (CP) over $\mathcal{C}$ is a tuple $\mathfrak{P} = (D, \Sigma, (L_a)_{a\in\Sigma}, d_0, \mathbf{m}_0)$, where D is a finite set of *global states*, Σ is an alphabet of *thread names*, $(L_a)_{a\in\Sigma}$ is a family of languages from $\mathcal{C}$ over the alphabet $\Sigma_D = D \cup \Sigma \cup (D \times D)$, $d_0 \in D$ is an *initial state*, and $\mathbf{m}_0 \in \mathbb{M}[\Sigma]$ is a multiset of *initial pending thread instances*. We assume that each L_a, $a \in \Sigma$, satisfies the condition $L_a \subseteq aD\big(\Sigma \cup (D \times D)\big)^* D$ (we provide the intuition behind this condition below).

A *configuration* $c = (d, \mathbf{m}) \in D \times \mathbb{M}[\Sigma_D^*]$ consists of a global state $d \in D$ and a multiset $\mathbf{m}$ of strings representing pending threads instances and partially executed threads. Given a configuration $c = (d, \mathbf{m})$, we write $c.d$ and $c.\mathbf{m}$ to denote the elements d and $\mathbf{m}$, respectively. The size of a configuration c is $|c.\mathbf{m}|$, i.e. the number of threads in the task buffer. We distinguish between threads that have been spawned but not executed (*pending threads*) and threads that have been partially executed (but swapped out). The pending thread instances are represented by single letters $a \in \Sigma$ (which corresponds to the name of the thread) while the partially executed threads of “type” $a \in \Sigma$ are represented by strings in $\mathsf{pref}(L_a)$ which end in a letter from $D \times D$.

Before presenting the formal semantics, let us provide some intuition. Suppose the current configuration is $(d, \mathbf{m})$. A non-deterministic scheduler picks one of the outstanding threads (either a pending thread $a \in \mathbf{m}$ or a partially executed thread $w \in \mathbf{m}$) and executes it for some time, until it terminates or until the scheduler decides to interrupt it. The execution of a thread a is abstractly modeled by the language L_a. A word $ad_1w_1(d_1', d_2)w_2(d_2', d_3)\dots(d_{k-1}', d_k)w_{k+1}d_{k+1} \in L_a$ represents a run of an instance of the thread a. The run starts executing in global state d_1. It spawns new threads $w_1 \in \Sigma^*$, then gets interrupted at global state d_1' by the scheduler. At some future point, the scheduler starts executing it again at global state d_2, when new threads w_2 are spawned before it is interrupted again at d_2'. The execution continues in this way until the thread terminates in global state d_{k+1}. Thus, the jump from one global state to another (from the perspective of the thread) when a context switch is made is represented by a letter from $D \times D$. The part of a run starting at global state d_i, spawning threads w_i and interrupted at d_i' is called a *segment*. Each interruption is called a *context switch*; the above word has k context switches.

Formally, the semantics of $\mathfrak{P}$ are given as a labelled transition system over the set of configurations with the transition relation $\Rightarrow \subseteq (D \times \mathbb{M}[\Sigma_D^*]) \times (D \times \mathbb{M}[\Sigma_D^*])$. The initial configuration is given by $c_0 = (d_0, \mathbf{m}_0)$.

The transition relation is defined using rules of four different types shown below. All four types of rules are of the general form $d \xrightarrow{[\![w]\!], \mathbf{n}'} d'$. A rule of this form allows the program to move from a configuration $(d, \mathbf{m})$ to configuration $(d', \mathbf{m}')$, i.e., $(d, \mathbf{m}) \Rightarrow (d', \mathbf{m}')$, iff $d \xrightarrow{[\![w]\!], \mathbf{n}'} d'$ matches a rule and $(\mathbf{m} \ominus [\![w]\!]) \oplus \mathbf{n}' = \mathbf{m}'$. Note that due to the definition of $\ominus$, $\mathbf{m}$ has to contain w for the rule to be applicable. We also write $\xRightarrow{w}$ to specify the particular

w used in the transition. As usual, the reflexive transitive closure of $\Rightarrow$ is denoted by $\Rightarrow^*$. A configuration c is said to be *reachable* if $c_0 \Rightarrow^* c$.

(R1) $d \xrightarrow{[\![a]\!],\mathsf{Parikh}(w)\oplus[\![adw(d',d'')]\!]} d'$ if $\exists w \in \Sigma^* \colon adw(d',d'') \in \mathsf{pref}(L_a)$.
Rule (R1) allows us to pick some thread a from $\mathbf{m}$ and atomically execute it until the point it is switched out by the scheduler. Note that the final letter (d', d'') of the thread indicates that it has been switched out at global d' and can be resumed when the global state is d''.
(R2) $d \xrightarrow{[\![a]\!],\mathsf{Parikh}(w)} d'$ if $\exists w \in \Sigma^* \colon adwd' \in L_a$.
Rule (R2) allows us to pick some thread a from $\mathbf{m}$ and atomically execute it to completion.
(R3) $d \xrightarrow{[\![aw'(d'',d)]\!],\mathsf{Parikh}(w)\oplus[\![aw'(d'',d)w(d',d''')]\!]} d'$ if $\exists w \in \Sigma^* \colon aw'(d'',d)w(d',d''') \in \mathsf{pref}(L_a)$
Rule (R3) allows us to pick some partially executed thread and execute it atomically until the point it is switched out by the scheduler.
(R4) $d \xrightarrow{[\![aw'(d'',d)]\!],\mathsf{Parikh}(w)} d'$ if $\exists w \in \Sigma^* \colon aw'(d'',d)wd' \in L_a$
Rule (R4) allows us to pick some partially executed thread and execute it to completion.

3.2 Runs and Context-bounded Runs

A *prerun* of a concurrent program $\mathfrak{P} = (D, \Sigma, (L_a)_{a\in\Sigma}, d_0, \mathbf{m}_0)$ is a finite or infinite sequence $\rho = (e_0, \mathbf{n}_0), w_1, (e_1, \mathbf{n}_1), w_2, \ldots$ of alternating elements of configurations $(e_i, \mathbf{n}'_i) \in D \times \mathbb{M}[\Sigma_D^*]$ and strings $w_i \in \Sigma^*$.

The set of preruns of $\mathfrak{P}$ will be denoted $\mathsf{Preruns}(\mathfrak{P})$. Note that if two concurrent programs $\mathfrak{P}$ and $\mathfrak{P}'$ have the same global states D and alphabet Σ, then $\mathsf{Preruns}(\mathfrak{P}) = \mathsf{Preruns}(\mathfrak{P}')$. The length $|\rho|$ of a finite prerun ρ is the number of configurations in ρ.

A *run* of a CP $\mathfrak{P} = (D, \Sigma, (L_a)_{a\in\Sigma}, d_0, \mathbf{m}_0)$ is a prerun $\rho = (d_0, \mathbf{m}_0), w_1, (d_1, \mathbf{m}_1), w_2, \ldots$ starting with the initial configuration $(d_0, \mathbf{m}_0)$, where for each $i \geq 0$, we have $(d_i, \mathbf{m}_i) \xRightarrow{w_{i+1}} (d_{i+1}, \mathbf{m}_{i+1})$. The set of runs of $\mathfrak{P}$ is denoted $\mathsf{Runs}(\mathfrak{P})$.

For a number k, the run ρ is said to be k-context-bounded (k-CB for short) if for each $c_i = (d_i, \mathbf{m}_i) \in \rho$ and for each $w \in \mathbf{m}_i$, we have $|w|_{D\times D} \leq k$. The set of k-context-bounded runs of $\mathfrak{P}$ is denoted by $\mathsf{Runs}_k(\mathfrak{P})$. In the case of finite runs which reach a certain configuration c, We say a configuration c is *k-reachable* if there is a finite k-CB run ρ ending in c.

3.3 Decision Problems

We study the following decision problems.

▶ **Definition 1.**

- **CB Safety (Global state reachability):**
  Instance: A concurrent program $\mathfrak{P}$, a context-bound k and a global state $d_f \in D$.
  Question: Is there a k-reachable configuration c such that $c.d = d_f$? If so, d_f is said to be k-reachable *(in $\mathfrak{P}$) and* k-unreachable *otherwise.*
- **CB Boundedness:**
  Instance: A concurrent program $\mathfrak{P}$ and a context-bound k.
  Question: Is there an $N \in \mathbb{N}$ such that for every k-reachable configuration c we have $|c.\mathbf{m}| \leq N$? If so, the concurrent program $\mathfrak{P}$ is k-bounded*; otherwise it is* k-unbounded.
- **CB Termination:**
  Instance: A concurrent program $\mathfrak{P}$, a context-bound k.
  Question: Is $\mathfrak{P}$ k-terminating, that is, is every k-CB run of $\mathfrak{P}$ finite?

3.4 Orders on Runs and Downclosures

Intuitively, k-safety, k-termination, and k-boundedness are preserved when the multiset of pending threads is "k-lossy": pending threads can get lost and we only consider runs where each thread makes at most k context switches. This loss corresponds to these pending threads never being scheduled by the scheduler. However, if a run demonstrates reachability of a global state, or non-termination, or unboundedness, in the k-lossy version, it corresponds also to a k-CB run in the original problem (and conversely). We make this intuition precise by introducing an ordering on runs and defining the downclosure.

Let $w, w' \in \Sigma D\big(\Sigma \cup (D \times D)\big)^*\big(D \cup (D \times D)\big)$ be words with $w = adw_1e_1w_2e_2 \dots w_le_l$ and $w' = a'd'w'_1e'_1w'_2e'_2 \dots w'_le'_l$, where $a, a' \in \Sigma$, $d, d' \in D$, $e_l, e'_l \in D \cup (D \times D)$, $w_i, w'_j \in \Sigma^*$ for $i, j \in [1, l]$, and $e_i, e'_j \in D \times D$ for $i, j \in [1, l-1]$. We define the state-preserving order $\sqsubseteq_D$ by $w \sqsubseteq_D w'$ iff $a = a'$, $d = d'$, $e_i = e'_i$ for each $i \in [1, l]$, and $w_i \sqsubseteq w'_i$, that is, w_i is a subword of w'_i, for each $i \in [1, l]$. We denote the corresponding notion of state-preserving downclosure under this order by $\Downarrow$. Intuitively, the $\sqsubseteq_D$ relation is a subword ordering on words that preserves the initial letter in Σ and all occurrences of $D \cup (D \times D)$, but potentially loses letters from each segment—that is, newly spawned threads can be lost.

We use the order $\sqsubseteq_D$ to naturally define the order $\preceq_D$ on $\mathbb{M}[\Sigma_D^*]$ by induction: for $\mathbf{m}, \mathbf{m'} \in \mathbb{M}[\Sigma_D^*]$ with $|\mathbf{m}|, |\mathbf{m'}| \geq 1$, we have $\mathbf{m} \preceq_D \mathbf{m'}$ iff there are $\mathbf{n}, \mathbf{n'} \in \mathbb{M}[\Sigma_D^*]$, $w, w' \in \Sigma_D^*$ with $\mathbf{m} = \mathbf{n} \oplus [\![w]\!]$ and $\mathbf{m'} = \mathbf{n'} \oplus [\![w']\!]$ such that $\mathbf{n} \preceq_D \mathbf{n'}$ and $w \sqsubseteq_D w'$. Furthermore, for all $\mathbf{m} \in \mathbb{M}[\Sigma_D^*]$, we have $\emptyset \preceq_D \mathbf{m}$.

We define an order $\trianglelefteq$ on preruns as follows: For preruns $\rho = (e_0, \mathbf{n}_0), w_1, (e_1, \mathbf{n}_1), w_2, \dots$ and $\rho' = (e'_0, \mathbf{n}'_0), w'_1, (e'_1, \mathbf{n}'_1), w'_2, \dots$, we have $\rho \trianglelefteq \rho'$ iff $|\rho| = |\rho'|$, $e_i = e'_i, w_i \sqsubseteq_D w'_i$ and $\mathbf{n}_i \preceq_D \mathbf{n}'_i$ for each $i \geq 0$. The downclosure $\downarrow R$ of a set R of preruns of $\mathfrak{P}$ is defined as $\downarrow R = \{\rho \in \mathsf{Preruns}(\mathfrak{P}) \mid \exists \rho' \in R.\ \rho \trianglelefteq \rho'\}$.

We write $\Downarrow\mathsf{Runs}(\mathfrak{P})$ for the downclosure with respect to $\trianglelefteq$ restricted to valid runs.

Some properties of a concurrent program $\mathfrak{P}$ only depend on the downclosure $\Downarrow\mathsf{Runs}_k(\mathfrak{P})$ of the set $\mathsf{Runs}_k(\mathfrak{P})$ of k-CB runs of the program $\mathfrak{P}$. For these properties, we may transform the program $\mathfrak{P}$ to a program $\Downarrow_k\mathfrak{P}$ such that the latter is easier to analyze but retains the properties of the former.

▶ **Definition 2.** *For a language L_a of a* CP*, let*

$$\Downarrow_k L_a = \Bigg\Downarrow \left(L_a \cap \left(\bigcup_{i=0}^{k} \left(aD(\Sigma^* D \times D)^i \Sigma^* D \right) \right) \right)$$

For any CP *$\mathfrak{P} = (D, \Sigma, (L_a)_{a \in \Sigma}, d_0, \mathbf{m}_0)$ and number k, we define the* CP *$\Downarrow_k\mathfrak{P} = (D, \Sigma, (\Downarrow_k L_a)_{a \in \Sigma}, d_0, \mathbf{m}_0)$. In other words, $\Downarrow_k\mathfrak{P}$ is the program obtained by taking the state-preserving downclosure of those words in L_a which contain at most k context switches.*

Note that, by well-quasi-ordering arguments, for any fixed k, the languages L_a of $\Downarrow_k\mathfrak{P}$ are all *regular*.

▶ **Proposition 3.** *Let $\mathfrak{P} = (D, \Sigma, (L_c)_{c \in \mathfrak{C}}, d_0, \mathbf{m}_0)$ be a concurrent program. Then $\Downarrow\mathsf{Runs}_k(\mathfrak{P}) = \Downarrow\mathsf{Runs}(\Downarrow_k\mathfrak{P})$. In particular,*

1. *For every $d \in D$, $\mathfrak{P}$ can k-reach d if and only if $\Downarrow_k\mathfrak{P}$ can k-reach d.*
2. *$\mathfrak{P}$ is k-terminating if and only if $\Downarrow_k\mathfrak{P}$ is k-terminating.*
3. *$\mathfrak{P}$ is k-bounded if and only if $\Downarrow_k\mathfrak{P}$ is k-bounded.*

Clearly, every run in $\mathsf{Runs}_k(\mathfrak{P})$ is also in $\mathsf{Runs}(\Downarrow_k\mathfrak{P})$. Conversely, we can show by induction on the length of the run that for every run $\rho \in \mathsf{Runs}(\Downarrow_k\mathfrak{P})$ there is a run $\rho' \in \mathsf{Runs}(\mathfrak{P})$ such that $\rho \trianglelefteq \rho'$. The result follows.

4 Decidability Results

We now characterize full trios $\mathcal{C}$ for which decision problems for concurrent programs over $\mathcal{C}$ are decidable. We shall make use of the following decidability results about regular languages.

▶ **Theorem 4.** **1.** [28, 10] *CB Safety is decidable for concurrent programs over regular languages.*
2. [28, 15] *CB Boundedness and CB termination are decidable for concurrent programs over regular languages.*

In fact, the above problems are decidable even if there is no bound on the number of context switches. The result in [10] is stated for a model called *Dynamic networks of Concurrent Finite-state Systems* (DCFS), but it is easy to see that there is a polynomial time reduction for the problems of safety, termination and boundedness for CP over regular languages to the corresponding problems for DCFS. The paper [15] shows decidability of CB termination and CB boundedness for the model of dynamic networks of concurrent pushdown systems, of which DCFS is a special case. There is also a simple reduction of these problems to the corresponding results for the model of asynchronous programs [28].

Our first decidability result is the following.

▶ **Theorem 5.** *Let $\mathcal{C}$ be a full trio. The following are equivalent:*
(i) *CB Safety is decidable for concurrent programs over $\mathcal{C}$.*
(ii) *CB Termination is decidable for concurrent programs over $\mathcal{C}$.*
(iii) *Emptiness is decidable for $\mathcal{C}$.*

The implications "(i)⇒(iii)" and The implications "(ii)⇒(iii)" are immediate from corrsponding results for asynchronous programs [40], since context bounded analysis problems generalize the corresponding analysis for asynchronous programs.

Before we prove the next implication, let us introduce a bit of notation. For each $i \in \mathbb{N}$, let R_i be the regular language $R_i = \Sigma D\Sigma^*((D \times D)\Sigma^*)^i D$, $R'_i = \Sigma D\Sigma^*((D \times D)\Sigma^*)^i(D \times D)$, for each $l \in \mathbb{N}$ we define $\mathcal{R}_l = \bigcup_{i=0}^{l}(R_i \cup R'_i)$. For any language L and $k \in \mathbb{N}$, the language $L \cap \mathcal{R}_k$ captures those words in L that contain at most k context switches.

For the implication "(iii)⇒(i)", we construct two semidecision procedures (Algorithm 1): the first one searches for regular over-approximations A_a of each language L_a such that the program $\mathfrak{P}'$ obtained by replacing each L_a by the corresponding A_a is safe. We can check whether our current guess for $\mathfrak{P}'$ is safe using Theorem 4. By Proposition 3, we know that in case $\mathfrak{P}$ is safe, then there must exist such a safe regular over-approximation. Concurrently, the second procedure searches for a k-CB run reaching the target global state d which witnesses the negation. Clearly, one of the two procedures must terminate. Note that we use an emptiness check to ensure that our current guess for A_a includes the set $L_a \cap \mathcal{R}_k$.

To show "(iii)⇒(ii)", we need an algorithm for termination of concurrent programs. As in the case of safety, it consists of two semi-decision procedures. The one for termination works just like the one for safety: It enumerates regular over-approximations and checks if one of them terminates. The procedure for non-termination requires some terminology:

Predictions We will use a notion of *prediction*, which assigns to each configuration $(e, \mathbf{n})$ of a run a multiset of strings that encode not only the *past* of each thread (as is done in $\mathbf{n}$), but also its *future*. To do this, we define the alphabet $\Gamma_D = \Sigma_D \cup \{\#\}$ that extends Σ_D a fresh letter $\#$. We shall encode predictions using strings of the form $au\#v$, which encode a thread with name a, past execution au, and future execution v. Additionally, we extend the

Algorithm 1 Checking CB Safety

Input: Concurrent program $\mathfrak{P} = (D, \Sigma, (L_a)_{a\in\Sigma}, d_0, \mathbf{m}_0)$ over $\mathcal{C}$, context bound $k \in \mathbb{N}$, state $d \in D$

```
run concurrently
    begin                                          /* find a safe over-approximation */
        foreach tuple (A_a)_{a∈Σ} of regular languages A_a ⊆ Σ* do
            if (L_a ∩ R_k) ∩ (Σ*_D \ A_a) = ∅ for each a ∈ Σ then
                if 𝔓' = (D, Σ, (A_a)_{a∈Σ}, d_0, m_0) does not k-reach d then
                    return d is not reachable.
    begin                                          /* find a run reaching d */
        foreach prerun ρ of 𝔓 do
            if ρ is a k-CB run that reaches d then
                return d reachable.
```

order $\preceq_D$ to strings of the form $au\#v$ by treating $\#$ as a letter from $D \times D$ which is to be preserved. Let us make this precise.

Suppose ρ is a (finite or infinite) prerun $(e_0, \mathbf{n}_0), w_1, (e_1, \mathbf{n}_1), \ldots$. An *annotation for* ρ is a sequence $\mathbf{f}_0, \mathbf{f}_1, \ldots \in \mathbb{M}[\Gamma_D^*]$ of multisets of strings such that the sequence has the same length as ρ. If ρ is a run, then we say that the annotation $\mathbf{f}_0, \mathbf{f}_1, \ldots$ is a *prediction* if

1. each string occurring in $\mathbf{f}_0, \mathbf{f}_1, \ldots$ is of the form $au\#v$ such that $auv \in \Sigma_D^*$ and $auv \in L_a$
2. for each $i \geq 0$, the multisets $\mathbf{n}_i$ and $\mathbf{f}_i$ have the same cardinality and there is a bijection between $\mathbf{n}_i$ and $\mathbf{f}_i$ so that (i) each word au in $\mathbf{n}_i$ is in bijection with some word $au\#v$ in $\mathbf{f}_i$ and (ii) if au is the active thread when going from $(e_i, \mathbf{n}_i)$ to $(e_{i+1}, \mathbf{n}_{i+1})$ and $au\#v$ is its corresponding string $au\#v$ in $\mathbf{f}_i$, then the system executes the next segment in v.

Note that then indeed, for each thread, its string in $\mathbf{n}_i$ records its past spawns, whereas the corresponding string in $\mathbf{f}_i$ contains all its future spawns (and possibly an additional suffix).

Of course, for each (finite or infinite) run, there exists a prediction: Just take the sequence of actions of each thread in the future. Moreover, taking a prefix of both a run and some accompanying prediction will yield a (shorter) run with a shorter prediction.

Self-covering runs Recall that for each alphabet Θ, we have an embedding relation $\preceq_D$ on the set $\mathbb{M}[\Theta_D^*]$, and in particular on $\mathbb{M}[\Gamma_D^*]$. We say that a finite run $(e_0, \mathbf{n}_0), w_1, (e_1, \mathbf{n}_1), \ldots, w_m, (e_m, \mathbf{n}_m)$, together with a prediction $\mathbf{f}_0, \ldots, \mathbf{f}_m$ is *k-self-covering* if for some $i < m$, we have $e_i = e_m, \mathbf{f}_i \preceq_D \mathbf{f}_m$, and also, all words in $\mathbf{n}_0, \mathbf{n}_1, \ldots$ contain at most k context-switches. As the name suggests, self-covering runs are witnesses for non-termination:

▶ **Lemma 6.** *For every $k \in \mathbb{N}$, a concurrent program has an infinite k-CB run if and only if it has a k-self-covering run.*

Here, it is crucial that for each $k \in \mathbb{N}$, the ordering $\sqsubseteq_D$ is a WQO on the set of words with at most k context-switches (on all of Σ_D^*, $\sqsubseteq_D$ is not a WQO).

We can now decide termination (Algorithm 2): the algorithm either (i) exhibits a k-self-covering run, which shows the existence of a k-bounded infinite run by Lemma 6, or (ii) finds a regular over-approximation that terminates, which means the original program is terminating. We can check termination of the regular over-approximation using Theorem 4. The algorithm also terminates: If there is an infinite k-bounded run, then Lemma 6 yields the existence of a k-self-covering run. Moreover, if the concurrent program does terminate,

Algorithm 2 Checking CB Termination

Input: Concurrent program $\mathfrak{P} = (D, \Sigma, (L_a)_{a\in\Sigma}, d_0, \mathbf{m}_0)$ over $\mathcal{C}$ and context bound $k \in \mathbb{N}$

```
run concurrently
    begin                                   /* find a terminating over-approximation */
        foreach tuple (A_a)_{a∈Σ} of regular languages A_a ⊆ Σ_D^* do
            if (L_a ∩ R_k) ∩ (Σ^* \ A_a) = ∅ for each a ∈ Σ then
                if P' = (D, Σ, (A_a)_{a∈Σ}, d_0, m_0) is k-terminating then
                    return P is k-terminating.
    begin                                                  /* find a self-covering run */
        foreach prerun ρ of P and an annotation σ do
            if ρ with σ is a k-self-covering run then
                return P is not k-terminating.
```

then Proposition 3 ensures the existence of a terminating regular over-approximation. This concludes our proof of Theorem 5.

Our second theorem is as follows.

▶ **Theorem 7.** *Let $\mathcal{C}$ be a full trio. The following are equivalent:*

(i) *CB Boundedness is decidable for concurrent programs over $\mathcal{C}$.*

(ii) *Finiteness is decidable for $\mathcal{C}$.*

The implication "(i)$\Rightarrow$(ii)" follows from the special case of asynchronous programs [40]. It was also observed in [40] that decidability of finiteness for $\mathcal{C}$ implies decidability of emptiness for $\mathcal{C}$. Further, by Theorem 5, we may assume that CB safety is decidable for CP over $\mathcal{C}$.

We now show the implication "(ii)$\Rightarrow$(i)". For a language $L \subseteq \Sigma_D^*$ and $n \in \mathbb{N}$, let $L|_n = L \cap \Sigma_D^{\leq n}$ be the language restricted to strings of length at most n and, in addition, for $k \in \mathbb{N}$, let $L'_a|_n = L_a|_n \cap \mathcal{R}_k$. Moreover, for an alphabet Θ, a language $L \subseteq \Theta^*$, and a word $w \in \Theta^*$, we define the left quotient of L by w as $w^{-1}L := \{u \in \Theta^* \mid wu \in L\}$. Our algorithm is based on the following characterization of unboundedness.

▶ **Lemma 8.** *The program $\mathfrak{P}$ is k-unbounded iff one of the two following conditions hold:*

(P1) *Either there exists some number n such that $\mathfrak{P}_n = (D, \Sigma, (L'_a|_n)_{a\in\Sigma}, d_0, \mathbf{m}_0)$ is unbounded, or*

(P2) *for some $a \in \Sigma$, there exists some word $w \in \mathsf{pref}(L_a)$ ending in a letter $(d, d') \in D \times D$ such that $\mathsf{pref}(w^{-1}L_a) \cap \Sigma^*$ is infinite and there exists a run ρ reaching a configuration c with $w \in c$ and $c.d = d'$.*

Essentially, (P1) captures the case where each thread spawns a finite number of other threads and (P2) the case that there is some reachable configuration at which a single thread can spawn an unbounded number of new threads. The above characterization allows us to implement Algorithm 3, which interleave three semidecision procedures: Checking properties (P1) and (P2) for positive certificates of unboundedness, as well as looking for certificates of boundedness by looking for bounded regular over-approximations. Here we can check boundedness for the latter by Theorem 4. Note that while checking for (P1), it is possible to compute each language $L'_a|_n$ explicitly since these languages are all finite. This is because, given any finite language $F \in \mathcal{C}$ and an explicitly given finite language A, we know $F = A$ iff $F \cap (\Sigma_D^* \setminus A) = \emptyset$ and for all $w \in A$, $F \cap \{w\} \neq \emptyset$, where the first condition checks if $F \subseteq A$ and the second if $A \subseteq F$. Therefore, by enumerating all strings w, we can build A iteratively.

Algorithm 3 Checking CB Boundedness

```
Input: Concurrent program 𝔓 = (D, Σ, (L_a)_{a∈Σ}, d_0, m_0) over 𝒞 and context bound k ∈ ℕ
run concurrently
    begin          /* (P1): Check if finite under-approximation is unbounded */
        foreach n ∈ ℕ do                     /* Explicitly find strings in L'_a|_n */
            foreach a ∈ Σ do
                X_a ← ∅, L'_a|_n ← L_a ∩ Σ_D^{≤n} ∩ 𝓡_k
                foreach w ∈ Σ_D^{≤n} do
                    if L'_a|_n ∩ {w} ≠ ∅ then
                        X_a ← X_a ∪ {w}
                    if L'_a|_n ∩ (Σ*_D \ X_a) = ∅ then
                        break
            if 𝔓_n = (D, Σ, (X_a)_{a∈Σ}, d_0, m_0) is unbounded then
                return 𝔓 unbounded.
    begin                   /* (P2): Check if unbounded segment can be reached */
        foreach prerun ρ of 𝔓, a ∈ Σ, w ∈ aDΣ*(D × DΣ*)^{≤k−2}(D × D) ∪ {a} do
            if ρ is a k-run that reaches c with w ∈ c, w = w'(d, d') where d' = c.d, and
             pref(w^{−1}L_a) ∩ Σ* is infinite then
                return 𝔓 unbounded.
            if ρ is a k-run that reaches c with w ∈ c, w = a where d' = c.d, and
             pref((wd')^{−1}L_a) ∩ Σ* is infinite then
                return 𝔓 unbounded.
    begin                              /* Find a bounded over-approximation */
        foreach tuple (A_a)_{a∈Σ} of regular languages A_a ⊆ (aΣ*_D ∩ 𝓡_k) do
            if (L_a ∩ 𝓡_k) ∩ (Σ*_D \ A_a) = ∅ for each a ∈ Σ then
                if 𝔓' = (D, Σ, (A_a)_{a∈Σ}, d_0, m_0) is bounded then
                    return 𝔓 bounded.
```

A Remark on Complexity Our procedures show decidability, but do not provide complexity results. For particular classes of languages, precise complexity bounds are known. For example, CB Safety, CB Termination, and CB Boundedness for concurrent programs over regular languages are all EXPSPACE-complete [28], and over context-free languages are 2EXPSPACE-complete [11, 14]. These bounds use explicit constructions of the downclosure. In particular, our results show decidability of the same problems for concurrent programs over higher-order recursion schemes. However, we do not get an explicit complexity bound. While there is an explicit construction of the downclosure of these languages [53, 30, 20], a precise complexity bound for the construction remains open.

5 Further Results

Other Decision Problems While we focus on safety, termination, and boundedness, there are decidability results for other properties and other classes of systems. The *fair termination* problem is a variant of termination, where we require that the scheduler is *fair*. Intuitively, a scheduler is fair if it schedules each partially executed thread that is infinitely often ready to execute. Context-bounded fair termination is decidable (but non-elementary) for context-free concurrent programs [15]. The problem is equivalent to Petri net reachability already for asynchronous programs [28]. It is undecidable for indexed languages.

Context-bounded analysis has also been studied for non-regular specifications. Lal et al. [38] showed decidability for context-bounded analysis for a subclass of weighted pushdown systems. Recently, Baumann et al. [13] studied the *context-bounded refinement problem* for non-regular specifications. In their setting, there is a fixed number of recursive (context-free) threads which also generate a language over a set of events. The specification is given by a Dyck language. They show that checking containment in the specification is coNP-complete, the same complexity as that of context-bounded safety verification, albeit requiring very different techniques. An analogous result was shown for the setting of asynchronous programs, but the complexity is EXPSPACE-complete [12].

Tools and Sequentialization A practical motivation for studying context-bounded reachability was that, empirically, many bugs in concurrent programs could be found with a small number of context switches. This led to the development of several academic and industrial tools, such as CHESS [44] and CSeq [27]. CHESS incorporated context bounding in an enumerative search. CSeq and several other tools implemented *sequentialization*: a preprocessing step that compiles the original concurrent program into a sequential program that preserves all k-context bounded runs, an idea going back to Lal and Reps [37]. Context-bounding was integrated with other exploration heuristics such as abstract interpretation and partial-order reduction [45, 21, 41].

Context-Bounded Analysis of Related Models Context-bounding was studied for other models of concurrency, such as parameterized state machines communicating through message-passing over a given topology [18], concurrent queue systems [49], programs over weak memory models [9, 1], abstract models such as valence automata [43], etc. In each case, the notion of "context" has to be refined based on the model.

Similar Restrictions The theory of context-bounding has inspired other natural bounds in the analysis of concurrent systems. For example, a well-studied restricion is *scope-bounding*: In a *k-scope-bounded* run, there can be an unbounded number of context-switches, but during the time span of a single function call (i.e. between a push and its corresponding pop), there can be at most k interruptions [52]. This covers more executions than context-bounding, which comes at the cost of PSPACE-completeness of safety verification [52]. Scope-boundedness has also been studied in terms of timed systems [4, 17], temporal-logic model-checking [6], resulting formal languages [51], and as an under-approximation for infinite-state systems beyond multi-pushdown systems [48].

Similarly, a *k-phase-bounded* run consists of k phases, in each of which at most one stack is popped [50, 8]. Another variant is *k-stage-bounded* runs: They consist of k stages, each of which allows only one thread to write to the shared memory, whereas the other threads can only read from it [7]. Further restrictions are *ordered multi-pushdown systems* [19, 5] and *delay-bounded scheduling* [26].

Powerful abstract notions of under-approximate analysis (which explain decidability of several concrete restrictions described above) are available in the concepts of *bounded tree-width* [39] and *bounded split-width* [3, 23, 22].

In conclusion, *context-bounding* is an elegant idea that has been very influential both in practice and in theory. In practice, it has been incorporated in several tools for automatic analysis of programs. Theoretically, it has led to a wealth of new models and analysis algorithms. At this point, the theory has marched ahead of implementations: it is an interesting open challenge to see how far the new algorithms can also lead to practical tools.

References

1 Parosh Aziz Abdulla, Mohamed Faouzi Atig, Ahmed Bouajjani, and Tuan Phong Ngo. Context-bounded analysis for POWER. In Axel Legay and Tiziana Margaria, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 23rd International Conference, TACAS 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings, Part II*, volume 10206 of *Lecture Notes in Computer Science*, pages 56–74, 2017. doi:10.1007/978-3-662-54580-5_4.

2 Alfred V. Aho. Indexed grammars - an extension of context-free grammars. *J. ACM*, 15(4):647–671, 1968. doi:10.1145/321479.321488.

3 C. Aiswarya, Paul Gastin, and K. Narayan Kumar. Verifying communicating multi-pushdown systems via split-width. In Franck Cassez and Jean-François Raskin, editors, *Automated Technology for Verification and Analysis - 12th International Symposium, ATVA 2014, Sydney, NSW, Australia, November 3-7, 2014, Proceedings*, volume 8837 of *Lecture Notes in Computer Science*, pages 1–17. Springer, 2014. doi:10.1007/978-3-319-11936-6_1.

4 S. Akshay, Paul Gastin, Shankara Narayanan Krishna, and Sparsa Roychowdhury. Revisiting underapproximate reachability for multipushdown systems. In *Tools and Algorithms for the Construction and Analysis of Systems - 26th International Conference, TACAS 2020, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2020, Dublin, Ireland, April 25-30, 2020, Proceedings, Part I*, volume 12078 of *Lecture Notes in Computer Science*, pages 387–404. Springer, 2020. doi:10.1007/978-3-030-45190-5_21.

5 Mohamed Faouzi Atig, Benedikt Bollig, and Peter Habermehl. Emptiness of ordered multi-pushdown automata is 2ETIME-complete. *Int. J. Found. Comput. Sci.*, 28(8):945–976, 2017. doi:10.1142/S0129054117500332.

6 Mohamed Faouzi Atig, Ahmed Bouajjani, K. Narayan Kumar, and Prakash Saivasan. Linear-time model-checking for multithreaded programs under scope-bounding. In Supratik Chakraborty and Madhavan Mukund, editors, *Automated Technology for Verification and Analysis - 10th International Symposium, ATVA 2012, Thiruvananthapuram, India, October 3-6, 2012. Proceedings*, volume 7561 of *Lecture Notes in Computer Science*, pages 152–166. Springer, 2012. doi:10.1007/978-3-642-33386-6_13.

7 Mohamed Faouzi Atig, Ahmed Bouajjani, K. Narayan Kumar, and Prakash Saivasan. On bounded reachability analysis of shared memory systems. In Venkatesh Raman and S. P. Suresh, editors, *34th International Conference on Foundation of Software Technology and Theoretical Computer Science, FSTTCS 2014, December 15-17, 2014, New Delhi, India*, volume 29 of *LIPIcs*, pages 611–623. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2014. doi:10.4230/LIPIcs.FSTTCS.2014.611.

8 Mohamed Faouzi Atig, Ahmed Bouajjani, K. Narayan Kumar, and Prakash Saivasan. Parity games on bounded phase multi-pushdown systems. In Amr El Abbadi and Benoît Garbinato, editors, *Networked Systems - 5th International Conference, NETYS 2017, Marrakech, Morocco, May 17-19, 2017, Proceedings*, volume 10299 of *Lecture Notes in Computer Science*, pages 272–287, 2017. doi:10.1007/978-3-319-59647-1_21.

9 Mohamed Faouzi Atig, Ahmed Bouajjani, and Gennaro Parlato. Context-bounded analysis of TSO systems. In Saddek Bensalem, Yassine Lakhnech, and Axel Legay, editors, *From Programs to Systems. The Systems perspective in Computing - ETAPS Workshop, FPS 2014, in Honor of Joseph Sifakis, Grenoble, France, April 6, 2014. Proceedings*, volume 8415 of *Lecture Notes in Computer Science*, pages 21–38. Springer, 2014. doi:10.1007/978-3-642-54848-2_2.

10 Mohamed Faouzi Atig, Ahmed Bouajjani, and Shaz Qadeer. Context-bounded analysis for concurrent programs with dynamic creation of threads. In *Proceedings of TACAS 2009*, pages 107–123, 2009.

11 Mohamed Faouzi Atig, Ahmed Bouajjani, and Shaz Qadeer. Context-bounded analysis for concurrent programs with dynamic creation of threads. *Log. Methods Comput. Sci.*, 7(4), 2011. doi:10.2168/LMCS-7(4:4)2011.

12 Pascal Baumann, Moses Ganardi, Rupak Majumdar, Ramanathan S. Thinniyam, and Georg Zetzsche. Checking refinement of asynchronous programs against context-free specifications. In *ICALP '23*, LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2023.

13 Pascal Baumann, Moses Ganardi, Rupak Majumdar, Ramanathan S. Thinniyam, and Georg Zetzsche. Context-bounded verification of context-free specifications. *Proc. ACM Program. Lang.*, 7(POPL):2141–2170, 2023. doi:10.1145/3571266.

14 Pascal Baumann, Rupak Majumdar, Ramanathan S. Thinniyam, and Georg Zetzsche. The complexity of bounded context switching with dynamic thread creation. In *47th International Colloquium on Automata, Languages, and Programming, ICALP 2020, July 8-11, 2020, Saarbrücken, Germany (Virtual Conference)*, volume 168 of *LIPIcs*, pages 111:1–111:16. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020. doi:10.4230/LIPIcs.ICALP.2020.111.

15 Pascal Baumann, Rupak Majumdar, Ramanathan S. Thinniyam, and Georg Zetzsche. Context-bounded verification of liveness properties for multithreaded shared-memory programs. *Proc. ACM Program. Lang.*, 5(POPL):1–31, 2021. doi:10.1145/3434325.

16 Jean Berstel. *Transductions and context-free languages*. Springer-Verlag, 1979.

17 Devendra Bhave, Shankara Narayanan Krishna, Ramchandra Phawade, and Ashutosh Trivedi. On timed scope-bounded context-sensitive languages. In *Developments in Language Theory - 23rd International Conference, DLT 2019, Warsaw, Poland, August 5-9, 2019, Proceedings*, volume 11647 of *Lecture Notes in Computer Science*, pages 168–181. Springer, 2019. doi:10.1007/978-3-030-24886-4_12.

18 Benedikt Bollig, Paul Gastin, and Jana Schubert. Parameterized verification of communicating automata under context bounds. In Joël Ouaknine, Igor Potapov, and James Worrell, editors, *Reachability Problems - 8th International Workshop, RP 2014, Oxford, UK, September 22-24, 2014. Proceedings*, volume 8762 of *Lecture Notes in Computer Science*, pages 45–57. Springer, 2014. doi:10.1007/978-3-319-11439-2_4.

19 Luca Breveglieri, Alessandra Cherubini, Claudio Citrini, and Stefano Crespi-Reghizzi. Multi-push-down languages and grammars. *Int. J. Found. Comput. Sci.*, 7(3):253–292, 1996. doi:10.1142/S0129054196000191.

20 Lorenzo Clemente, Paweł Parys, Sylvain Salvati, and Igor Walukiewicz. The diagonal problem for higher-order recursion schemes is decidable. In *Proceedings of the 31st Annual ACM/IEEE Symposium on Logic in Computer Science, LICS '16, New York, NY, USA, July 5-8, 2016*, pages 96–105. ACM, 2016. doi:10.1145/2933575.2934527.

21 Katherine E. Coons, Madan Musuvathi, and Kathryn S. McKinley. Bounded partial-order reduction. In Antony L. Hosking, Patrick Th. Eugster, and Cristina V. Lopes, editors, *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2013, part of SPLASH 2013, Indianapolis, IN, USA, October 26-31, 2013*, pages 833–848. ACM, 2013. doi:10.1145/2509136.2509556.

22 Aiswarya Cyriac. *Verification of communicating recursive programs via split-width. (Vérification de programmes récursifs et communicants via split-width)*. PhD thesis, École normale supérieure de Cachan, France, 2014. URL: https://tel.archives-ouvertes.fr/tel-01015561.

23 Aiswarya Cyriac, Paul Gastin, and K. Narayan Kumar. MSO decidability of multi-pushdown systems via split-width. In Maciej Koutny and Irek Ulidowski, editors, *CONCUR 2012 - Concurrency Theory - 23rd International Conference, CONCUR 2012, Newcastle upon Tyne, UK, September 4-7, 2012. Proceedings*, volume 7454 of *Lecture Notes in Computer Science*, pages 547–561. Springer, 2012. doi:10.1007/978-3-642-32940-1_38.

24 Werner Damm. The IO-and OI-hierarchies. *Theoretical Computer Science*, 20(2):95–207, 1982.

25 Werner Damm and Andreas Goerdt. An automata-theoretical characterization of the OI-hierarchy. *Information and Control*, 71(1):1–32, 1986.

26 Michael Emmi, Shaz Qadeer, and Zvonimir Rakamaric. Delay-bounded scheduling. In Thomas Ball and Mooly Sagiv, editors, *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26-28, 2011*, pages 411–422. ACM, 2011. doi:10.1145/1926385.1926432.

27 Bernd Fischer, Omar Inverso, and Gennaro Parlato. Cseq: A sequentialization tool for C - (competition contribution). In Nir Piterman and Scott A. Smolka, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 19th International Conference, TACAS 2013*, volume 7795 of *Lecture Notes in Computer Science*, pages 616–618. Springer, 2013. doi:10.1007/978-3-642-36742-7_46.

28 Pierre Ganty and Rupak Majumdar. Algorithmic verification of asynchronous programs. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 34(1):6, 2012.

29 Sheila A. Greibach. Remarks on blind and partially blind one-way multicounter machines. *Theoretical Computer Science*, 7(3):311–324, 1978. doi:10.1016/0304-3975(78)90020-8.

30 Matthew Hague, Jonathan Kochems, and C.-H. Luke Ong. Unboundedness and downward closures of higher-order pushdown automata. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*, pages 151–163. ACM, 2016. doi:10.1145/2837614.2837627.

31 Matthew Hague, Andrzej S. Murawski, C.-H. Luke Ong, and Olivier Serre. Collapsible pushdown automata and recursion schemes. In *Proceedings of the Twenty-Third Annual IEEE Symposium on Logic in Computer Science, LICS 2008, 24-27 June 2008, Pittsburgh, PA, USA*, pages 452–461, 2008. doi:10.1109/LICS.2008.34.

32 Leonard H Haines. On free monoids partially ordered by embedding. *Journal of Combinatorial Theory*, 6(1):94–98, 1969.

33 John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to automata theory, languages, and computation, 3rd Edition*. Pearson international edition. Addison-Wesley, 2007.

34 Omar Inverso, Ermenegildo Tomasco, Bernd Fischer, Salvatore La Torre, and Gennaro Parlato. Bounded verification of multi-threaded programs via lazy sequentialization. *ACM Trans. Program. Lang. Syst.*, 44(1):1:1–1:50, 2022. doi:10.1145/3478536.

35 Matthias Jantzen. On the hierarchy of Petri net languages. *RAIRO - Theoretical Informatics and Applications - Informatique Théorique et Applications*, 13(1):19–30, 1979. URL: http://www.numdam.org/item?id=ITA_1979__13_1_19_0.

36 Salvatore La Torre, P. Madhusudan, and Gennaro Parlato. Reducing context-bounded concurrent reachability to sequential reachability. In Ahmed Bouajjani and Oded Maler, editors, *Computer Aided Verification, 21st International Conference, CAV 2009, Grenoble, France, June 26 - July 2, 2009. Proceedings*, volume 5643 of *Lecture Notes in Computer Science*, pages 477–492. Springer, 2009. doi:10.1007/978-3-642-02658-4_36.

37 Akash Lal and Thomas W. Reps. Reducing concurrent analysis under a context bound to sequential analysis. *Formal Methods Syst. Des.*, 35(1):73–97, 2009. doi:10.1007/s10703-009-0078-9.

38 Akash Lal, Tayssir Touili, Nicholas Kidd, and Thomas W. Reps. Interprocedural analysis of concurrent programs under a context bound. In C. R. Ramakrishnan and Jakob Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008*, volume 4963 of *Lecture Notes in Computer Science*, pages 282–298. Springer, 2008. doi:10.1007/978-3-540-78800-3_20.

39 P. Madhusudan and Gennaro Parlato. The tree width of auxiliary storage. In Thomas Ball and Mooly Sagiv, editors, *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26-28, 2011*, pages 283–294. ACM, 2011. doi:10.1145/1926385.1926419.

40 Rupak Majumdar, Ramanathan S. Thinniyam, and Georg Zetzsche. General decidability results for asynchronous shared-memory programs: Higher-order and beyond. *Log. Methods Comput. Sci.*, 18(4), 2022. doi:10.46298/lmcs-18(4:2)2022.

41 Iason Marmanis, Michalis Kokologiannakis, and Viktor Vafeiadis. Reconciling preemption bounding with DPOR. In Sriram Sankaranarayanan and Natasha Sharygina, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 29th International Conference, TACAS 2023, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Paris, France, April 22-27, 2023, Proceedings, Part I*, volume

13993 of *Lecture Notes in Computer Science*, pages 85–104. Springer, 2023. doi:10.1007/978-3-031-30823-9_5.

42 AN Maslov. The hierarchy of indexed languages of an arbitrary level. *Doklady Akademii Nauk*, 217(5):1013–1016, 1974.

43 Roland Meyer, Sebastian Muskalla, and Georg Zetzsche. Bounded context switching for valence systems. In Sven Schewe and Lijun Zhang, editors, *29th International Conference on Concurrency Theory, CONCUR 2018, September 4-7, 2018, Beijing, China*, volume 118 of *LIPIcs*, pages 12:1–12:18. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2018. doi:10.4230/LIPIcs.CONCUR.2018.12.

44 Madanlal Musuvathi and Shaz Qadeer. Iterative context bounding for systematic testing of multithreaded programs. In *Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation, PLDI 2007, San Diego, CA, USA, June 10-13, 2007*, pages 446–455. ACM, 2007. doi:10.1145/1250734.1250785.

45 Truc L. Nguyen, Bernd Fischer, Salvatore La Torre, and Gennaro Parlato. Concurrent program verification with lazy sequentialization and interval analysis. In Amr El Abbadi and Benoît Garbinato, editors, *Networked Systems - 5th International Conference, NETYS 2017, Marrakech, Morocco, May 17-19, 2017, Proceedings*, volume 10299 of *Lecture Notes in Computer Science*, pages 255–271, 2017. doi:10.1007/978-3-319-59647-1_20.

46 Shaz Qadeer and Jakob Rehof. Context-bounded model checking of concurrent software. In Nicolas Halbwachs and Lenore D. Zuck, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 11th International Conference, TACAS 2005, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2005, Edinburgh, UK, April 4-8, 2005, Proceedings*, volume 3440 of *Lecture Notes in Computer Science*, pages 93–107. Springer, 2005. doi:10.1007/978-3-540-31980-1_7.

47 Koushik Sen and Mahesh Viswanathan. Model checking multithreaded programs with asynchronous atomic methods. In *CAV '06: Proc. 18th Int. Conf. on Computer Aided Verification*, volume 4144 of *LNCS*, pages 300–314. Springer, 2006.

48 Aneesh K. Shetty, Shankara Narayanan Krishna, and Georg Zetzsche. Scope-bounded reachability in valence systems. In Serge Haddad and Daniele Varacca, editors, *32nd International Conference on Concurrency Theory, CONCUR 2021, August 24-27, 2021, Virtual Conference*, volume 203 of *LIPIcs*, pages 29:1–29:19. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021. doi:10.4230/LIPIcs.CONCUR.2021.29.

49 Salvatore La Torre, P. Madhusudan, and Gennaro Parlato. Context-bounded analysis of concurrent queue systems. In C. R. Ramakrishnan and Jakob Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*, volume 4963 of *Lecture Notes in Computer Science*, pages 299–314. Springer, 2008. doi:10.1007/978-3-540-78800-3_21.

50 Salvatore La Torre, Parthasarathy Madhusudan, and Gennaro Parlato. A robust class of context-sensitive languages. In *22nd IEEE Symposium on Logic in Computer Science (LICS 2007), 10-12 July 2007, Wroclaw, Poland, Proceedings*, pages 161–170. IEEE Computer Society, 2007. doi:10.1109/LICS.2007.9.

51 Salvatore La Torre, Margherita Napoli, and Gennaro Parlato. Scope-bounded pushdown languages. *Int. J. Found. Comput. Sci.*, 27(2):215–234, 2016. doi:10.1142/S0129054116400074.

52 Salvatore La Torre, Margherita Napoli, and Gennaro Parlato. Reachability of scope-bounded multistack pushdown systems. *Inf. Comput.*, 275:104588, 2020. doi:10.1016/j.ic.2020.104588.

53 Georg Zetzsche. An approach to computing downward closures. In *ICALP 2015*, volume 9135, pages 440–451. Springer, 2015. Full version: http://arxiv.org/abs/1503.01068.