Context-Bounded Verification of Liveness Properties for Multithreaded Shared-Memory Programs

PASCAL BAUMANN, 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

We study context-bounded verification of liveness properties of multi-threaded, shared-memory programs, where each thread can spawn additional threads. Our main result shows that context-bounded fair termination is decidable for the model; context-bounded implies that each spawned thread can be context switched a fixed constant number of times. Our proof is technical, since fair termination requires reasoning about the composition of unboundedly many threads each with unboundedly large stacks. In fact, techniques for related problems, which depend crucially on replacing the pushdown threads with finite-state threads, are not applicable. Instead, we introduce an extension of vector addition systems with states (VASS), called VASS with balloons (VASSB), as an intermediate model; it is an infinite-state model of independent interest. A VASSB allows tokens that are themselves markings (balloons). We show that context bounded fair termination reduces to fair termination for VASSB. We show the latter problem is decidable by showing a series of reductions: from fair termination to configuration reachability for VASSB and thence to the reachability problem for VASS. For a lower bound, fair termination is known to be non-elementary already in the special case where threads run to completion (no context switches).

We also show that the simpler problem of context-bounded termination is 2EXPSPACE-complete, matching the complexity bound—and indeed the techniques—for safety verification. Additionally, we show the related problem of *fair starvation*, which checks if some thread can be starved along a fair run, is also decidable in the context-bounded case. The decidability employs an intricate reduction from fair starvation to fair termination. Like fair termination, this problem is also non-elementary.

CCS Concepts: • **Theory of computation → Concurrency**; • **Software and its engineering → Software verification**.

Additional Key Words and Phrases: verification, liveness, multithreaded programs, decidability, computational complexity

ACM Reference Format:
Pascal Baumann, Rupak Majumdar, Ramanathan S. Thinniyam, and Georg Zetzsche. 2021. Context-Bounded Verification of Liveness Properties for Multithreaded Shared-Memory Programs. *Proc. ACM Program. Lang.* 5, POPL, Article 44 (January 2021), 31 pages. https://doi.org/10.1145/3434325

Authors' addresses: Pascal Baumann, Max Planck Institute for Software Systems (MPI-SWS), Paul-Ehrlich-Straße, Building G26, Kaiserslautern, 67663, Germany, pbaumann@mpi-sws.org; Rupak Majumdar, Max Planck Institute for Software Systems (MPI-SWS), Paul-Ehrlich-Straße, Building G26, Kaiserslautern, 67663, Germany, rupak@mpi-sws.org; Ramanathan S. Thinniyam, Max Planck Institute for Software Systems (MPI-SWS), Paul-Ehrlich-Straße, Building G26, Kaiserslautern, 67663, Germany, thinniyam@mpi-sws.org; Georg Zetzsche, Max Planck Institute for Software Systems (MPI-SWS), Paul-Ehrlich-Straße, Building G26, Kaiserslautern, 67663, Germany, georg@mpi-sws.org.

2475-1421/2021/1-ART44
https://doi.org/10.1145/3434325

1 INTRODUCTION

We study decision problems related to liveness verification of shared-memory multithreaded programs. In a shared-memory multithreaded program, a number of *threads* execute concurrently. Each thread executes possibly recursive sequential code, and can spawn new threads for concurrent execution. The threads communicate through shared global variables that they can read and write. The execution of the program is guided by a non-deterministic *scheduler* that picks one of the spawned threads to execute in each time step. If the scheduler replaces the currently executing thread with a different one, we say the current active thread is *context switched*.

Shared-memory multithreaded programming is ubiquitous and static verification of safety or liveness properties of such programs is a cornerstone of formal verification research. Indeed, there is a vast research literature on the problem—from a foundational understanding of the computability and complexity of (subclasses of) models, to program logics, and to efficient tools for analysis of real systems.

In this paper, we focus on *decidability* issues for *liveness* verification for multithreaded shared memory programs with the ability to spawn threads. Liveness properties, intuitively, specify that "something good" happens when a program executes. A simple example of a liveness property is *termination*: the property that a program eventually terminates. In fact, termination is a "canonical" liveness property: for a very general class of liveness properties, through monitor constructions, verifying liveness properties reduces to verifying termination [Apt and Olderog 1991; Vardi 1991].

Unfortunately, under the usual notion of non-deterministic schedulers, some programs may fail to terminate for uninteresting reasons. Consider the following program:

```
1 global bit := 1;
2 main() { spawn foo; spawn bar; }
3 foo() { if bit = 1 then spawn foo; }
4 bar() { bit := 0; }
```

A main thread spawns two additional threads `foo` and `bar`. The thread `foo` checks if a global bit is set and, if so, re-spawns itself. The thread `bar` resets the global bit. There is a non-terminating execution of this program in which `bar` is never scheduled. However, a scheduler that never schedules a thread that is ready to run would be considered unfair. Instead, one formulates the problem of *fair termination*: termination under a *fair* non-deterministic scheduler. We abstract away from the exact mechanism of the scheduler, and only require that every spawned thread that is infinitely often ready to run is eventually scheduled. Then, every fair run of the above program is terminating: eventually `bar` is scheduled, after which `foo` does not spawn a new thread.

Fair termination of concurrent programs is highly undecidable. A celebrated result by Harel [1986] shows that fair termination is Π^1_1-complete; in fact, the problem is already Π^1_1-complete when the global state is finite and there are a finite number of recursive threads.[1] In contrast, safety verification, modeled as state reachability, is merely Σ^0_1-complete.

Since the high undecidability relies on an unbounded exchange of information among threads, a recent and apposite approach to verifying concurrent recursive programs is to explore only a representative subset of program behaviors by limiting the number of inter-thread interactions [Musuvathi and Qadeer 2007; Qadeer and Rehof 2005]. This approach, called context bounding by Qadeer and Rehof [2005], considers the verification problem as a family of problems, one for each K. The K-context bounded instance, for any fixed $K \geq 0$, considers only those executions where each thread is context switched at most K times by the scheduler. In the limit as $K \to \infty$, the K-bounded approach explores all behaviors where each thread runs a finite number of times.

[1] Recall that the class Π^1_1 in the analytic hierarchy is the class of all relations on $\mathbb{N}$ that can be defined by a universal *second-order* number-theoretic formula.

In practice, bounded explorations with small values of K have proved to be effective to uncover many safety and liveness bugs in real systems.

In this paper, we prove the following results. We first show that K-context bounded termination for multithreaded recursive programs with spawns is decidable and 2EXPSPACE-complete when $K \geq 1$. Then, we show that K-context bounded fair termination is decidable but non-elementary. Our result implies fair termination is Π_1^0-complete when each thread is context-switched a finite number of times. (Note that this does not contradict the Π_1^1-completeness of the general problem, in which a thread can be context switched infinitely often.) We also study a stronger notion of fairness called *fair non-starvation*, where threads are given unique identities in order to distinguish threads with the same local configuration, and show that fair non-starvation is also decidable.

Our results generalize the special case of $K = 0$ studied by Ganty and Majumdar [2012] as *asynchronous programs*. When $K = 0$, each thread executes to completion without being interrupted in the middle. Ganty and Majumdar show the decidability of safety and liveness verification for this model. In particular, they prove safety and termination are both EXPSPACE-complete and fair termination and fair starvation are decidable but non-elementary.[2] Their proof depends on the observation that, since threads are not interrupted, one can replace the pushdown automata for each thread by finite automata that accept Parikh-equivalent languages. Unfortunately, their technique does not generalize when context switches are allowed.

For $K \geq 1$, Atig et al. [2009] showed that the safety verification problem is decidable in 2EXPSPACE. Ten years later, a matching lower bound was shown by Baumann et al. [2020a]. The key observation in the decision procedure is that safety is preserved under downward closures: one can analyze a related program where some spawned threads are "forgotten." Since the downward closure of a context free language is effectively regular, one can replace the pushdown automaton for each thread by a finite automaton accepting the downward closure. In fact, our proof of termination also follows easily from this observation, as termination is also preserved by downward closures.

Unfortunately, fair termination and fair non-starvation are not preserved under downward closures. Thus, we cannot apply the preceding techniques to replace pushdown automata by finite automata in our construction. Thus, our proof is more intricate and requires several insights into the computational model.

The key difficulty in our decision procedure is to maintain a finite representation for *unboundedly* many active threads, each with *unboundedly* large local stacks and potentially spawning *unboundedly* many new threads, and to compose their context-switched executions into a global execution. In order to maintain and compose such configurations, we introduce a new model, called *VASS with balloons* (VASSB), that extends the usual model of a vector addition systems with states (VASS) with "balloons": a token in a VASSB can be a usual VASS token or a balloon token that is itself a vector. Intuitively, balloon tokens represent the possible new threads a thread can spawn along one of its execution segments.

We show through a series of constructions that the fair termination problem reduces to the fair termination problem for VASSB, and thence to the configuration reachability problem for VASSB. Finally, we show that configuration reachability for VASSB is decidable by a reduction to the reachability problem for VASS. This puts VASSB in the rare class of infinite-state systems which generalize VASS and yet maintain a decidable reachability (not just coverability!) problem.

[2]Their result shows a polynomial-time equivalence between fair termination and reachability in vector addition systems with states (VASS, a.k.a. Petri nets). The complexity bounds follow from our current knowledge of the complexity of VASS reachability [Czerwiński et al. 2019].

Finally, we show a reduction from the fair starvation problem to fair termination. The reduction relies on two combinatorial insights. The first is that if a program has an infinite fair run, then it has one in which there exists a bound on the number of threads spawned by each thread. The second is a novel pumping argument based on Ramsey's theorem; it implies that it suffices to track a finite amount of data about each thread to determine whether some thread can be starved.

In conclusion, we prove decidability of liveness verification for multithreaded shared memory programs with the ability to dynamically spawn threads, an extremely expressive model of multithreaded programming. This model sits at the boundary of decidability and subsumes many other models studied before.

For space reasons, the detailed proofs be found in the full version of the paper [Baumann et al. 2020b].

Related Work. Safety verification for concurrent recursive programs is already undecidable with just two threads and finite global store [Ramalingam 2000]. Many results on context-bounded safety verification consider a model with a *fixed* number of threads, without spawns. The complexity of safety verification for this model is well understood at this point. The key idea underlying the best algorithms reduce the problem to analyzing a sequential pushdown system [Lal and Reps 2009] by guessing the bounded sequences of context switches for each thread and using the finite state to ensure the sequential runs can be stitched together.

When the model allows *spawning* of new threads, as ours does, existing decision procedures are significantly more complex, both in their technicalities and in computational cost. There are relatively few results on decidability of liveness properties of infinite-state systems. Atig et al. [2012a] show a sufficient condition for fair termination for context-bounded executions of a fixed number of threads, where they look for ultimately periodic executions, in which each thread is context switched at most K times in the loop. They show that the search for such ultimately periodic executions can be reduced to safety verification. In our model, fair infinite runs may involve unboundedly many threads with unbounded stacks and need not be periodic—for example, there can always be more and more newly spawned threads.

Multi-pushdown systems model multithreaded programs with a fixed number of threads. Many decision procedures are known when the executions of such systems are restricted through different bounds such as context, scope, or phase [Atig et al. 2012b, 2017; Torre et al. 2016], and also through limitations on communication patterns [Lal et al. 2008]. These problems are orthogonal to us, either in the modeling capabilities or in the properties verified.

Decidability of linear temporal logic is known for weaker models of multithreaded recursive programs, such as symmetric parameterized programs [Kahlon 2008] or leader-follower programs with non-atomic reads and writes [Durand-Gasselin et al. 2017; Fortin et al. 2017; Muscholl et al. 2017]. These programs cannot perform compare-and-swap operations, and therefore, their computational power is quite limited (in fact, LTL model checking is PSPACE-complete). A number of heuristic approaches to fair termination of multithreaded programs provide sound but incomplete algorithms, but for a more general class of programs involving infinite-state data variables [Cook et al. 2007, 2011; Farzan et al. 2016; Kragl et al. 2020; Padon et al. 2018]. The goal there is to provide a sound proof rule for verification but not to prove a decidability result.

In terms of fair termination problems for VASS, the theme of computational hardness continues. For example, the classical notion of *fair runs*, in which an infinitely activated transition has to be fired infinitely often, leads to undecidability [Carstensen 1987] and even Σ^1_1-completeness [Howell et al. 1991]. However, *weakly fair* termination, where only those transitions that are almost always activated have to be fired infinitely often, is decidable [Jančar 1990]. A rich taxonomy of fairness

notions with corresponding decidability results can be found in [Howell et al. 1991]. However, all of these notions appear to be incomparable with our fairness notion for VASSB.

Our model of VASSB treads the boundary of models that generalize VASS for which reachability can be proved to be decidable. We note that there are several closely related models, VASS with a stack [Leroux et al. 2015] and branching VASS [Verma and Goubault-Larrecq 2005], for which decidability of reachability is a long-standing open problem, and others, nested Petri nets [Lomazova and Schnoebelen 1999], for which reachability is undecidable.

2 DYNAMIC NETWORKS OF CONCURRENT PUSHDOWN SYSTEMS (DCPS)

2.1 Preliminary Definitions

Multisets. A *multiset* $\mathbf{m}\colon S \to \mathbb{N}$ over a set S maps each element of S to a natural number. Let $\mathbb{M}[S]$ be the set of all multisets over S. We treat sets as a special case of multisets where each element is mapped onto 0 or 1. We sometimes write $\mathbf{m} = [\![a_1, a_1, a_3]\!]$ for the multiset $\mathbf{m} \in \mathbb{M}[S]$ such that $\mathbf{m}(a_1) = 2$, $\mathbf{m}(a_3) = 1$, and $\mathbf{m}(a) = 0$ for each $a \in S \backslash \{a_1, a_3\}$. The empty multiset is denoted $\emptyset$. The size of a multiset $\mathbf{m}$, denoted $|\mathbf{m}|$, is given by $\sum_{a \in S} \mathbf{m}(a)$. This definition applies to sets as well.

Given two multisets $\mathbf{m}, \mathbf{m}' \in \mathbb{M}[S]$ we define $\mathbf{m} + \mathbf{m}' \in \mathbb{M}[S]$ to be a multiset such that for all $a \in S$, we have $(\mathbf{m} + \mathbf{m}')(a) = \mathbf{m}(a) + \mathbf{m}'(a)$. For $c \in \mathbb{N}$, we define $c\mathbf{m}$ as the multiset that maps each $a \in S$ to $c \cdot \mathbf{m}(a)$. We also define the natural order $\preceq$ on $\mathbb{M}[S]$ as follows: $\mathbf{m} \preceq \mathbf{m}'$ iff there exists $\mathbf{m}^\Delta \in \mathbb{M}[S]$ such that $\mathbf{m} + \mathbf{m}^\Delta = \mathbf{m}'$. We also define $\mathbf{m} - \mathbf{m}'$ for $\mathbf{m}' \preceq \mathbf{m}$ analogously: for all $a \in S$, we have $(\mathbf{m} - \mathbf{m}')(a) = \mathbf{m}(a) - \mathbf{m}'(a)$.

Pushdown Automata. A *pushdown automaton (*PDA*)* $\mathcal{P}_{(g,\gamma)} = (Q, \Sigma, \Gamma, E, q_0, \gamma_0, Q_F)$ consists of a finite set of *states* Q, a finite input alphabet Σ, a finite alphabet of *stack symbols* Γ, an *initial state* $q_0 \in Q$, an *initial stack symbol* $\gamma_0 \in \Gamma$, a set of *final states* $Q_F \subseteq Q$, and a transition relation $E \subseteq (Q \times \Gamma) \times \Sigma_\varepsilon \times (Q \times \Gamma^{\leq 2})$, where $\Sigma_\varepsilon = \Sigma \cup \{\varepsilon\}$ and $\Gamma^{\leq 2} = \{\varepsilon\} \cup \Gamma \cup \Gamma^2$. For $((q, \gamma), a, (q', w)) \in E$ we also write $q \xrightarrow{a|\gamma/w} q'$.

The set of *configurations* of $\mathcal{P}$ is $Q \times \Gamma^*$. The *initial configuration* is (q_0, γ_0). The set of *final configurations* is $Q_F \times \Gamma^*$. For each $a \in \Sigma \cup \{\varepsilon\}$, the relation $\overset{a}{\Longrightarrow}$ on configurations of $\mathcal{P}$ is defined as follows: $(q, \gamma w) \overset{a}{\Longrightarrow} (q', w'w)$ for all $w \in \Gamma^*$ iff (1) there is a transition $q \xrightarrow{a|\gamma/w'} q' \in E$, or (2) there is a transition $q \xrightarrow{a|\varepsilon} q' \in E$ and $\gamma = w' = \varepsilon$.

For two configurations c, c' of $\mathcal{P}$, we write $c \Rightarrow c'$ if $c \overset{a}{\Longrightarrow} c'$ for some a. Furthermore, we write $c \overset{u}{\Longrightarrow}{}^* c'$ for some $u \in \Sigma^*$ if there is a sequence of configurations c_0 to c_n with

$$c = c_0 \overset{a_1}{\Longrightarrow} c_1 \overset{a_1}{\Longrightarrow} c_2 \cdots c_{n-1} \overset{a_n}{\Longrightarrow} c_n = c',$$

such that $a_1 \ldots a_n = u$. We then call this sequence a *run* of $\mathcal{P}$ over u. We also write $c \Rightarrow^* c'$ if the word u does not matter. A run of $\mathcal{P}$ is *accepting* if c is initial and c' is final. The *language* accepted by $\mathcal{P}$, denoted $L(\mathcal{P})$ is the set of words $\in \Sigma^*$, over which there is an accepting run of $\mathcal{P}$.

Given two configurations c, c' of $\mathcal{P}$ with $c \Rightarrow^* c'$, we say that c' is *reachable* from c and that c is *backwards-reachable* from c'. If c is the initial configuration, we simply say that c' is reachable.

Parikh Images and Semi-linear Sets. The Parikh image of a word $u \in \Sigma^*$ is a function $\mathrm{Parikh}(u) : \Sigma \to \mathbb{N}$ such that, for every $a \in \Sigma$, we have $\mathrm{Parikh}(u)(a) = |u|_a$, where $|u|_a$ denotes the number of occurrences of a in u. We extend the definition to the Parikh image of a language $L \subseteq \Sigma^*$: $\mathrm{Parikh}(L) = \{\mathrm{Parikh}(u) \mid u \in L\}$. We associate the natural isomorphism between $\mathbb{N}^\Sigma$ and $\mathbb{N}^{|\Sigma|}$ and consider the functions as vectors of natural numbers.

A subset of $\mathbb{M}[S]$ is *linear* if it is of the form $\{\mathbf{m}_0 + t_1\mathbf{m}_1 + \ldots + t_n\mathbf{m}_n \mid t_1, \ldots, t_n \in \mathbb{N}\}$ for some multisets $\mathbf{m}_0, \mathbf{m}_1, \ldots, \mathbf{m}_n \in \mathbb{M}[S]$. We call $\mathbf{m}_0$ the *base vector* and $\mathbf{m}_1, \ldots, \mathbf{m}_n$ the *period vectors*. A linear set has a finite representation based on its base and period vectors. A *semi-linear* set is a finite union of linear sets.

Theorem 2.1 ([Parikh 1966]). *For any context-free language L, the set* Parikh(L) *is semi-linear. A representation of the semi-linear set* Parikh(L) *can be effectively constructed from a* PDA *for L.*

2.2 Dynamic Networks of Concurrent Pushdown Systems

A *Dynamic Network of Concurrent Pushdown Systems (*DCPS*)* $\mathcal{A} = (G, \Gamma, \Delta, g_0, \gamma_0)$ consists of a finite set of *(global) states* G, a finite alphabet of *stack symbols* Γ, an *initial state* $g_0 \in G$, an *initial stack symbol* $\gamma_0 \in \Gamma$, and a finite set of *transition rules* Δ. The set of transition rules Δ is partitioned into four kinds of rules: *creation rules* Δ_c, *interruption rules* Δ_i, *resumption rules* Δ_r, and *termination rules* Δ_t. Elements of Δ_c have one of two forms: (1) $g|\gamma \hookrightarrow g'|w'$, or (2) $g|\gamma \hookrightarrow g'|w' \triangleright \gamma'$, where $g, g' \in G$, $\gamma, \gamma' \in \Gamma$, $w' \in \Gamma^*$, and $|w'| \leq 2$. Rules of type (1) allow the DCPS to take a single step in one of its threads. Rules of type (2) additionally spawn a new thread with top of stack γ'. Elements of Δ_i have the form $g|\gamma \mapsto g'|w'$, where $g, g' \in G$, $\gamma \in \Gamma$, and $w' \in \Gamma^*$ with $1 \leq |w'| \leq 2$. Elements of Δ_r have the form $g \mapsto g' \triangleleft \gamma$, where $g, g' \in G$ and $\gamma \in \Gamma$. Elements of Δ_t have the form $g \mapsto g'$, where $g, g' \in G$.

The *size* $|\mathcal{A}|$ of $\mathcal{A}$ is defined as $|G| + |\Gamma| + |\Delta|$: the number of symbols needed to describe the global states, the stack alphabet, and the transition rules.

The set of configurations of $\mathcal{A}$ is $G \times \big((\Gamma^* \times \mathbb{N}) \cup \{\#\}\big) \times \mathbb{M}[\Gamma^* \times \mathbb{N}]$. Given a configuration $\langle g, (w, i), \mathbf{m}\rangle$, we call g the *(global) state*, (w, i) the *local configuration* of the *active thread*, and $\mathbf{m}$ the multiset of the *local configurations* of the *inactive threads*. In a configuration $\langle g, \#, \mathbf{m}\rangle$, we call $\#$ a *schedule point*.

The initial configuration of $\mathcal{A}$ is $\langle g_0, \#, [\![(\gamma_0, 0)]\!]\rangle$. For a configuration c of $\mathcal{A}$, we will sometimes write $c.g$ for the state of c and $c.\mathbf{m}$ for the multiset of threads of c (both active and inactive). The *size* of a configuration $c = \langle g, (w, i), \mathbf{m}\rangle$ is defined as $|c| = |w| + \sum_{(w',j)\in\mathbf{m}} |w'|$.

Intuitively, a DCPS represents a multi-threaded, shared memory program. The global states G represent the shared memory. Each thread is potentially recursive. It maintains its own stack w over the stack alphabet Γ and uses the transition rules in Δ_c to manipulate the global state and its stack. It can additionally spawn new threads using rules of type (2) in Δ_c. In a local configuration, the natural number i keeps track of how many times a thread has already been context switched by the underlying scheduler. Any newly spawned thread has its context switch number set to 0.

The steps of a single thread defines the following *thread step* relation $\to$ on configurations of $\mathcal{A}$: we have $\langle g, (\gamma w, i), \mathbf{m}\rangle \to \langle g', (w'w, i), \mathbf{m}'\rangle$ for all $w \in \Gamma^*$ iff (1) there is a rule $g|\gamma \hookrightarrow g'|w'$ in Δ_c and $\mathbf{m}' = \mathbf{m}$ or (2) there is a rule $g|\gamma \hookrightarrow g'|w' \triangleright \gamma'$ in Δ_c and $\mathbf{m}' = \mathbf{m} + [\![(\gamma', 0)]\!]$. We extend the *thread step* relation $\to^+$ to be the irreflexive-transitive closure of $\to$; thus $c \to^+ c'$ if there is a sequence $c \to c_1 \to \ldots c_k \to c'$ for some $k \geq 0$.

A non-deterministic scheduler switches between concurrent threads. The active thread is the one currently being executed and the multiset $\mathbf{m}$ keeps all other partially executed threads in the system. Any spawned thread is put in $\mathbf{m}$ for future execution with an initial context switch number 0. The scheduler may interrupt a thread based on the interruption rules and non-deterministically resume a thread based on the resumption rules.

The actions of the scheduler define the *scheduler step* relation $\mapsto$ on configurations of $\mathcal{A}$:

$$\textsc{Swap}\ \frac{g|\gamma \mapsto g'|w' \in \Delta_i}{\langle g, (\gamma w, i), \mathbf{m}\rangle \mapsto \langle g', \#, \mathbf{m} + [\![w'w, i+1]\!]\rangle} \qquad \textsc{Resume}\ \frac{g \mapsto g' \triangleleft \gamma \in \Delta_r}{\langle g, \#, \mathbf{m} + [\![\gamma w, i]\!]\rangle \mapsto \langle g', (\gamma w, i), \mathbf{m}\rangle} \qquad \textsc{Term}\ \frac{g \mapsto g' \in \Delta_t}{\langle g, (\varepsilon, i), \mathbf{m}\rangle \mapsto \langle g', \#, \mathbf{m}\rangle}$$

If a thread can be interrupted, then Swap swaps it out and increases the context switch number of the thread. The rule Resume picks a thread that is ready to run based on the current global state and its top of stack symbol and makes it active. The rule Term removes a thread on termination (empty stack).

A *run* of a DCPS is a finite or infinite sequence of alternating thread execution and scheduler step relations

$$c_0 \rightarrow^+ c_0' \mapsto c_1 \rightarrow^+ c_1' \mapsto \dots$$

such that c_0 is the initial configuration. The run is *K-context switch bounded* if, moreover, for each $j \geq 0$, the configuration $c_j = (g, (w, i), \mathbf{m})$ satisfies $i \leq K$. In a K-context switch bounded run, each thread is context switched at most K times and the scheduler never schedules a thread that has already been context switched $K + 1$ times. When the distinction between thread and scheduler steps is not important, we write a run as a sequence $c_0 \Rightarrow c_1 \dots$.

2.3 Identifiers and the Run of a Thread

Our definition of DCPS does not have thread identifiers associated with a thread. However, it is convenient to be able to identify the run of a single thread along the execution. This can be done by decorating local configurations with unique identifiers and modifying the thread step for $g|\gamma \hookrightarrow g'|w \triangleright \gamma'$ to add a thread $(\ell, \gamma', 0)$ to the multiset of inactive threads, where ℓ is a fresh identifier. By decorating any run with identifiers, we can freely talk about the run of a single thread, the multiset of threads spawned by a thread, etc.

Let us focus on the run of a specific thread, that starts executing from some global state g with an initial stack symbol γ. In the course of its run, the thread updates its own local stack and spawns new threads, but it also gets swapped out and swapped back in.

We show that the run of a thread corresponds to the run of an associated PDA that can be extracted from $\mathcal{A}$. This PDA updates the global state and the stack based on the rules in Δ_c, but additionally (1) makes visible as the input alphabet the initial symbols (from Γ) of the spawned threads, and (2) non-deterministically guesses jumps between global states corresponding to the effect of context switches. There are two kinds of jumps. A jump (g_1, γ, g_2) in the PDA corresponds to the thread being switched out leading to global state g_1 and later resuming at global state g_2 with γ on top of its stack (without being active in the interim). A jump $(g, \bot)$ corresponds to the last time the PDA is swapped out (leading to global state g). We also make these guessed jumps visible as part of the input alphabet. Thus, the input alphabet of the PDA is $\Gamma \cup G \times \Gamma \times G \cup G \times \{\bot\}$.

For any $g \in G$ and $\gamma \in \Gamma$, we define the PDA $\mathcal{P}_{(g,\gamma)} = (Q, \Sigma, \Gamma_\bot, E, \mathsf{init}, \bot, \{\mathsf{init}, \mathsf{end}\})$, where $Q = G \cup G \times \Gamma \cup \{\mathsf{init}, \mathsf{end}\}$, $\Gamma_\bot = \Gamma \cup \{\bot\}$, $\Sigma = \Gamma \cup G \times \Gamma \times G \cup G \times \{\bot\}$, E is the smallest transition relation such that

(1) There is a transition $\mathsf{init} \xrightarrow{\varepsilon|\bot/\gamma\bot} g$ in E,
(2) For every $g_1|\gamma_1 \hookrightarrow g_2|w \in \Delta_c$ there is a transition $g_1 \xrightarrow{\varepsilon|\gamma_1/w} g_2$ in E,
(3) For every $g_1|\gamma_1 \hookrightarrow g_2|w \triangleright \gamma_2 \in \Delta_c$ there is a transition $g_1 \xrightarrow{\gamma_2|\gamma_1/w} g_2$ in E,
(4) For every $g_1|\gamma_1 \hookrightarrow g_2|w \in \Delta_i$, $g_3 \in G$, and $\gamma_2 \in \Gamma$ there is a transition $g_1 \xrightarrow{(g_2,\gamma_2,g_3)|\gamma_1/w} (g_3, \gamma_2)$, and a transition $(g_3, \gamma_2) \xrightarrow{\varepsilon|\gamma_2/\gamma_2} g_3$ in E,
(5) For every $g_1|\gamma_1 \hookrightarrow g_2|w \in \Delta_i$ and every $\gamma_2 \in \Gamma_\bot$ there is a transition $g_1 \xrightarrow{\varepsilon|\gamma_1/w} (g_2, \bot)$, and a transition $(g_2, \bot) \xrightarrow{(g_2,\bot)|\gamma_2/\gamma_2}$ end in E,
(6) For every $g_1 \hookrightarrow g_2 \in \Delta_t$ there is a transition $g_1 \xrightarrow{(g_2,\bot)|\bot/\bot}$ end in E.

The set of behaviors of the PDA $\mathcal{P}_{(g,\gamma)}$ which correspond to a thread execution with precisely i ($i \leq K+1$) context switches is given by the following language:

$$L^{(i)}_{(g,\gamma)} = L(\mathcal{P}_{(g,\gamma)}) \cap ((\Gamma^* \cdot G \times \Gamma \times G)^{i-1}(\Gamma^* \cdot G \times \{\bot\}))$$

The language $L^{(i)}_{(g,\gamma)}$ is a context-free language. In the definition, we use the end of stack symbol $\bot$ to recognize when the stack is empty.

2.4 Decision Problems and Main Results

Previous Work: Safety. The *reachability problem* for DCPS asks, given a global state g of $\mathcal{A}$, if there is a run $c_0 \Rightarrow c_1 \ldots \Rightarrow c_n$ such that $c_n.g = g$. It is well-known that reachability is undecidable (e.g., one can reduce the emptiness problem for intersection of context free languages). Therefore, it is customary to consider *context-switch bounded* decision questions. Given $K \in \mathbb{N}$, a state g of $\mathcal{A}$ is K-context switch bounded reachable if there is a K-context switch bounded run $c_0 \Rightarrow \ldots \Rightarrow c_n$ with $c_n.g = g$. For a fixed K, the *K-bounded state reachability problem* (SRP$[K]$) for a DCPS is defined as follows:

Given A DCPS $\mathcal{A}$ and a global state g
Question Is g K-context switch bounded reachable in $\mathcal{A}$?

This problem is known to be decidable; the 2EXPSPACE upper bound for each K was proved by Atig et al. [2009] and a matching lower bound for $K \geq 1$ by Baumann et al. [2020a]. In case $K = 0$, the problem is known to be EXPSPACE-complete [Ganty and Majumdar 2012].

This Paper: Liveness. We now turn to context-bounded liveness specifications. The simplest liveness specification is *(non-)termination*: does a program halt? For a fixed $K \in \mathbb{N}$, the *K-bounded non-termination problem* NTERM$[K]$ is defined as follows:

Given A DCPS $\mathcal{A}$.
Question Is there an infinite K-context switch bounded run?

When $K = 0$, the non-termination problem is known to be EXPSPACE-complete [Ganty and Majumdar 2012]. We show the following result.

Theorem 2.2 (Termination). *For each $K \geq 1$, the problem* NTERM$[K]$ *is* 2EXPSPACE*-complete.*

Fairness. An infinite run is *fair* if, intuitively, any thread that can be executed is eventually executed by the scheduler. Fairness is used as a way to rule out non-termination due to uninteresting scheduler choices.

We say a thread $t = (\gamma w, i)$ is *ready* at a configuration $c = (g, \#, \mathbf{m})$ if $t \in \mathbf{m}$ and there is some rule $g \mapsto g' \lhd \gamma$ in Δ_r. A thread t is *scheduled* at c if the scheduler step makes t the active thread. A run is unfair to thread t if it is ready infinitely often but never scheduled. A *fair* run ρ is one which is not unfair to any thread. Restricting our attention to K-context switch bounded runs gives us the corresponding notion of fair K-context switch bounded runs.

For fixed $K \in \mathbb{N}$, the *K-context bounded fair non-termination problem* FNTERM$[K]$ asks:

Given A DCPS $\mathcal{A}$.
Question Is there an infinite, fair K-context switch bounded run?

Note that since our model does not have individual thread identifiers, fairness is defined only over equivalence classes of threads that have the same stack w and the same context switch number i. The reason for our taking into account stacks and context switch numbers is the following. It is a simple observation that there exists an infinite fair run in our sense if and only if there exists a run in the corresponding system *with thread identifiers*–that is fair to each individual thread. This is because an angelic scheduler could always pick the earliest spawned thread among those

with the same stack and context switch number. Therefore, our results allow us to reason about multi-threaded systems with identifiers.

This raises the question of whether there are runs that are fair in our sense, but where a non-angelic scheduler would still yield unfairness for some thread identity. In other words, is it possible that a fair run *starves* a specific thread. For example, consider a program in which the main thread spawns two copies of a thread `foo`. Each thread `foo`, when scheduled, simply spawns another copy of `foo` and terminates. Here is a fair run of the program (we omit the global state as it is not relevant), where we have decorated the threads with identifiers:

$$(\#, [\![(\texttt{main},0)^0]\!]) \overset{*}{\Rightarrow} ((\texttt{main},0)^0, [\![]\!]) \overset{*}{\Rightarrow} (\#, [\![(\texttt{foo},0)^1, (\texttt{foo},0)^2]\!]) \overset{*}{\Rightarrow} ((\texttt{foo},0)^2, [\![(\texttt{foo},0)^1]\!]) \overset{*}{\Rightarrow}$$

$$(\#, [\![(\texttt{foo},0)^1, (\texttt{foo},0)^3]\!]) \overset{*}{\Rightarrow} ((\texttt{foo},0)^3, [\![(\texttt{foo},0)^1]\!]) \overset{*}{\Rightarrow} \dots$$

The run is fair, but a specific thread marked with identifier 1 is never picked.

Formally, a thread $t = (w, i)$ is *starved* in an infinite fair run $\rho = c_0 \Rightarrow c_1 \Rightarrow \dots$ iff there is some j such that $c_i.\mathbf{m}(t) \geq 1$ for all $i \geq j$ and whenever t is resumed at c_k for $k \geq j$, we have $c.\mathbf{m}(t) \geq 2$.

For fixed $K \in \mathbb{N}$, the *K-bounded fair starvation problem* $\mathsf{STARV}[K]$ is defined as follows:

Given A DCPS $\mathcal{A}$.

Question Is there an infinite, fair K-context switch bounded run that starves some thread?

We show the following results.

Theorem 2.3 (Fair Non-Termination). *For each $K \in \mathbb{N}$, the problem $\mathsf{FNTERM}[K]$ is decidable.*

Theorem 2.4 (Fair Starvation). *For each $K \in \mathbb{N}$, the problem $\mathsf{STARV}[K]$ is decidable.*

Previously, decidability results were only known when $K = 0$ [Ganty and Majumdar 2012]. Recall that a decision problem is *nonelementary* if it is not in $\bigcup_{k \geq 0} k$-EXPTIME. Our algorithms are nonelementary: they involve an (elementary) reduction to the reachability problem for vector addition systems with states (VASS). This is unavoidable: already for $K = 0$, the fair non-termination and fair starvation problems are non-elementary, because there is a reduction from the reachability problem for VASS [Ganty and Majumdar 2012], which is non-elementary [Czerwiński et al. 2019].

In the rest of the paper, we prove Theorems 2.2, 2.3, and 2.4.

3 WARM-UP: NON-TERMINATION

In this section, we prove Theorem 2.2. The theorem follows easily from previous results for safety verification [Atig et al. 2011; Baumann et al. 2020a]. We recall the main ideas as a step toward the more complex proof for *fair* termination.

3.1 Downward Closures: From DCPS to DCFS

A DCPS $\mathcal{A} = (G, \Gamma, \Delta, g_0, \gamma_0)$ is called a dynamic network of concurrent *finite* systems (DCFS) if in each transition rule in $\Delta_c \cup \Delta_i$, we have $|w'| \leq 1$. Intuitively, a DCFS corresponds to the special case where each thread is a finite-state process (and each stack is bounded by 1).

We reduce the K-bounded non-termination problem for DCPS to the non-termination problem for DCFS. Fix $K \in \mathbb{N}$ and a DCPS $\mathcal{A} = (G, \Gamma, \Delta, g_0, \gamma_0)$. The crucial observation of Atig et al. [2011] is that answer to the K-bounded reachability problem remains unchanged if we allow threads to "drop" some spawned threads. That is, for every $g|\gamma \hookrightarrow g'|w' \triangleright \gamma'$, we also add the rule $g|\gamma \hookrightarrow g'|w'$ to Δ_c. Informally, the "forgotten" spawned thread γ' is never scheduled. Clearly, a global state is reachable in the original DCPS iff it is reachable in the new DCPS.

We observe that this transformation also preserves non-termination: if there is a (K-bounded) non-terminating run in the original DCPS, there is one in the new one.

The ability to forget spawned tasks allows us to transform the language $L^{(i)}_{(g,\gamma)}$ of each thread into a *regular* language by taking *downward closures*.

We need some definitions. For any alphabet Σ, define the subword relation $\sqsubseteq \subseteq \Sigma^* \times \Sigma^*$ as follows: for every $u, v \in \Sigma^*$, we have $u \sqsubseteq v$ iff u can be obtained from v by deleting some letters from v. For example, *acbba* $\sqsubseteq$ *bacbacbac* but *abba* $\not\sqsubseteq$ *baba*. The *downward closure* $w{\downarrow}$ with respect to the subword order of a word $w \in \Sigma^*$ is defined as $w{\downarrow} := \{w' \in \Sigma^* \mid w' \sqsubseteq w\}$. The downward closure $L{\downarrow}$ of a language $L \subseteq \Sigma^*$ is given by $L{\downarrow} := \{w' \in \Sigma^* \mid \exists w \in L \colon w' \sqsubseteq w\}$. Recall that the downward closure $L{\downarrow}$ of any language L is a regular language [Haines 1969]. Moreover, a finite automaton accepting the downward closure of a context-free language can be effectively constructed [Courcelle 1991]. The size of the resulting automaton is at most exponential in the size of the PDA for the context-free language [Bachmeier et al. 2015].

Now consider the following language:

$$\hat{L}_{(g,\gamma)} = \bigcup_{i=1}^{K+1} \left(L^{(i)}_{(g,\gamma)}{\downarrow} \cap \left(\left(\Gamma^* \cdot (G \times \Gamma \times G)\right)^{i-1} (\Gamma^* \cdot G \times \{\bot\}) \right) \right)$$

This language is regular and can be effectively constructed from the PDA $\mathcal{P}_{(g,\gamma)}$. It accepts all behaviors of a thread that is context switched at most $K+1$ times such that, by adding additional spawned tasks, one gets back a run of the original thread in $\mathcal{A}$.

The DCFS simulates the downward closure of the DCPS by simulating the composition of the automata for each downward closure. The construction is identical to [Atig et al. 2011, Lemma 5.3]. Thus, we can conclude with the following lemma.

Lemma 3.1. *The K-bounded non-termination problem for* DCPS *can be reduced in exponential time to the non-termination problem for* DCFS. *The resulting* DCFS *is of size at most exponential in the size of the* DCPS.

3.2 From DCFS Non-Termination to VASS Non-Termination

A *vector addition system with states* (VASS) is a tuple $V = (Q, P, E)$ where Q is a finite set of *states*, P is a finite set of *places*, and E is a finite set of edges of the form $q \xrightarrow{\delta} q'$ where $\delta \in \mathbb{Z}^P$. A *configuration* of the VASS is a pair $(q, u) \in Q \times \mathbb{M}[P]$. The edges in E induce a transition relation on configurations: there is a transition $(q, u) \xrightarrow{\delta} (q', u')$ if there is an edge $q \xrightarrow{\delta} q'$ in E such that $u'(p) = u(p) + \delta(p)$ for all $p \in P$. A *run* of the VASS is a finite or infinite sequence of configurations $c_0 \xrightarrow{\delta_0} c_1 \xrightarrow{\delta_1} \ldots$. The *non-termination* problem for VASS asks, given a VASS and an initial configuration c_0, is there an infinite run starting from c_0.

Lemma 3.2. *The K-context bounded non-termination problem for* DCFS *can be reduced in polynomial time to non-termination problem for* VASS.

Proof. Let $\mathcal{A} = (G, \Gamma, \delta, g_0, \gamma_0)$ be a DCFS. We define a VASS $V(\mathcal{A}) = (G \times (\Gamma \times \{0, \ldots, K\} \cup \{\#\}), (\Gamma \cup \{\varepsilon\}) \times \{0, \ldots, K+1\}, E)$. Intuitively, a configuration $((g, \gamma, i), u)$ of the VASS represents a configuration of the DCFS where the global state is g, the active thread has stack γ and has been previously context switched i times, and for each $\gamma' \in \Gamma$ and $i \in \{0, \ldots, K+1\}$, the value $u(\gamma', i)$ represents the number of pending threads with stack γ' which have each been context switched i times. A global state $(g, \#)$ indicates a state where the scheduler picks a new thread. The edges in E update the configurations to simulate the steps of the DCFS.

For each transition $g|\gamma \hookrightarrow g'|\gamma' \in \Delta_c$ and for each $i \in \{0, \ldots, K\}$, the VASS has a transition that changes (g, γ, i) to (g', γ', i). For each transition $g|\gamma \hookrightarrow g'|\gamma' \triangleright \gamma'' \in \Delta_c$ and for each $i \in \{0, \ldots, K\}$, the VASS has a transition that changes (g, γ, i) to (g', γ', i) and puts a token in $(\gamma'', 0)$. For each

transition $g|\gamma \mapsto g'|\gamma' \in \Delta_{\mathrm{i}}$ and for each $i \in \{0, \ldots, K\}$, the VASS has a transition that changes g to $(g', \#)$ while putting a token into $(\gamma', i+1)$. For each $g \mapsto g' \lhd \gamma \in \Delta_{\mathrm{r}}$, there is a transition $(g, \#)$ to (g', γ, i) that takes a token from (γ, i). For each $g \mapsto g' \in \Delta_{\mathrm{t}}$, there is a transition (g, ε, i) to $(g', \#)$.

Clearly, there is a bijection between the runs of $\mathcal{A}$ and the runs of the VASS from $((g_0, \#), [[\gamma_0, 0]])$. Thus, there is an infinite run in $\mathcal{A}$ iff there is an infinite run in $V(\mathcal{A})$ from $((g_0, \#), [[\gamma_0, 0]])$. ∎

3.3 Proof of Theorem 2.2

The 2EXPSPACE upper bound follows by combining Lemmas 3.1, 3.2, and the EXPSPACE upper bound for the non-termination problem for VASS [Rackoff 1978].

The 2EXPSPACE lower bound follows from the observation made already in [Baumann et al. 2020a] that the 2EXPSPACE-hardness of K-bounded reachability already holds for *terminating* DCPS, in which every run is terminating. It is now a simple reduction to take an instance of the K-bounded state reachability problem for a terminating DCPS and add a "gadget" that produces an infinite run whenever the target global state is reached.

4 FAIR NON-TERMINATION

We now turn to proving Theorem 2.3. Unfortunately, fair termination is not preserved under downward closure. The example in Section 1 has no fair infinite run, since eventually (under fairness), `bit` is set to 1 by the instance of `bar` and the program terminates. However, the downward closure that omits `bar` has a fair infinite run. Thus, we cannot replace the PDAs for each thread with finite-state automata and there is no obvious reduction to VASS.

Our proof is more complicated. First, we introduce an extension, VASS with *balloons* (VASSB), of VASS (Section 4.1). A VASSB extends a VASS with balloon states and balloon places, and allows keeping multisets of state-vector pairs over balloons. We can use this additional power to store spawned threads. As we shall see (Section 4.2), we can reduce DCPS to VASSB. Later, we shall show decidability of fair infinite behaviors for VASSB, completing the proof.

4.1 VASS with Balloons

A *VASS with balloons* (VASSB) is a tuple $\mathcal{V} = (Q, P, \Omega, \Phi, E)$, where Q is a finite set of *states*, P is a finite set of *places*, Ω is a finite set of *balloon states*, Φ is a finite set of *balloon places*, and E is a finite set of edges of the form $q \xrightarrow{op} q'$, where op is one of a finite set OP of operations of the following form:

(1) $op = \delta$ with $\delta \in \mathbb{Z}^P$,
(2) $op = \mathrm{inflate}(\sigma, S)$, where $\sigma \in \Omega$ and $S \subseteq \mathbb{N}^\Phi$ is a semi-linear subset of $\mathbb{N}^\Phi$.
(3) $op = \mathrm{deflate}(\sigma, \sigma', \pi, p)$, where $\sigma, \sigma' \in \Omega$, $\pi \in \Phi$, $p \in P$.
(4) $op = \mathrm{burst}(\sigma)$, where $\sigma \in \Omega$.

A *configuration* of a VASSB is an element of $Q \times \mathbb{M}[P] \times \mathbb{M}[\Omega \times \mathbb{M}[\Phi]]$. That is, a configuration $c = (q, \mathbf{m}, \mathbf{n})$ consists of a state $q \in Q$, a multiset $\mathbf{m} \in \mathbb{M}[P]$, and a multiset $\mathbf{n} \in \mathbb{M}[\Omega \times \mathbb{M}[\Phi]]$ of *balloons*. We assume $\mathbf{n}$ has finite support. A *semiconfiguration* is a configuration $(q, \mathbf{m}, \emptyset)$. For semiconfigurations, we simply write $(q, \mathbf{m}) \in Q \times \mathbb{M}[P]$. For a configuration c, we write $c.q$, $c.\mathbf{m}$, and $c.\mathbf{n}$ to denote the components of c. For a balloon $b \in \Omega \times \mathbb{M}[\Phi]$, we write $b.\sigma$ and $b.\mathbf{k}$ to indicate its balloon state and contents respectively and write $c.\mathbf{n}(b)$ for the number of balloons b in c.

The edges in E define a transition relation on configurations. For an edge $q \xrightarrow{op} q'$, and configurations $c = (q, \mathbf{m}, \mathbf{n})$ and $c' = (q', \mathbf{m}', \mathbf{n}')$, we define $c \xrightarrow{op} c'$ iff one of the following is true:

(1) If $op = \delta \in \mathbb{Z}^P$ and $\mathbf{m}' = \mathbf{m} + \delta$ and $\mathbf{n}' = \mathbf{n}$.

(2) If $op = \mathsf{inflate}(\sigma, S)$ and $\mathbf{m}' = \mathbf{m}$ and $\mathbf{n}' = \mathbf{n} + [\![(\sigma, \mathbf{k})]\!]$ for some $\mathbf{k} \in S$. That is, we create a new balloon with state σ and multiset $\mathbf{k}$ for some $\mathbf{k} \in S$.
(3) If $op = \mathsf{deflate}(\sigma, \sigma', \pi, p)$ and there is a balloon $b = (\sigma, \mathbf{k}) \in \Omega \times \mathbb{M}[\Phi]$ with $\mathbf{n}(b) \geq 1$ and $\mathbf{m}' = \mathbf{m} + \mathbf{k}(\pi) \cdot [\![p]\!]$ and $\mathbf{n}' = (\mathbf{n} - [\![b]\!]) + [\![(\sigma', \mathbf{k}')]\!]$, where $\mathbf{k}'(\pi) = 0$ and $\mathbf{k}'(\pi') = \mathbf{k}(\pi')$ for all $\pi' \in \Phi \setminus \{\pi\}$. That is, we pick a balloon $(\sigma, \mathbf{k})$ from $\mathbf{n}$, transfer the contents in place π from $\mathbf{k}$ to place p in $\mathbf{m}$, and update the balloon state σ to σ'. Here we say the balloon $(\sigma, \mathbf{k})$ was *deflated*.
(4) If $op = \mathsf{burst}(\sigma)$ and there is a balloon $b = (\sigma, \mathbf{k}) \in \Omega \times \mathbb{M}[\Phi]$ with $\mathbf{n}(b) \geq 1$ and $\mathbf{m}' = \mathbf{m}$ and $\mathbf{n}' = \mathbf{n} - [\![b]\!]$. This means we pick some balloon b with state σ from our multiset $\mathbf{n}$ of balloons and remove it, making any tokens still contained in its balloon places disappear as well. Here we say the balloon b is *burst*.

The edge set E is the disjoint union of the sets E_p, E_n, E_d, E_b which stand for the edges with operations from (1),(2),(3),(4) respectively. We write $c \to c'$ if $c \xrightarrow{op} c'$ for some edge $q \xrightarrow{op} q'$ in E. A *run* $\rho = c_0 \xrightarrow{op_0} c_1 \xrightarrow{op_1} c_2 \xrightarrow{op_2} \cdots$ is a finite or infinite sequence of configurations. The size of $\mathcal{V} = (Q, P, \Omega, \Phi, E)$ is given by $|\mathcal{V}| = |Q| + |P| + |\Omega| + |\Phi| + |E|$.

An infinite run ρ is *progressive* iff the following holds:

(1) For every configuration $c_i = (q_i, \mathbf{m}_i, \mathbf{n}_i)$ and every balloon $b = (\sigma, \mathbf{k}) \in \Omega \times \mathbb{M}[\Phi]$ with $\mathbf{n}_i(b) \geq 1$ there is a c_j, $j > i$, such that op_j either bursts or deflates b.
(2) Moreover, for every configuration $c_i = (q_i, \mathbf{m}_i, \mathbf{n}_i)$ and every place $p \in P$ with $\mathbf{m}_i(p) \geq 1$ there is a c_j, $j > i$, such that a token is removed from p; that is, $op_j \equiv \delta$ with $\delta(p) < 0$.

We define the balloon-norm of a configuration $c = (q, \mathbf{m}, \mathbf{n})$ as $\|c\| = \max\{\sum_{p \in \Phi} \mathbf{k}(p) \mid \exists \sigma\ \mathbf{n}(\sigma, \mathbf{k}) > 0\}$. A progressive run is *shallow* if there is a number $B \in \mathbb{N}$ such that $\|c_j\| \leq B$ for all $j \geq 0$. In other words, shallowness of a run means that each balloon in every configuration on the run contains at most B tokens in the balloon places. Note that this does not mean the size of the configurations become bounded: the number of balloons and the number of tokens in P can still be unbounded.

The *progressive run problem for* VASSB is the following:

Given A VASSB $\mathcal{V} = (Q, P, \Omega, \Phi, E)$ and an initial semiconfiguration c_0.
Question Does $\mathcal{V}$ have an infinite progressive run starting from c_0?

In Section 5, we shall prove the following theorems.

Theorem 4.1. *The progressive run problem for* VASSB *is decidable.*

The following is a by-product of the proof of Theorem 4.1, which will be used in Section 6.

Theorem 4.2. *A* VASSB $\mathcal{V}$ *has a progressive run iff it has a shallow progressive run.*

4.2 From DCPS to VASSB

Instead of reducing fairness for DCPS to VASSB, we would like to use a stronger notion, which simplifies many of our proofs. To this end, we introduce the notion of progressiveness that we already defined for VASSB now for DCPS as well: given a bound $K \in \mathbb{N}$, an infinite run ρ of a DCPS is called *progressive* if the rule Term is only ever applied when the active thread is at K context switches, and for every local configuration (w, i) of an inactive thread in a configuration of ρ, there is a future point in ρ where the rule Resume is applied to (w, i), making it the local configuration of the active thread.

Intuitively, no type of thread can stay around infinitely long in a progressive run without being resumed, and every thread that terminates does so after exactly K context switches. Note that progressiveness is a stronger condition than fairness, because it does not allow threads to "get stuck"

or go above the context switch bound K. However, we can always transform a DCPS where we want to consider fair runs into one where we can consider progressive runs instead. The transformation is formalized by the following lemma.

Lemma 4.3. *Given a bound $K \in \mathbb{N}$ and a* DCPS $\mathcal{A}$*, we can construct a* DCPS $\widetilde{\mathcal{A}}$ *such that:*

- *$\mathcal{A}$ has an infinite fair K-context switch bounded run iff $\widetilde{\mathcal{A}}$ has an infinite progressive K-context switch bounded run.*
- *$\mathcal{A}$ has an infinite fair K-context switch bounded run that starves a thread iff $\widetilde{\mathcal{A}}$ has an infinite progressive K-context switch bounded run that starves a thread.*

Idea. To prove Lemma 4.3 we modify the DCPS $\mathcal{A}$ by giving every thread a bottom of stack symbol $\perp$ and saving its context switch number in its top of stack symbol. We also save this number in the global state whenever a thread is active. This way we can still swap a thread out and back in again once it has emptied its stack, and we also can keep track of how often we need to repeat that, before we reach K context switches and allow it to terminate.

Furthermore, we also keep a subset G' of the global states of $\mathcal{A}$ in our new global states, which restricts the states that can appear when no thread is active. This way we can guess that a thread will be "stuck" in the future, upon which we terminate it instead (going up to K context switches first) and also spawn a new thread keeping track of its top of stack symbol in the bag. Then later we restrict the subset G' to only those global states that do not have Resume rules for the top of stack symbols we saved in the bag. This then verifies our guess of "being stuck". The second part of the lemma is used in Section 6, where we reason about starvation.

We now state the main reduction to VASSB.

Theorem 4.4. *Given a bound $K \in \mathbb{N}$ and a* DCPS $\mathcal{A}$ *we can construct a* VASSB $\mathcal{V}$ *with a state q_0 such that $\mathcal{A}$ has an infinite progressive K-context switch bounded run iff $\mathcal{V}$ has an infinite progressive run from $(q_0, \emptyset, \emptyset)$.*

Idea. One of the main insights regarding the behavior of DCPS is that the order of the spawns of a thread during one round of being active does not matter. None of the spawned threads during one such segment can influence the run until the active thread changes. Thus we only need to look at the semi-linear Parikh image of the language of spawns for each segment. One can then identify a thread with the state changes and spawns it makes during segments 0 through K. The state changes can be stored in a balloon state and the spawns for each segment in balloon places that correspond to $K + 1$ copies of the stack alphabet. The inflate operation then basically guesses the exact multiset of spawns of the corresponding thread.

Representing threads by balloons in this way does not keep track of stack contents, which was important for ensuring the progressiveness of a DCPS run. However, starting from a progressive DCPS run we can always construct a progressive run for the VASSB by always continuing with the oldest thread in a configuration if given multiple choices, and then building the balloons accordingly. Always picking the oldest balloon also works for the reverse direction.

Now let us argue about the construction in more detail. Given a DCPS with stack alphabet Γ and a context switch bound K, construct a VASSB whose configurations mirror the ones of the DCPS in the following way. The set of places is Γ and it is used to capture threads that have not been scheduled yet and therefore only carry a single stack symbol. Formally each thread with context switch number 0 and stack content $\gamma \in \Gamma$ is represented by a token on place γ. The set of balloon places is $\Gamma \times \{0, \ldots, K\}$ and they are supposed to carry the future spawns of any given thread during segments 0 to K. Every thread t with context switch number ≥ 1 is then represented by a balloon where the number of tokens on balloon place (γ, i) is equal to the number of threads with stack content γ that t will spawn during its ith segment. The spawns for segment 0 are transferred

to the place set Γ immediately after such a balloon is created, since the represented thread is now supposed to have made its first context switch. Furthermore, each balloon state consists of the context switch sequence and context switch number of its corresponding thread t. The set of states of the VASSB mirrors the global states of the DCPS.

For this idea to work, we need to compute the semi-linear set of spawns that each type of thread can make, so that we can correctly inflate the corresponding balloon using this set. Here, the *type* of a terminating thread consists of the stack symbol γ it spawns with, the global state g in which it first becomes active, the sequence of context switches it makes, and the state in which it terminates. Given a DCPS $\mathcal{A} = (G, \Gamma, \Delta, g_0, \gamma_0)$ and a context switch bound K, the formal definition of the set of thread types is

$$\mathcal{T}(\mathcal{A}, K) \coloneqq G \times \Gamma \times (G \times \Gamma \times G)^K \times G.$$

Since we want to decide existence of an infinite progressive run of $\mathcal{A}$, we can restrict ourselves to threads that make exactly K context switches. Now let $t = (g_0', \gamma_0', (g_1, \gamma_1, g_1') \dots (g_K, \gamma_K, g_K'), g_{K+1}) \in \mathcal{T}(\mathcal{A}, K)$ be a thread type. We want to use $\mathcal{P}_{(g_0', \gamma_0')}$, the PDA of a thread of this type, to accept the language of spawns such a thread can make. However, we have two requirements on this language, that the PDA does not yet fulfill. Firstly, in the spirit of progressiveness, we only want to consider threads that reach the empty stack and terminate. Secondly, we want the spawns during each segment of the thread execution to be viewed separately from one another.

For the first requirement, we modify the transition relation of $\mathcal{P}_{(g_0', \gamma_0')}$, such that transitions of the form $(g_2, \bot) \xrightarrow{(g_2, \bot)|\gamma_2/\gamma_2}$ end defined in (5) are only kept in the relation for $\gamma_2 = \bot$. This ensures that the PDA no longer considers thread executions that do not reach the empty stack.

Regarding the second requirement, we can simply introduce $K + 1$ copies of Γ to the input alphabet of $\mathcal{P}_{(g_0', \gamma_0')}$. It is then redefined as $\Sigma = \Gamma \times \{0, \dots, K\} \cup G \times \Gamma \times G \cup G \times \{\bot\}$, while the stack alphabet and states stay the same. Any transition previously defined on input $\gamma \in \Gamma$ is now copied for inputs $(\gamma, 0)$ to (γ, k).

Let $\widetilde{\mathcal{P}}_{(g_0', \gamma_0')}$ be the PDA these changes result in. Then the context-free language that characterizes the possible spawns of a thread of type t is given by the following:

$$L_t \coloneqq L(\widetilde{\mathcal{P}}_{(g_0', \gamma_0')}) \cap (\Gamma \times 0)^* \cdot (g_1, \gamma_1, g_1') \cdot (\Gamma \times 1)^* \cdots (g_K, \gamma_K, g_K') \cdot (\Gamma \times K)^* \cdot (g_{K+1}, \bot)$$

Here we intersect the language of the PDA with a regular language, which forces it to adhere to the type t and groups the spawns correctly. If the language L_t is nonempty, using Parikh's theorem (Theorem 2.1), we can compute the semi-linear set characterizing the Parikh-image of this language projected to $\Gamma \times \{0, \dots, K\}$, which we denote $\mathsf{sl}(t)$. We also define the set of all semi-linear sets that arise in this way as $\mathsf{SL}(\mathcal{A}, K) \coloneqq \{\mathsf{sl}(t) \mid t \in \mathcal{T}(\mathcal{A}, K), L_t \neq \emptyset\}$.

Now we can construct a VASSB whose configurations correspond to the ones of the DCPS in the way we mentioned earlier. From $(q_0, \emptyset, \emptyset)$ we put a token on γ_0 to simulate spawning the initial thread and thus begin the simulation of the DCPS. We can then construct a progressive run of the VASSB from a progressive run of the DCPS by constructing the individual balloons as if the scheduler always picked the oldest thread out of all choices with the same local configuration. The converse direction works for similar reasons by always picking the oldest balloon to continue with.

The proof also allows us to reason about *shallow* progressive runs of DCPS. Following the same notion for VASSB, we call a run of a DCPS *shallow* if there is a bound $B \in \mathbb{N}$ such that each thread on that run spawns at most B threads. Obverse that in the VASSB construction of this section the spawns of DCPS threads are mapped to the contents of balloons, which is how the two notions of shallowness correspond to each other. Thus we can go from progressive DCPS-run to progressive VASSB-run by Theorem 4.4, to shallow progressive VASSB-run by Theorem 4.2, to

shallow progressive DCPS run by Theorem 4.4 combined with the observation on the two notions of shallowness. This is formalized in the following:

Corollary 4.5. *A DCPS $\mathcal{A}$ has a progressive run iff it has a shallow progressive run.*

5 FROM PROGRESSIVE RUNS FOR VASSB TO REACHABILITY

In this section, we prove Theorems 4.1 and 4.2. We outline the main ideas and technical lemmas used to obtain the proofs. The formal proofs can be found in the full version of the paper [Baumann et al. 2020b].

We first establish that finite witnesses exist for infinite progressive runs. As a byproduct, this yields Theorem 4.2. Then, we show that finding finite witnesses for progressive runs reduces to reachability in VASSB. Finally, we prove that reachability is decidable for VASSB.

5.1 From Progressive Runs to Shallow Progressive Runs

Fix a VASSB $\mathcal{V} = (Q, P, \Omega, \Phi, E)$. A *pseudoconfiguration* $\mathrm{p}(c) = (q, \mathbf{m}, \partial\mathbf{n}) \in Q \times \mathbb{M}[P] \times \mathbb{M}[\Omega]$ of a configuration $c = (q, \mathbf{m}, \mathbf{n})$ is given by $\partial\mathbf{n}(\sigma) = \sum_{\mathbf{k} \in \mathbb{M}[\Phi]} \mathbf{n}(\sigma, \mathbf{k})$. That is, a pseudoconfiguration is obtained by counting the number of balloons in a given state $\sigma \in \Omega$ but ignoring the contents. The *support* $\mathrm{supp}(\mathbf{m})$ of a multiset $\mathbf{m}$ is the set of places $\{p \mid \mathbf{m}(p) > 0\}$ where $\mathbf{m}$ takes non-zero values.

For configurations $c = (q, \mathbf{m}, \mathbf{n})$ and $c' = (q', \mathbf{m}', \mathbf{n}')$, we write $c \leq c'$ if $q = q'$, $\mathbf{m} \leq \mathbf{m}'$, and $\mathbf{n} \leq \mathbf{n}'$. Moreover, we write $\mathrm{p}(c) \leq_{\mathrm{p}} \mathrm{p}(c')$ if $q = q'$, $\mathbf{m} \leq \mathbf{m}'$, and $\partial\mathbf{n} \leq \partial\mathbf{n}'$. Both $\leq$ and $\leq_{\mathrm{p}}$ are *well-quasi orders* (wqo), that is, any infinite sequence of configurations (resp. pseudoconfigurations) has an infinite increasing subsequence with respect to $\leq$ (resp. $\leq_{\mathrm{p}}$) (see, e.g., [Abdulla et al. 1996; Finkel and Schnoebelen 2001] for details). This follows because wqos are closed under multiset embeddings. Moreover, $\mathcal{V}$ is *monotonic* w.r.t. $\leq$: if $c_1 \to c_1'$ and $c_1 \leq c_2$, then there is some c_2' such that $c_1' \leq c_2'$ and $c_2 \to c_2'$.

In analogy with algorithms for finding infinite runs in VASS (in particular the procedure for checking fair termination in the case $K = 0$ in [Ganty and Majumdar 2012]), one might try to find a self-covering run w.r.t. the ordering $\leq$. However, checking for such a run would require comparing an unbounded collection of pairs of balloons. In order to overcome this issue, we use a *token-shifting surgery* which moves tokens from one balloon to another. The surgery is performed on the given progressive run ρ, converting it into a progressive run ρ' with a special property: there exist infinitely many configurations in ρ' which contain only empty balloons. By restricting ourselves to such configurations, we are able to show the existence of a special kind of self-covering run where the cover and the original configuration only contain empty balloons and thus it suffices to compare them using the ordering $\leq_{\mathrm{p}}$. First, we need the following notion of a witness for progressiveness.

For $A \subseteq P, B \subseteq \Omega$, a run $\rho_{A,B} = c_0 \xrightarrow{*} c \xrightarrow{*} c'$ is called an *A,B-witness* for progressiveness if it satisfies the following properties:

(1) For any c'' occurring between c and c', we have $\mathrm{supp}(c''.\mathbf{m}) \subseteq A$,
(2) for each $p \in A$, there exists op between c and c' in $\rho_{A,B}$ such that $op = \delta$ where $\delta(p) < 0$,
(3) for any balloon b, we have $c.\mathbf{n}(b) \geq 1$ iff $c'.\mathbf{n}(b) \geq 1$ iff $(b.\mathbf{k} = \emptyset$ and $b.\sigma \in B)$,
(4) for any $\sigma \in B$, there exists op occurring between c and c' such that $op = \mathrm{deflate}(\sigma, \cdot, \cdot, \cdot)$ or $op = \mathrm{burst}(\sigma)$ is applied to an empty balloon with state σ, and
(5) $\mathrm{p}(c) \leq_{\mathrm{p}} \mathrm{p}(c')$ and $\mathrm{supp}(c.\mathbf{m}) = \mathrm{supp}(c'.\mathbf{m})$.

In order to formalize the idea of a token-shifting surgery, we associate a unique identity with each balloon. In particular, we may associate the unique number $i \in \mathbb{N}$ with the balloon b which is inflated by the i^{th} operation op_i in a run $\rho = c_0 \xrightarrow{op_1} c_1 \xrightarrow{op_2} c_2 \cdots$, giving us a *balloon-with-id* (b, i). The id of a balloon is preserved on application of deflate and burst operations and thus we

can speak of *the balloon i* and the sequence of operations seq_i that it undergoes. Given a run ρ, we produce a corresponding *canonical run-with-id* τ inductively as follows: the balloon identities are assigned as above and the balloon with least id is chosen for execution every time. Extending the notion of a balloon-with-id to a configuration-with-id d (where every balloon has an id) and a run-with-id τ (which consist of sequences of configurations-with-id), we define the notion of a *progressive* run-with-id τ, which is one such that the sequence seq_i associated with any id i in τ is either infinite, or seq_i is finite with the last operation being a burst. If every id i undergoes a burst operation in a run-with-id τ, we say "τ bursts every balloon." We abuse terminology by saying "ρ bursts every balloon" for a run ρ to mean that there is a *corresponding* run-with-id τ which bursts every balloon. It is easy to see that the canonical run-with-id τ associated with a progressive run ρ is "almost" progressive. By always picking the id which has been idle for the longest time, we can convert τ into a progressive run-with-id. Thus there exists a progressive run if and only if there exists a progressive run-with-id, but the latter retains more information, allowing us to argue formally in proofs. With these notions in hand, we prove the following lemma:

Lemma 5.1. *Given a* VASSB $\mathcal{V}$ *and a semiconfiguration s of $\mathcal{V}$, one can construct a* VASSB $\mathcal{V}' = (Q', P', \Omega', \Phi', E')$ *and a semiconfiguration s' of $\mathcal{V}'$ such that the following are equivalent:*

(1) $\mathcal{V}$ has a progressive run from s,
(2) $\mathcal{V}'$ has a progressive run from s' that bursts every balloon, and
(3) $\mathcal{V}'$ has an A, B-witness for some $A \subseteq P'$ and $B \subseteq \Omega'$.

The remainder of this subsection is devoted to the proof of Lemma 5.1. The lemma is proved in two steps: first we show (1) $\iff$ (2), then we show (2) $\iff$ (3). We do some preprocessing before (1) $\iff$ (2), by showing that one can convert $\mathcal{V}$ into a VASSB $\mathcal{V}'$ with two special properties: (i) the *zero-base* property, by which every linear set of $\mathcal{V}'$ has base vector equal to $\mathbf{0}$, and (ii) the property of being *typed*, which means that we guess and verify the sequence of deflates performed by a balloon b that could potentially transfer a non-zero number of tokens by including this information in its balloon state $b.\sigma$. A deflate operation which transfers a non-zero number of tokens is called a *non-trivial* deflate. The *type* $t = (L, S)$ of a balloon i consists of the linear set L used during its inflation, along with the sequence $S = (\pi_1, p_1), (\pi_2, p_2), \cdots, (\pi_n, p_n)$ of deflate operations on i, such that for each $j \in \{1, \cdots, n\}$, the first deflate operation acting on the balloon place π_j sends tokens to the place p_j.

Lemma 5.2. *Given a* VASSB $\mathcal{V}$ *along with its semiconfigurations s_0, s_1, we can construct a zero-base, typed* VASSB $\mathcal{V}'$ *and its semiconfigurations s_0', s_1' such that:*

(1) There is a progressive run of $\mathcal{V}$ from s_0 iff there is a progressive run of $\mathcal{V}'$ from s_0'.
(2) There is a run $s_0 \xrightarrow{*} s_1$ in $\mathcal{V}$ iff there is a run $s_0' \xrightarrow{*} s_1'$ in $\mathcal{V}'$.

The zero-base property is easily obtained by making sure that the portion of tokens transferred which correspond to the base vector are separately transferred using E_p-edges. The addition of the types into the global state can be done by expanding the set of balloon states exponentially. A proof of the lemma is given in the full version [Baumann et al. 2020b].

We return to the proof of Lemma 5.1. The direction (1) $\Rightarrow$ (2) requires us to show that $\mathcal{V}'$ can be assumed to "burst every balloon" in a progressive run-with-id. Consider a balloon i occurring in an arbitrary progressive run-with-id τ''. In order to convert the given τ'' into a progressive τ' in which every balloon is burst, we need to burst those id's i such that seq_i is infinite in τ''. Since only a finite number of non-trivial deflates can be performed by a given balloon i, this implies that seq_i consists of a finite prefix in which non-trivial deflates are performed, followed by an infinite suffix of trivial deflates. Every such balloon i can then be burst and replaced by a special VASS token. The

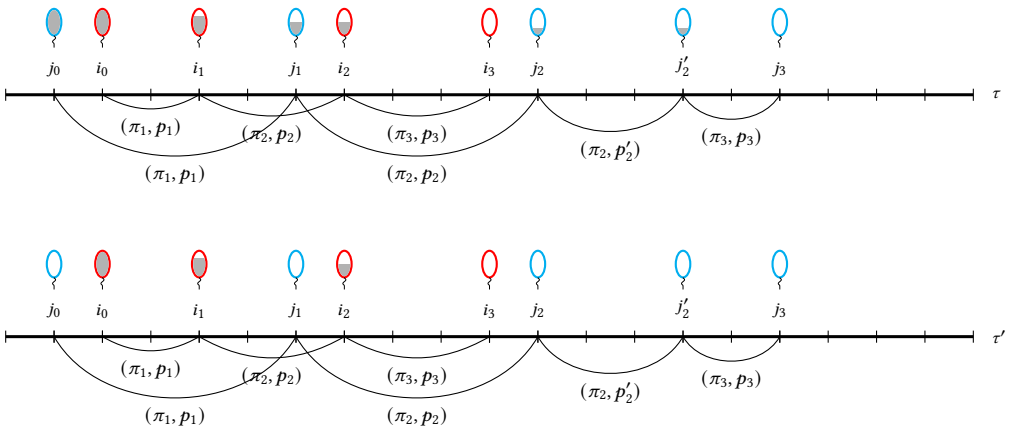

Fig. 1. Top: Initial run τ with two non-empty balloons which perform the same sequence of three non-trivial deflates $(\pi_1, p_1), (\pi_2, p_2), (\pi_3, p_3)$. Bottom: Modified run τ' after shifting tokens from the cyan balloon inflated at j_0 to the red balloon inflated at i_0.

infinite trivial suffix is then simulated by using addition and subtraction operations of these special tokens using additional places, since any such balloon i will not transfer any more tokens. The converse direction (1) $\Leftarrow$ (2) is a reversal of the construction where we replace the special VASS tokens with infinite sequences of trivial deflate operations.

Token-Shifting. We move on to show (2) $\iff$ (3). The key idea is a token-shifting surgery. A *token-shifting surgery* creates a run τ' from a run τ as depicted in Figure 1. We start with the run-with-id $\tau = d_0 \xrightarrow{op_1} d_1 \xrightarrow{op_2} d_2 \cdots$ in which two balloons have the same type $t = (L, S)$ with $S = (\pi_1, p_1), (\pi_2, p_2), (\pi_3, p_3)$ being the sequence of (potentially) non-trivial deflates. Recall that an index i relates to the operation op_i in τ. The region of a balloon which is shaded grey visualizes the total number of tokens contained in the balloon. This number is seen to decrease after each non-trivial deflate operation. An empty balloon is white in color. Balloons having the same identity have the same outline color: red for the balloon b_1 inflated at i_0 and cyan for the balloon b_2 inflated at j_0. At i_3 (resp. j_3) all tokens have been transferred and the red balloon (resp. cyan balloon) is empty. The crucial property satisfied is that $i_k < j_k$ for $k \in \{1, 2, 3\}$, i.e., every deflate from S of the red balloon occurs before the corresponding deflate of the cyan balloon. In the modified run τ', we have the inflation of an empty cyan balloon and the inflation of the red balloon with the sum of the tokens of both red and cyan balloons in τ. We require the zero-base property in order to be able to shift the tokens in this manner: observe that a linear set with zero base vector is closed under addition. Note that the cyan balloon undergoes a trivial deflate at j_2' where no tokens are transferred: this deflate is not part of S and is not relevant for the token-shifting. Thus the run-with-id τ may be modified to give a *valid* run-with-id τ'. Note that the configurations-with-id d_j' of τ' satisfy $d_j' \geq d_j$ for each j and so by monotonicity every operation that is applied at d_j can be applied at d_j'.

We now show how token-shifting is applied to prove (2) $\iff$ (3) in Lemma 5.1. First, from a progressive run τ of $\mathcal{V}'$ such that every balloon is burst, we produce a run τ' of $\mathcal{V}'$ from which it will be easy to extract an A, B-witness. Let T_∞ be the set of types of balloons which occur infinitely often in τ. Since every balloon is eventually burst in τ, there has to be a configuration d_0 such that after d_0, every occurring balloon has a type in T_∞. We now inductively pick a sequence of

configurations $d_1, d_2, \cdots$ and a sequence of sets of balloons $I_1, I_2, \cdots$ with the following properties, for each $k \geq 1$:

(1) I_k contains exactly one balloon inflated after d_{k-1} for each type in T_∞ and
(2) every balloon in I_k is burst before d_k.

For every balloon i not in any of the sets $I_1, I_2, \cdots$, which is inflated between d_k and d_{k+1} for $k \geq 1$, we shift its tokens to the corresponding balloon j in I_k of the same type as i. Clearly this is allowed since all of the deflate operations of j occur before i is inflated. Thus we obtain the run $\tau' = d_0'' \xrightarrow{*} d_0' \xrightarrow{*} d_1' \xrightarrow{*} d_2' \ldots$ from τ. The prefix $d_0'' \xrightarrow{*} d_0'$ of the modified run τ' may contain balloons of arbitrary type. Between d_0' and d_1', there are only balloons in T_∞. After d_2', we have an infinite suffix where all balloons are of a type from T_∞ and the only non-empty balloons are those belonging to I_k for some $k \geq 2$. This means that the configurations-with-id d_k' for each $k \geq 2$ only contain empty balloons. Since $\leq_{\mathrm{p}}$ is a well-quasi-ordering, the sequence $d_2', d_3', \ldots$ must contain configurations d_l' and d_m' with $d_l' \leq_{\mathrm{p}} d_m'$. We obtain an A, B-witness as follows. We choose the set P_∞ which is the set of places which are non-empty infinitely often along τ' for the set A. Since the set of possible balloon states and places in a given configuration is finite, by Pigeonhole Principle, we may assume that d_l' and d_m' have the same set of non-empty places $A \subseteq P'$ and balloon states $B \subseteq \Omega$. We may also assume that progressiveness checks (2) and (4) corresponding to A and B occur between d_l' and d_m'.

Conversely, an (A, B)-witness $\rho_{A,B}$ can be "unrolled" to give a progressive run τ'' of $\mathcal{V}'$. Furthermore, since the unrolling τ'' only contains balloons with contents present in the finite run $\rho_{A,B}$, giving us a shallow progressive run as stated in Theorem 4.2.

5.2 Reduction to Reachability

The *reachability problem* REACH for VASSB asks:

Given A VASSB $\mathcal{V} = (Q, P, \Omega, \Phi, E)$ and two semiconfigurations c_0 and c.

Question Is there a run $c_0 \xrightarrow{*} c$?

The more general version of the problem, where c_0 and c can be arbitrary configurations (i.e., with balloon contents), easily reduces to this problem. However, the exposition is simpler if we restrict to semiconfigurations here. In this subsection, we shall reduce the progressive run problem for VASSB to the *reachability problem* for VASSB. In the next subsection, we shall reduce the reachability problem to the reachability problem for VASS, which is known to be decidable [Kosaraju 1982; Mayr 1981].

Lemma 5.3. *The progressive run problem for* VASSB *reduces to the problem* REACH *for* VASSB.

Fix a VASSB $\mathcal{V}$. Using Lemma 5.1, we look for progressive witnesses. Let $A \subseteq P$ and $B \subseteq \Omega$. We shall iterate over the finitely many choices for $T = (A, B)$ and check that $\mathcal{V}$ has an infinite progressive run with a A, B-witness by reducing to the configuration reachability problem for an associated VASSB $\mathcal{V}(T)$.

The VASSB $\mathcal{V}(T)$ simulates $\mathcal{V}$ and guesses the two configurations c_1 and c_2 such that $c_0 \xrightarrow{*} c_1 \xrightarrow{*} c_2$ satisfies the conditions for a progressive witness. It operates in five total stages, with three main stages and two auxiliary ones sandwiched between the main stages. In the first main stage, it simulates two identical copies of the run of $\mathcal{V}$ starting from c_0. The global state is shared by the two copies while we have separate sets of places. We cannot maintain separate sets of balloons for each copy since the inflate operation is inherently non-deterministic and hence the balloon contents may be different in the two balloons produced. The trick to maintaining two copies of the same balloon is to in fact only inflate a single instance of a "doubled" balloon which uses "doubled"

vectors and two copies of balloon places. Deflate operations are then performed twice on each doubled balloon, moving tokens to the corresponding copies of places.

The VASSB $\mathcal{V}(T)$ also tracks the number of balloons in each balloon state (independent of their contents), for each copy of the run. At some point, it guesses that the current configuration is c_1 (in both copies) and moves to the first auxiliary stage. The first auxiliary stage checks whether all the balloons in c_1 are empty. Control is then passed to the second main stage.

In the second main stage, the first copy of the run is frozen to preserve $\mathrm{p}(c_1)$ and $\mathcal{V}(T)$ continues to simulate $\mathcal{V}$ on the second copy. This is implemented by producing only "single" balloons during this stage and only deflating the second copy of the places in the "double" balloons which were produced in the first main stage. While simulating $\mathcal{V}$ on the second copy, $\mathcal{V}(T)$ additionally checks the progressiveness constraints (2) and (4) corresponding to an A, B-witness in its global state. The second main stage non-deterministically guesses when the second copy reaches c_2 (and ensures all progress constraints have been met) and moves on to the second auxiliary stage. Here the fact that all the balloons in c_2 are empty is checked and then control passes to the third main stage. In the third main stage, $\mathcal{V}(T)$ verifies that the two configurations c_1 and c_2 also satisfy conditions (1), (3), and (5) for an A, B-witness. A successful verification puts $\mathcal{V}(T)$ in a specific final semi-configuration.

5.3 From Reachability in VASSB to Reachability in VASS

In this subsection, we show that reachability for VASSB reduces to reachability in ordinary VASS. We write $\exp_k(x)$ for the k-fold exponential function i.e. $\exp_1(x)$ is 2^x, $\exp_2(x)$ is 2^{2^x} etc.

A run $\rho = s_1 \xrightarrow{*} s_2$ of a VASSB $\mathcal{V}$ between two semiconfigurations is said to be *N-balloon-bounded* for some $N \in \mathbb{N}$ if there exist at most N non-empty balloons which are inflated in ρ. The following lemma is the crucial observation for our reduction.

Lemma 5.4. *Given any* VASSB $\mathcal{V} = (Q, P, \Omega, \Phi, E)$*, there exists* $N \in \mathbb{N}$ *with* $N \leq O(\exp_4(|\mathcal{V}|))$ *such that for any two semiconfigurations* s_1, s_2*, if* $(\mathcal{V}, s_1, s_2) \in$ REACH*, then there exists a run* $\rho = s_1 \xrightarrow{*} s_2$ *of* $\mathcal{V}$ *that is* N*-balloon-bounded.*

Before we prove Lemma 5.4, let us see how it allows us to reduce reachability in VASSB to reachability in VASS.

Lemma 5.5. *The problem* REACH *for* VASSB *reduces to the problem* REACH *for* VASS.

From a given VASSB $\mathcal{V}$, we construct a VASS $\mathcal{V}'$ which has extra places $\Omega \times \{1, \cdots, N\} \times \Phi$ for storing the contents of all the non-empty balloons, as well as extra places Ω that store the number of balloons which were created empty for each balloon state σ of $\mathcal{V}$. The global state of $\mathcal{V}'$ is used to keep track of the total number of non-empty balloons created as well as their state changes. Deflate and burst operations are replaced by appropriate token transfers such that there is only one opportunity for $\mathcal{V}'$ to transfer tokens of any non-empty balloon by using the global state. This results in a faithful simulation in the forward direction, as well as the easy extraction of a run of $\mathcal{V}$ from a run of $\mathcal{V}'$ in the converse direction.

Proof of Lemma 5.4. We now prove Lemma 5.4, which will complete the proof of Theorem 4.1.

We observe that if, for every balloon state σ, the number of balloons that are inflated in ρ with state σ is bounded by N, this implies a bound of $|\Omega|N$ on the total number of balloons inflated in ρ. Hence, we can equivalently show the former bound assuming a particular balloon state. We assume that $\mathcal{V}$ is both zero-base and typed while preserving reachability by Lemma 5.2. This implies that the type information is contained in the state of a balloon. The lemma is then proved by showing that if more than N non-empty balloons of a particular state σ are inflated in a run $\rho = s_1 \xrightarrow{*} s_2$,

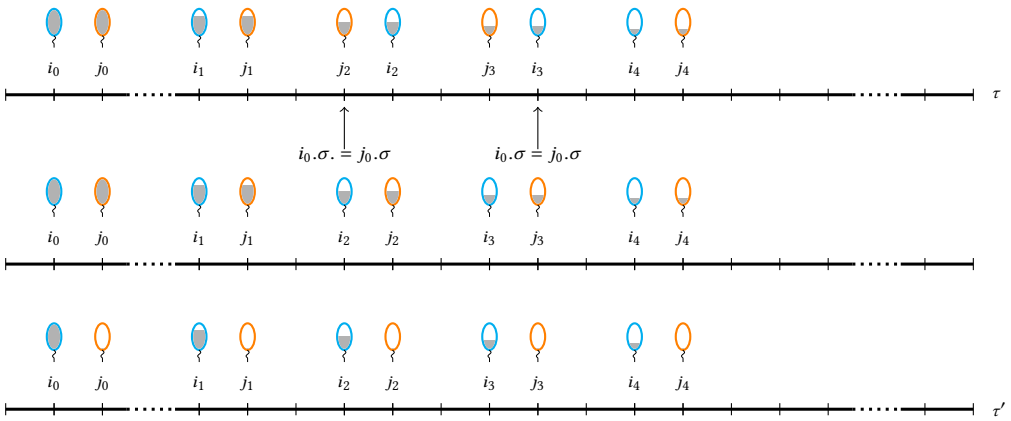

Fig. 2. Top: Initial run τ with two non-empty balloons of the same type: cyan balloon inflated at i_0 and orange balloon inflated at j_0. Middle: Switching cyan and orange balloons in the part of τ between j_2 and i_3. Bottom: Modified run τ' obtained by shifting token from orange balloon to cyan balloon.

then it is possible to perform an *id-switching* surgery, resulting in a run $\rho' = s_1 \xrightarrow{*} s_2$ which creates one less non-empty balloon with state σ.

The id-switching surgery is depicted in Figure 2 . The formal proof of correctness uses runs-with-id. Fix a run-with-id $\tau = d_0 \xrightarrow{op_1} d_1 \xrightarrow{op_2} d_2 \cdots$. Suppose the cyan and orange balloons are inflated at i_0 and j_0 respectively with the same balloon state. Since the type information is included in the balloon state, this implies that they are also of the same type $t = (L_t, S_t)$. The points marked with indices i_1, i_2, i_3, i_4 (resp. j_1, j_2, j_3, j_4) are those at which the cyan (resp. orange) balloon undergoes a deflate from S_t. While $i_1 < j_1$ and $i_4 < j_4$, we have $j_2 < i_2$ and $j_3 < i_3$; therefore token-shifting is not possible. However, let us assume that the state of the cyan and orange balloons is the same at d_{j_2-1} and d_{i_3} of τ. This implies that the operations performed on the two balloons in $\tau[j_2, i_3]$ can be switched as shown in the middle of Figure 2. Note that this need not be a valid run as the number of tokens transferred by the orange at j_2 may exceed that transferred by the cyan balloon at i_2 and the extra tokens may be required for the run $\tau[j_2, i_2]$ to be valid. However, the id-switching now enables a token-shifting operation since $j_k < i_k$ for each $k \in \{0, 1, \ldots, 4\}$. Thus, combining the switch with a token-shifting operation which moves all tokens from orange to cyan results in the valid run τ' shown at the bottom of Figure 2, which contains one less non-empty balloon of type t than τ. It remains to show that such an id-switching surgery is always possible in a run τ when the number of non-empty balloons of a type t exceeds the bound N given in the lemma.

Ramsey's Theorem. To this end, we employ the well-known (finite) Ramsey's theorem [Ramsey 1930, Theorem B], which we recall first. For a set S and $k \in \mathbb{N}$, we denote by $\mathbb{P}_k(S)$ the set of all k-element subsets of S. An *r-colored (complete) graph* is a tuple $(V, E_1, \ldots, E_r)$, where V is a finite set of *vertices* and the sets $E_1, \ldots, E_r$ form a partition of all possible edges (i.e. two-element subsets), i.e. $\mathbb{P}_2(V) = E_1 \dot\cup \cdots \dot\cup E_r$. A subset $U \subseteq V$ of vertices is *monochromatic* if all edges between members of U have the same color, in other words, if $\mathbb{P}_2(U) \subseteq E_j$ for some $j \in [1, r]$. Ramsey's theorem says that for each $r, n \in \mathbb{N}$, there is a number $R(r; n)$ such that any r-colored graph with at least $R(r; n)$ vertices contains a monochromatic subset of size n. It is a classical result by Erdős and Rado [Erdős and Rado 1952, Theorem 1] that $R(r; n) \leq r^{r(n-2)+1}$.

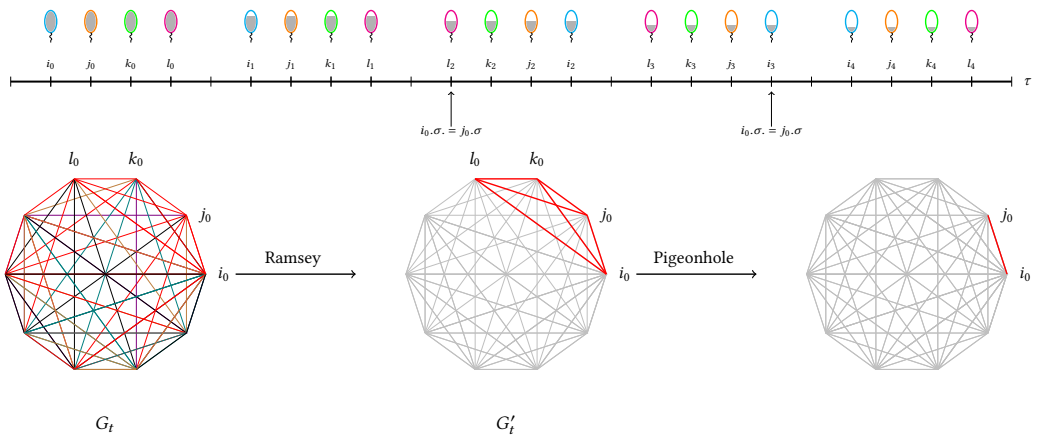

Fig. 3. Above:Four balloons inflated at $i_0 < j_0 < k_0 < l_0$ of the same type t performing four deflate operations (subscripts denote deflate operations of the same balloon). Note that the ordering relationship of deflate operations between any pair of the four balloons is the same: between balloons i_0 and j_0, their deflate sequences are related as $i_1 < j_1, j_2 < i_2, j_3 < i_3, i_4 < j_4$, which is represented by the string 0110. The edge-color red is used to represent 0110 in the figure. The balloons i_0 and j_0 share the same states at configurations $d_{l_2} - 1$ and d_{i_3} of τ.
Below: The same four balloons inflated at $i_0 < j_0 < k_0 < l_0$ shown as forming a monochromatic subgraph G'_t in the graph G_t. For large enough $|G'_t|$, by Pigeonhole Principle we find i_0, j_0 which share the same states.

The application of Ramsey's Theorem is shown in Figure 3. The bottom half of the figure depicts the construction of a graph G_t whose vertices are balloons, on which Ramsey's theorem is applied. This results in the identification of a monochromatic clique with vertices i_0, j_0, k_0, l_0. The top half of the figure shows the deflate operations on the balloons inflated at i_0, j_0, k_0, l_0 in the run τ.

Formally, we construct a graph G_t with vertex set V_t of all id's in τ of a fixed type t. By assumption, $|V_t| \geq N$. For id's $i, j \in V_t$ with $i < j$ and $S = (\pi_1, p_1), \cdots, (\pi_n, p_n)$, define a sequence $s_{i,j} \in \{0,1\}^{|S|}$ by $s_{i,j}(k) = 0$ if and only if i undergoes the deflate transferring tokens from π_k to p_k before j does.

In Figure 3, assuming that $|S| = 4$, we have $s_{i_0,j_0} = 0110$. Interpreting each word from $\{0,1\}^{|S|}$ as a color, we obtain a finite coloring of the edges of G_t. Red colored edges in G_t are to be interpreted as the string 0110, with other colors representing other strings. For a large enough value of N, Ramsey's Theorem gives us a monochromatic subgraph G'_t of G_t induced by a set of vertices V'_t. As shown in the figure, any pair of balloons chosen from the cyan, orange, green and magenta balloons inflated at i_0, j_0, k_0, l_0 respectively, behave in the same way with respect to their order of deflates and thus form a monochromatic subgraph G'_t colored red.

Let a maximal contiguous sequence of 1's in $s_{i,j}$ be called a 1-*block*. Since the number of balloon states is finite, this implies that for a large enough value of $|V'_t|$, there will exist two id's $i_0, j_0 \in |V'_t|$ (represented by the cyan and orange balloons respectively) which share the same states at configurations at the beginning and end of every 1-block by the Pigeonhole Principle. The id-switching surgery can be performed on the cyan and orange balloons, which are the ones considered in the id-switching surgery of Figure 2. While Ramsey's Theorem gives a double-exponential bound on N in order to obtain a large monochromatic subgraph, the second condition requiring same states at the beginning and end of 1-blocks further increases our requirement to $\exp_4$ for id-switching to be enabled.

This concludes the proof of Lemma 5.4 as well as Theorem 4.1.

6 STARVATION

We now prove Theorem 2.4. Let us first explain the additional difficulty of the starvation problem. For deciding progressive termination, we observed that each thread execution can be abstracted by its type and the threads it spawns. (In other words, two executions that agree in these data are interchangeable without affecting progressiveness of a run.) However, for starvation of a thread, it is also important whether each thread visits some stack content w after i context switches. Here, w is not known in advance and has to be agreed upon by an infinite sequence of threads.

Very roughly speaking, we reduce starvation to progressive termination as follows. For each thread, we track its spawned multiset up to some bound B. Using Ramsey's theorem [Ramsey 1930, Theorem B], we show that if we choose B high enough, then this abstraction already determines whether a sequence of thread executions can be replaced with different executions that actually visit some agreed upon stack content w after i context switches. The latter condition permitting replacement of threads will be called "consistency."

A further subtlety is that consistency of the abstractions up to B only guarantees consistency of the (unabstracted) executions if the run is shallow. Here, Corollary 4.5 will yield a shallow run, so that we may conclude consistency of the unabstracted executions.

6.1 Terminology

In our terminology, a thread is a pair (w, i), where w is a stack content and i is a context switch number. To argue about starvation, it is convenient to talk about how a thread evolves over time. By a *(thread) execution* we refer to the sequence of (pushdown and swap) instructions that belong to a single thread, from its creation via spawn until its termination. A thread execution can spawn new threads during each of its segments. We say that a thread execution e *produces* the multiset $\mathbf{m} \in \mathbb{M}[\Lambda]$, where $\Lambda = \Gamma \times \{0, \ldots, K\}$ if the following holds: For each $i \in \{0, \ldots, K\}$ and $\gamma \in \Gamma$, the thread execution e spawns $\mathbf{m}((\gamma, i))$ new threads with top of stack γ in segment i. In this case, we also call $\mathbf{m}$ the *production* of e.

According to Lemma 4.3, in order to decide STARV$[K]$, it suffices to decide whether in a given DCPS $\mathcal{A}$, there exists a *progressive* run that starves some thread (w, i). Therefore, we say that a run ρ is *starving* if it is progressive and starves some thread (w, i). Let us first formulate starvation in terms of thread executions. We observe that a progressive run ρ starves a thread (w, i) if and only if there are configurations $c_1, c_2, \ldots$ and executions $e_1, e_2, \ldots$ in ρ such that:

(1) For each $j = 1, 2, \ldots$, in configuration c_j, both e_j and e_{j+1} are in state (w, i),
(2) e_j is switched to in the step after c_j, and
(3) e_{j+1} is not switched to until c_{j+1}.

For the "if" direction, note that if a progressive run ρ starves (w, i), then (w, i) must be in the bag from some point on and whenever (w, i) becomes active, there are at least two instances of (w, i) in the bag. We choose $c_1, c_2, \ldots$ as exactly those configurations in ρ after which (w, i) becomes active. Moreover, e_j is the thread execution that is switched to after c_j. Furthermore, since in c_j, there must be another instance of (w, i) in the bag, there must be some execution e'_j whose state (w, i) is in the bag at c_j. However, since e_{j+1} will start from (w, i) in c_{j+1} and e'_j is in (w, i) at c_j, we may assume that $e_{j+1} = e'_j$. With this choice, we clearly satisfy (1)–(3) above.

For the "only if" direction, note that conditions (1)–(3) allow (w, i) to become active in between c_j and c_{j+1}. However, since e_{j+1} is not switched to between c_j and c_{j+1}, we know that any time (w, i) becomes active, there must be another instance of (w, i).

6.2 Consistency

Our first step in deciding starvation is to find a reformulation that does not explicitly mention the stack w. Instead, it states the existence of w as a consistency condition, which we will develop now.

Of course, it suffices to check whether a DCPS can starve some thread (w, i) when $i \in [1, K]$ is fixed. Therefore, from now on, we choose some $i \in [1, K]$ and want to decide whether there is a stack $w \in \Gamma^*$ such that the thread (w, i) can be starved by our DCPS.

First, a note on notation. In the following, we will abbreviate the set $\mathcal{T}(\mathcal{A}, K)$ of thread types with $\mathcal{T}$. We will work with families $(X_t)_{t\in\mathcal{T}}$ of subsets $X_t \subseteq X$ of some set X indexed by types $t \in \mathcal{T}$. We identify the set of such tuples indexed by $\mathcal{T}$ with $\mathbb{P}(X)^{\mathcal{T}}$. Sometimes, it is more natural to treat them as tuples $(X_1, \ldots, X_k)$ with $k = |\mathcal{T}|$. For simplicity, we will call both objects tuples.

For each type $t \in \mathcal{T}$, we consider the following set

$$S_t = \{(w, \mathbf{m}) \in \Gamma^* \times \mathbb{M}[\Lambda] \mid \text{there is an execution of type } t \text{ that produces } \mathbf{m} \text{ and reaches stack } w \text{ after segment } i\}$$

The set S_t encodes the following information: Is there a thread execution of type t that produces $\mathbf{m} \in \mathbb{M}[\Lambda]$ and at the same time arrives in w after i segments? The tuple $\mathfrak{S}_{\mathcal{A}} = (S_t)_{t\in\mathcal{T}}$ encodes this information for all types at once.

We will analyze $\mathfrak{S}_{\mathcal{A}}$ to show that if our decision procedure claims that there exists a starving run, then we can construct one. This construction will involve replacing one execution with another that (i) has the same type, (ii) arrives in w after i segments, and (iii) spawns more threads. Formally, the inserted execution must be larger w.r.t. the following order: For $\mathbf{m}, \mathbf{m}' \in \mathbb{M}[\Lambda]$, we have $\mathbf{m} \preceq_1 \mathbf{m}'$ if and only if $\mathbf{m} \preceq \mathbf{m}'$ and also $\mathrm{supp}(\mathbf{m}) = \mathrm{supp}(\mathbf{m}')$. Recall that $\mathrm{supp}(\mathbf{m}) = \{x \in \Lambda \mid \mathbf{m}(x) > 0\}$ is the support of $\mathbf{m} \in \mathbb{M}[\Lambda]$. Here, the condition $\mathrm{supp}(\mathbf{m}) = \mathrm{supp}(\mathbf{m}')$ makes sure that the replacement does not introduce thread spawns with new stack symbols, as this might destroy progressiveness of the run.

Let $S \subseteq \Gamma^* \times \mathbb{M}[\Lambda]$ be a set. For $w \in \Gamma^*$, we define

$$S{\downarrow_w} = \{\mathbf{m} \in \mathbb{M}[\Lambda] \mid \exists \mathbf{m}' \in \mathbb{M}[\Lambda] \colon \mathbf{m} \preceq_1 \mathbf{m}',\ (w, \mathbf{m}') \in S\}.$$

Observe that $\mathbf{m} \in S_t{\downarrow_w}$ expresses that there exists an execution of type t that visits w after segment i and produces a vector $\mathbf{m}' \succeq_1 \mathbf{m}$.

Our definition of consistency involves the tuple $\mathfrak{S}_{\mathcal{A}}$. However, since some technical proofs will be more natural in a slightly more abstract setting, we define consistency for a general tuple $\mathfrak{S} = (S_1, \ldots, S_k)$ of subsets $S_l \subseteq \Gamma^* \times \mathbb{M}[\Lambda]$. Hence, the following definitions should be understood with the case $\mathfrak{S} = \mathfrak{S}_{\mathcal{A}}$ in mind. Suppose we have a run with thread executions $e_1, e_2, \ldots$ and for each type t, let V_t be the set of productions of all executions in $\{e_1, e_2, \ldots\}$ that have type t. We want to formulate a condition expressing the existence of a stack w such that for any $t \in \mathcal{T}$ and any multiset $\mathbf{m} \in V_t$, there exists an execution of type t that visits w (after i context switches) and produces a multiset $\mathbf{m}' \succeq_1 \mathbf{m}$. This would allow us to replace each e_j by an execution e_j' that actually visits w: Since $\mathbf{m}' \succeq_1 \mathbf{m}$, we know that e_j' produces more threads of each stack symbol (and can thus still sustain the run), but also does not introduce new kinds of threads (because $\mathbf{m}'$ and $\mathbf{m}$ have the same support), so that progressiveness will not be affected by the replacement.

Let us make this formal. We say that a tuple $\mathfrak{B} = (V_1, \ldots, V_k)$ with $V_l \subseteq \mathbb{M}[\Lambda]$ is $\mathfrak{S}$-*consistent* if there exists a $w \in \Gamma^*$ with $V_l \subseteq S_l{\downarrow_w}$ for each $l \in [1, k]$. In this case, we call w an $\mathfrak{S}$-*consistency witness* for $\mathfrak{B}$. For words $w, w' \in \Gamma^*$, we write $w \leq_{\mathfrak{S}} w'$ if $S_l{\downarrow_w} \subseteq S_l{\downarrow_{w'}}$ for every $l \in [1, k]$.

6.3 Starvation in Terms of Consistency

This allows us to state the following reformulation of starvation, where w does not appear explicitly. A progressive run ρ is said to be *consistent* if there are configurations $c_1, c_2, \ldots$ and thread executions $e_1, e_2, \ldots$ that produce $\mathbf{m}_1, \mathbf{m}_2, \ldots$ and such that:

(1) For each $j = 1, 2, \ldots$, in configuration c_j, the executions e_j and e_{j+1} have completed i segments,
(2) e_j is switched to in the step after c_j,
(3) e_{j+1} is not switched to until c_{j+1}, and:
(4) Let $V_t = \{\mathbf{m}_j \mid j \in \mathbb{N}, \text{execution } e_j \text{ has type } t\}$. Then the tuple $\mathfrak{V} = (V_t)_{t\in\mathcal{T}}$ is $\mathfrak{S}_{\mathcal{A}}$-consistent.

Note that the consistency condition in (4) expresses that there exists a stack content w such that we could, instead of each e_j, perform a thread execution that actually visits w. It is thus straightforward to show:

Lemma 6.1. *A* DCPS *has a starving run if and only if it has a consistent run.*

6.4 Tracking Consistency

Our next step is to find some finite data that we can track about each of the produced vectors $\mathbf{m}_1, \mathbf{m}_2, \ldots$ such that this data determines whether the tuple $(V_t)_{t\in\mathcal{T}}$ is $\mathfrak{S}_{\mathcal{A}}$-consistent. We do this by abstracting vectors "up to a bound." Let $B \in \mathbb{N}$. We define the map $\alpha_B \colon \mathbb{M}[\Lambda] \to \mathbb{M}[\Lambda]$ by $\alpha_B(\mathbf{m}) = \mathbf{m}'$, where $\mathbf{m}'(x) = \min(\mathbf{m}(x), B)$ for $x \in \Lambda$. We naturally extend α_B to subsets of $\mathbb{M}[\Lambda]$ (point-wise) and to tuples of subsets of $\mathbb{M}[\Lambda]$ (component-wise). Note that for a tuple $\mathfrak{V} = (V_t)_{t\in\mathcal{T}}$ with $V_t \subseteq \mathbb{M}[\Lambda]$ for $t \in \mathcal{T}$, the tuple $\alpha_B(\mathfrak{V})$ belongs to the finite set $\mathbb{P}([0, B]^\Lambda)^\mathcal{T}$. The following theorem tells us that by abstracting w.r.t. some suitable B, we do not lose information about $\mathfrak{S}_{\mathcal{A}}$-consistency.

Theorem 6.2. *Given a* DCPS $\mathcal{A}$*, there is an effectively computable bound $B \in \mathbb{N}$ such that the following holds. If $\mathfrak{V} = (V_t)_{t\in\mathcal{T}}$ is a tuple of* ***finite*** *subsets $V_t \subseteq \mathbb{M}[\Lambda]$, then $\mathfrak{V}$ is $\mathfrak{S}_{\mathcal{A}}$-consistent if and only if $\alpha_B(\mathfrak{V})$ is $\mathfrak{S}_{\mathcal{A}}$-consistent.*

Roughly speaking, Theorem 6.2 allows us to check for the existence of a consistent run by checking whether there is one with an $\mathfrak{S}_{\mathcal{A}}$-consistent tuple $\alpha_B(\mathfrak{V})$. However, we may only conclude consistency of $\mathfrak{V}$ (and hence of the run) from consistency of $\alpha_B(\mathfrak{V})$ if $\mathfrak{V}$ is finite. To remedy this, we shall employ the fact that a DCPS with a progressive run also has a shallow progressive run (Corollary 4.5). We will show that if our algorithm detects a run ρ with consistent $\alpha_B(\mathfrak{V})$, then there also exists a run ρ' with finite $\mathfrak{V}$ such that $\alpha_B(\mathfrak{V})$ is consistent, meaning by Theorem 6.2, ρ' has to be consistent.

Moreover, given a tuple of finite subsets, we can decide $\mathfrak{S}_{\mathcal{A}}$-consistency:

Theorem 6.3. *Given a tuple $\mathfrak{V} = (V_t)_{t\in\mathcal{T}}$ of finite subsets $V_t \subseteq \mathbb{M}[\Lambda]$, it is decidable whether $\mathfrak{V}$ is $\mathfrak{S}_{\mathcal{A}}$-consistent.*

6.5 Deciding Starvation

We will prove Theorems 6.2 and 6.3 later in this section. Before we do that, let us show how they are used to decide starvation. First, we use Theorem 6.2 to compute $B \in \mathbb{N}$. Let us fix B for the decision procedure. Let $\mathfrak{u} \in \mathbb{P}([0, B]^\Lambda)^\mathcal{T}$ with $\mathfrak{u} = (U_t)_{t\in\mathcal{T}}$. We use a lower-case letter for this tuple to emphasize that it is of bounded size. An infinite progressive run ρ of $\mathcal{A}$ is said to be $(i, \mathfrak{u})$-*starving* if it contains configurations $c_1, c_2, \ldots$ and executions $e_1, e_2, \ldots$ that produce $\mathbf{m}_1, \mathbf{m}_2, \ldots$ such that:

(1) For each $j = 1, 2, \ldots$, in configuration c_j, the executions e_j and e_{j+1} have completed i segments,
(2) e_j is switched to in the step after c_j,
(3) e_{j+1} is not switched to until c_{j+1}, and:

(4) Let $V_t = \{\mathbf{m}_j \mid j \in \mathbb{N}, \text{execution } e_j \text{ has type } t\}$. Then $\alpha_B(V_t) \subseteq U_t$ for each $t \in \mathcal{T}$.

Now using the bound B from Theorem 6.2, we can show the following.

Lemma 6.4. *If $\mathcal{A}$ has a starving run, then it has an $(i, \mathfrak{u})$-starving run for some $i \in [1, K]$ and some $\mathfrak{S}_{\mathcal{A}}$-consistent $\mathfrak{u} \in \mathbb{P}([0, B]^{\Lambda})^{\mathcal{T}}$. Moreover, if $\mathcal{A}$ has a **shallow** $(i, \mathfrak{u})$-starving run for some $i \in [1, K]$ and some $\mathfrak{S}_{\mathcal{A}}$-consistent $\mathfrak{u} \in \mathbb{P}([0, B]^{\Lambda})^{\mathcal{T}}$, then it has a starving run.*

Here, we need to assume shallowness for the converse direction because we need finiteness of $\mathfrak{B}$ in the converse of Theorem 6.2.

Because of Lemma 6.4, we can proceed as follows to decide starvation. We first guess a tuple $\mathfrak{u} \in \mathbb{P}([0, B]^{\Lambda})^{\mathcal{T}}$ and check whether it is $\mathfrak{S}_{\mathcal{A}}$-consistent using Theorem 6.3. Then, we construct a DCPS $\mathcal{A}_{(i,\mathfrak{u})}$ such that $\mathcal{A}_{(i,\mathfrak{u})}$ has a progressive run if $\mathcal{A}$ has an $(i, \mathfrak{u})$-starving run. Moreover, we use the fact every DCPS that has a progressive infinite run also has a shallow infinite run (Corollary 4.5). This will allow us to turn a progressive run of $\mathcal{A}_{(i,\mathfrak{u})}$ into a shallow $(i, \mathfrak{u})$-starving run of $\mathcal{A}$, which must be starving by Lemma 6.4. Let us now see how to construct $\mathcal{A}_{(i,\mathfrak{u})}$.

6.6 **Freezing** DCPS

For constructing $\mathcal{A}_{(i,\mathfrak{u})}$, it is convenient to have a simple locking mechanism available, which we call "freezing." It will be easy to see that this can be implemented in DCPS. In a freezing DCPS, there is one distinguished "frozen" thread in each configuration. It cannot be resumed using the ordinary resume rules. It can only be resumed using an unfreeze operation, which at the same time freezes another thread. We use this to make sure that the e_{j+1} stays inactive between c_j and c_{j+1}.

Syntactically, a *freezing* DCPS is a tuple $\mathcal{A} = (G, \Gamma, \Delta, g_0, \gamma_0, \gamma_f)$, where $(G, \Gamma, \Delta, g_0, \gamma_0)$ is a DCPS, except that the rules Δ also contain a set Δ_{u} of *unfreezing rules* of the form $g \mapsto g' \lhd \gamma \circledast \gamma'$ and γ_f is the initial frozen thread with a single stack symbol. The unfreezing rules allow the DCPS to unfreeze and resume a thread with top of stack γ, while also freezing a thread with top of stack γ'. A *configuration* is a tuple in $G \times (\Gamma^* \times \mathbb{N} \cup \{\#\}) \times \mathbb{M}[\hat{\Gamma}^* \times \mathbb{N}]$, where $\hat{\Gamma} = \Gamma \cup \Gamma^{\circledast}$ and $\Gamma^{\circledast} = \{\gamma^{\circledast} \mid \gamma \in \Gamma\}$. A thread is *frozen* if its top-of-stack belongs to $\Gamma^{\circledast}$. It will be clear from the steps that in each reachable configuration, there is exactly one frozen thread.

A freezing DCPS has the same steps as those of the corresponding DCPS. In particular, those apply only to top-of-stack symbols in Γ. In addition, there is one more rule:

Unfreeze

$$\frac{g \mapsto g' \lhd \gamma \circledast \gamma'}{\langle g, \#, \mathbf{m} + [\![\gamma^{\circledast} w, l]\!] + [\![\gamma' w', j]\!]\rangle \mapsto \langle g', (\gamma w, l), \mathbf{m} + [\![\gamma'^{\circledast} w', j]\!]\rangle}$$

Hence, the frozen thread $(\gamma^{\circledast} w, l)$ is unfrozen and resumes, while the thread $(\gamma' w', j)$ becomes the new frozen thread. Moreover, the initial configuration is $\langle g_0, \#, [\![(\gamma_0, 0)]\!] + [\![(\gamma_f^{\circledast}, 0)]\!]\rangle$.

Given these additional steps, progressive termination is defined as for DCPS. (In particular, the progressiveness condition also applies to frozen threads.)

Lemma 6.5. *Given a freezing* DCPS *$\mathcal{A}$, it is decidable whether $\mathcal{A}$ has a progressive run. Moreover, if $\mathcal{A}$ has a progressive run, then it has a shallow progressive run.*

Lemma 6.5 can be shown using a straightforward reduction to progressive termination of ordinary DCPS. The freezing is realized by introducing stack symbols $\Gamma^{\circledast} = \{\gamma^{\circledast} \mid \gamma \in \Gamma\}$. An unfreeze rule $g \mapsto g' \lhd \gamma \circledast \gamma'$ for a thread $(\gamma^{\circledast} w, l)$ is then simulated by a simple locking mechanism using a bounded number of context switches: It turns a thread with stack $\gamma' w'$ into one with stack $\gamma'^{\circledast} w'$ (using context switches) and then resumes $(\gamma^{\circledast} w, j)$, where initially, $\gamma^{\circledast}$ is replaced with γ. Other than that, for threads with top of stack in $\Gamma^{\circledast}$, there are no resume rules. Since each thread in a

freeze DCPS can only be frozen and unfrozen at most K times, the constructed DCPS uses at most $2K + 1$ context-switches to simulate a run of the freeze DCPS.

6.7 Reduction to Progressive Runs in Freezing DCPS

We now reduce starvation to progressive runs in freezing DCPS. We first guess a pair $(i, \mathfrak{u})$ with $i \in [1, K]$ and a $\mathfrak{S}_{\mathcal{A}}$-consistent $\mathfrak{u} \in \mathbb{P}([1, B]^{\Lambda})^{\mathcal{T}}$, $\mathfrak{u} = (U_t)_{t \in \mathcal{T}}$, and construct a freezing DCPS $\mathcal{A}_{(i,\mathfrak{u})}$ so that $\mathcal{A}$ has an $(i, \mathfrak{u})$-starving run if and only if $\mathcal{A}_{(i,\mathfrak{u})}$ has a progressive run. Moreover, if $\mathcal{A}_{(i,\mathfrak{u})}$ has a progressive run, then $\mathcal{A}$ even has a shallow $(i, \mathfrak{u})$-starving run. Therefore, $\mathcal{A}$ has a starving run if and only if for some choice of $(i, \mathfrak{u})$, $\mathcal{A}_{(i,\mathfrak{u})}$ has a progressive run.

Intuitively, we do this by tracking for each thread execution the multiset $\alpha_B(\mathbf{m})$, where $\mathbf{m}$ is its production. Using frozen threads, we make sure that every progressive run in $\mathcal{A}$ contains executions $e_1, e_2, \ldots$ to witness $(i, \mathfrak{u})$-starvation. To verify the $(i, \mathfrak{u})$-starvation, we also track each thread's type and current context-switch number. Hence, we store a tuple $(t, j, \bar{\mathbf{m}}, \bar{\mathbf{n}})$, where (i) t is the type, (ii) j is the current context-switch number, (iii) $\bar{\mathbf{m}}$ is the guess for $\alpha_B(\mathbf{m})$, where $\mathbf{m} \in \mathbb{M}[\Lambda]$ is the entire production of the execution, and (iv) $\bar{\mathbf{n}}$ is $\alpha_B(\mathbf{n})$, where $\mathbf{n} \in \mathbb{M}[\Lambda]$ is the multiset spawned so far.

While a thread is inactive, the extra information is stored on the top of the stack, resulting in stack symbols $(\gamma, t, j, \bar{\mathbf{m}}, \bar{\mathbf{n}})$. In particular, when we spawn a new thread, we immediately guess its type t and the abstraction $\bar{\mathbf{m}}$, and we set $j = 0$ and $\bar{\mathbf{n}} = \emptyset$. The freezing and unfreezing works as follows. Initially, we have the frozen thread $\gamma_\dagger$ (where $\gamma_\dagger$ is a fresh stack symbol). To unfreeze it, we have to freeze a thread of some type t where $\bar{\mathbf{m}}$ belongs to U_t (recall that this is a component of $\mathfrak{u}$):

$$g \mapsto g' \lhd \gamma_\dagger \mathbin{❄} (\gamma, t, i, \bar{\mathbf{m}}, \bar{\mathbf{n}})$$

for every $g, g' \in G$, $t \in \mathcal{T}$, $\bar{\mathbf{m}} \in U_t$. To unfreeze (and thus resume) a thread with top of stack $(\gamma, t, i, \bar{\mathbf{m}}, \bar{\mathbf{n}})$, we have to freeze a thread $(\gamma', t', i, \bar{\mathbf{m}}', \bar{\mathbf{n}}')$ with $\bar{\mathbf{m}}' \in U_{t'}$. Unfreezing requires context-switch number i, because the executions $e_1, e_2, \ldots$ must be in segment i in $c_1, c_2, \ldots$:

$$g \mapsto \widehat{g'} \lhd (\gamma, t, i, \bar{\mathbf{m}}, \bar{\mathbf{n}}) \mathbin{❄} (\gamma', t', i, \bar{\mathbf{m}}', \bar{\mathbf{n}}')$$

for each resume rule $g \mapsto g' \lhd \gamma, t, \bar{\mathbf{m}}$, and $\bar{\mathbf{n}}$, provided that g is the state specified in t to enter from in the ith segment. Here, $\widehat{g'}$ is a decorated version of g', in which the thread can only transfer the extra information related to $t, i, \bar{\mathbf{m}}, \bar{\mathbf{n}}$ back to the global state. Symmetrically, when interrupting a thread that information is transferred back to the stack and the segment counter j is incremented. To resume an ordinary (i.e. unfrozen) inactive thread, we have a resume rule $g \mapsto g' \lhd (\gamma, t, j, \bar{\mathbf{m}}, \bar{\mathbf{n}})$ for each resume rule $g \mapsto g' \lhd \gamma$ and each t, j, $\bar{\mathbf{m}}$, and $\bar{\mathbf{n}}$ — if g is specified as the entering global state for segment j in t. While a thread is active, it keeps $\bar{\mathbf{n}}$ up to date by recording all spawns (and reducing via α_B). Finally, when a thread terminates, it checks that the components $\bar{\mathbf{m}}$ and $\bar{\mathbf{n}}$ agree.

It is clear from the construction that $\mathcal{A}$ has a $(i, \mathfrak{u})$-starving run if and only if $\mathcal{A}_{(i,\mathfrak{u})}$ has a progressive run. Moreover, Lemma 6.5 tells us that if $\mathcal{A}_{(i,\mathfrak{u})}$ has a progressive run, then it has a shallow progressive run. This shallow progressive run clearly yields a shallow $(i, \mathfrak{u})$-starving run of $\mathcal{A}$. According to Lemma 6.4, this implies that $\mathcal{A}$ has a starving run. This establishes the following lemma, which implies that starvation is decidable for DCPS.

Lemma 6.6. *$\mathcal{A}$ has a starving run if and only if for some $i \in [1, K]$ and some $\mathfrak{S}_{\mathcal{A}}$-consistent $\mathfrak{u} \in \mathbb{P}([0, B]^{\Lambda})^{\mathcal{T}}$, the freezing* DCPS *$\mathcal{A}_{(i,\mathfrak{u})}$ has a progressive run.*

6.8 Proving Theorems 6.2 and 6.3

It remains to prove Theorems 6.2 and 6.3. We will use a structural description of the sets S_t (Lemma 6.7), which requires some terminology. An *automaton over* $\Gamma^* \times \mathbb{M}[\Lambda]$ is a tuple $\mathcal{M} = (Q, E, q_0, q_f)$, where Q is a finite set of *states*, $E \subseteq Q \times \Gamma^* \times \mathbb{M}[\Lambda] \times Q$ is a finite set of

edges, $q_0 \in Q$ is its *initial state*, and $q_f \in Q$ is its *final state*. We write $p \xrightarrow{u|\mathbf{m}} q$ if there is a sequence $(p_0, u_1, \mathbf{m}_1, p_1), (p_1, u_2, \mathbf{m}_2, p_2), \ldots, (p_{n-1}, u_n, \mathbf{m}_n, p_n)$ of edges in $\mathcal{M}$ with $p = p_0, q = p_n$, $u = u_1 \cdots u_n$, and $\mathbf{m} = \mathbf{m}_1 + \cdots + \mathbf{m}_n$. The set *accepted by* $\mathcal{M}$ is the set of all $(w, \mathbf{m}) \in \Gamma^* \times \mathbb{M}[\Lambda]$ with $q_0 \xrightarrow{w|\mathbf{m}} q_f$. A subset of $\Gamma^* \times \mathbb{M}[\Lambda]$ is *rational* if it is accepted by some automaton over $\Gamma^* \times \mathbb{M}[\Lambda]$.

Lemma 6.7. *For every $t \in \mathcal{T}$, the set S_t is effectively rational.*

This can be deduced from [Zetzsche 2013, Lemma 6.2]. Since the latter would require introducing a lot of machinery, we include a direct proof in the full version [Baumann et al. 2020b]. Both proofs are slight extensions of Büchi's proof of regularity of the set of reachable stacks in a pushdown automaton [Büchi 1964, Theorem 1]. The only significant difference is the following: While [Büchi 1964] essentially introduces shortcut edges for runs that go from one stack w back to w, we glue in a finite automaton that produces the same output over Λ as such runs. This is possible since the set of the resulting multisets is always semi-linear by Parikh's theorem [Parikh 1966, Theorem 2].

Because of Lemma 6.7, the following immediately implies Theorem 6.3:

Lemma 6.8. *Given a tuple $\mathfrak{S} = (S_1, \ldots, S_k)$ of rational subsets $S_j \subseteq \Gamma^* \times \mathbb{M}[\Lambda]$ and a tuple $\mathfrak{u} = (U_1, \ldots, U_k)$ of finite subsets $U_j \subseteq \mathbb{M}[\Lambda]$, it is decidable whether $\mathfrak{u}$ is $\mathfrak{S}$-consistent.*

Proof. Since S_j is rational, for each $\mathbf{m} \in \mathbb{M}[\Lambda]$ and $j \in [1, k]$, we can compute a finite automaton for the language $T_{j,\mathbf{m}} = \{w \in \Gamma^* \mid \mathbf{m} \in S_j{\downarrow_w}\}$. Then $\mathfrak{u}$ is $\mathfrak{S}$-consistent if and only if the intersection $\bigcap_{j \in [1,k]} \bigcap_{\mathbf{m} \in U_j} T_{j,\mathbf{m}}$ of regular languages is non-empty, which is clearly decidable. ∎

Moreover, because of Lemma 6.7, Theorem 6.2 is a direct consequence of the following.

Proposition 6.9. *Given rational subsets $S_1, \ldots, S_k \subseteq \Gamma^* \times \mathbb{M}[\Lambda]$, we can compute a bound B such that for the tuple $\mathfrak{S} = (S_1, \ldots, S_k)$, the following holds: If $\mathfrak{V} = (V_1, \ldots, V_k)$ is a tuple of **finite** subsets $V_j \subseteq \mathbb{M}[\Lambda]$, then $\mathfrak{V}$ is $\mathfrak{S}$-consistent if and only if $\alpha_B(\mathfrak{V})$ is $\mathfrak{S}$-consistent.*

Thus, it remains to prove Proposition 6.9, which is the purpose of the rest of this section. Note that in Proposition 6.9, the requirement that the V_j be *finite* is crucial. For example, suppose $k = 1$ and $S = \{(\mathsf{a}^n, n \cdot [\![\mathsf{b}]\!]) \mid n \in \mathbb{N}\}$ and $\mathfrak{S} = (S)$. Then a set $V \subseteq \mathbb{M}[\{\mathsf{b}\}]$ is $\mathfrak{S}$-consistent if and only if V is finite. Hence, there is no bound B such that $\alpha_B(V)$ reflects $\mathfrak{S}$-consistency of any V. For Proposition 6.9, we use Ramsey's theorem (see Section 5.3) to prove the following pumping lemma.

Lemma 6.10. *Given a tuple $\mathfrak{S} = (S_1, \ldots, S_k)$ of rational subsets $S_j \subseteq \Gamma^* \times \mathbb{M}[\Lambda]$, we can compute a bound M such that the following holds. In a word w with M marked positions, we can pick two marked positions so that for the resulting decomposition $w = xyz$, we have $xyz \leq_{\mathfrak{S}} xy^\ell z$ for every $\ell \geq 1$.*

Proof. Let $\mathcal{M}_j$ be an automaton for S_j with state set Q_j for $j \in [1, k]$. We may assume that the sets Q_j are pairwise disjoint and we define $Q = \bigcup_{j=1}^k Q_j$ and $n = |Q|$. To each $u \in \Gamma^*$, we assign a subset $\kappa(u) \subseteq Q \times \mathbb{P}(\Lambda)$, where $(q, \Theta) \in Q_j \times \mathbb{P}(\Lambda)$ belongs to $\kappa(u)$ if there is a cycle in $\mathcal{M}_j$ that starts (and ends) in q, reads u, and reads a multiset with support Θ. Hence, for $q \in Q_j$ and $\Theta \subseteq \Lambda$, we have $(q, \Theta) \in \kappa(u)$ if and only if $q \xrightarrow{u|\mathbf{m}} q$ in $\mathcal{M}_j$ for some $\mathbf{m} \in \mathbb{M}[\Lambda]$ with $\mathrm{supp}(\mathbf{m}) = \Theta$.

We specify M later. Suppose M positions $s_1, \ldots, s_M$ are marked in w. We build a colored graph on M vertices and we label the edge from j to j' by the set $\kappa(u)$, where u is the infix of w between s_j and $s_{j'}$. Hence, the graph is r-colored, where $r = 2^{|Q| \cdot 2^{|\Lambda|}}$ is the number of subsets of $Q \times \mathbb{P}(\Lambda)$. We now apply Ramsey's theorem. We compute M so that $M \geq R(r; n+1)$, e.g. $M = r^{r(n-1)+1}$. Then our graph must contain a monochromatic subset of size $n+1$. Let $t_1, \ldots, t_{n+1}$ be the corresponding positions in w. Moreover, let $w = xy_1 \cdots y_n z$ be the decomposition of w such that y_j is the infix between t_j and t_{j+1}. We claim that with $y = y_1 \cdots y_n$, we have indeed $xyz \leq_{\mathfrak{S}} xy^\ell z$ for every $\ell \geq 1$.

Consider a word $xy^\ell z$ and some multiset $\mathbf{m} \in S_j{\downarrow_{xyz}}$. We have to show that $\mathbf{m} \in S_j{\downarrow_{xy^\ell z}}$. Since $\mathbf{m} \in S_j{\downarrow_{xyz}}$, there is a run of $\mathcal{M}_j$ reading $(xyz, \mathbf{m}')$ for some $\mathbf{m}' \geq_1 \mathbf{m}$. Since $\mathcal{M}_j$ has $\leq n$ states, some state must repeat at two borders of the decomposition $y = y_1 \cdots y_n$. Suppose our run reads $(y_f \cdots y_g, \bar{\mathbf{m}})$ on a cycle on $q \in Q_j$ for some $\bar{\mathbf{m}}$ in $\mathcal{M}_j$. By monochromaticity, we know that $\kappa(y_f \cdots y_g) = \kappa(y_h)$ for every $h \in [1, n]$. Observe that we can write

$$xy^\ell z = xy_1 \cdots y_{f-1}(y_f \cdots y_n y_1 \cdots y_{f-1})^{\ell-1} y_f \cdots y_n z. \tag{1}$$

For every $h \in [1, n]$, we have $\kappa(y_f \cdots y_g) = \kappa(y_h)$, and hence $q \xrightarrow{y_h \mid \mathbf{m}_h} q$ in $\mathcal{M}_j$ for some $\mathbf{m}_h \in \mathbb{M}[\Lambda]$ with $\mathrm{supp}(\mathbf{m}_h) = \mathrm{supp}(\bar{\mathbf{m}})$. Then, in particular, $\mathrm{supp}(\mathbf{m}_h) \subseteq \mathrm{supp}(\mathbf{m}') = \mathrm{supp}(\mathbf{m})$. Therefore, Eq. (1) shows that $\mathcal{M}_j$ accepts $(xy^\ell z, \mathbf{m}' + (\ell - 1)\sum_{h=1}^n \mathbf{m}_h)$. Since we now have $\mathbf{m} \preceq_1 \mathbf{m}' + (\ell - 1)\sum_{h=1}^n \mathbf{m}_h$, this implies $\mathbf{m} \in S{\downarrow_{xy^\ell z}}$. ∎

Using Lemma 6.10, we can obtain the final ingredient of Proposition 6.9:

Lemma 6.11. *Given a tuple $\mathfrak{S} = (S_1, \ldots, S_k)$ of rational subsets $S_j \subseteq \Gamma^* \times \mathbb{M}[\Lambda]$, we can compute a bound B such that the following holds. Let $\mathfrak{V} = (V_1, \ldots, V_k)$ be a $\mathfrak{S}$-consistent tuple and suppose $\alpha_B(\mathbf{m}) \in V_j$. Then adding $\mathbf{m}$ to V_j preserves $\mathfrak{S}$-consistency.*

The idea is the following. Let $\mathfrak{V}' = (V_1', \ldots, V_k')$ be obtained from $\mathfrak{V}$ by adding $\mathbf{m}$ to V_j. Let us say that $w \in \Gamma^*$ *covers* some $\mathbf{n} \in V_j'$ if and only if $\mathbf{n} \in S_j{\downarrow_w}$. Since $\mathfrak{V}$ is $\mathfrak{S}$-consistent, there is a $w \in \Gamma^*$ that covers all elements of $\mathfrak{V}$. Then w covers $\alpha_B(\mathbf{m})$. Moreover, $\mathbf{m}$ agrees with $\alpha_B(\mathbf{m})$ on all coordinates where $\mathbf{m}$ is $< B$. We now have to construct a w' that covers $\mathbf{m}$ in the remaining coordinates. A simple pumping argument for each coordinate of $\mathbf{m}$ over an automaton for S_j (say, with B larger than the number of states) would yield a word w' that even covers $\mathbf{m}$. However, this might destroy coverage of all the other multisets in $\mathfrak{V}$. Therefore, we use Lemma 6.10. It allows us to choose B high enough so that pumping to $w' = xy^\ell z$ covers $\mathbf{m}$, but also guarantees $w \leq_{\mathfrak{S}} w'$. The latter implies that going from w to w' does not lose any coverage.

Finally, Proposition 6.9 follows from Lemma 6.11 by induction.

7 CONCLUSION

We have shown decidability of verifying liveness for DCPS in the context-bounded case. Our results imply that fair termination for DCPS is Π_1^0-complete when each thread is restricted to context switch a finite number of times. Our result extends to liveness properties that can be expressed as a Büchi condition. We can reduce to fair non-termination by simply adding the states of a Büchi automaton to the global states via a product construction. From there, the Büchi acceptance condition can be simulated by using a special thread that forces a visit to a final state when scheduled, and then reposts itself before terminating. Scheduling this thread fairly along an infinite execution thus results in infinitely many visits to final states.

While we have focused on termination- and liveness-related questions, our techniques also imply further decidability results on commonly studied decision questions for concurrent programs. A run of a DCPS is *bounded* if there is a bound $B \in \mathbb{N}$ so that the number of pending threads in every configuration along the run is at most B. The K-bounded boundedness problem asks if every K-context bounded run is bounded. Since boundedness is preserved under downward closures, our techniques for non-termination can be modified to show the problem is also 2EXPSPACE-complete.

The K-bounded *configuration reachability* problem for DCPS asks if a given configuration is reachable. Our reductions from DCPS to VASSB, and the decidability of reachability for VASSB, imply K-bounded configuration reachability is decidable for DCPS.

Thus, combined with previous results on safety verification [Atig et al. 2011] and the case $K = 0$ [Ganty and Majumdar 2012], our paper closes the decidability frontier for all commonly studied K-bounded verification problems for all $K \geq 0$.

ACKNOWLEDGMENTS

This research was sponsored in part by the Deutsche Forschungsgemeinschaft project 389792660 TRR 248–CPEC and by the European Research Council under the Grant Agreement 610150 (http://www.impact-erc.eu/) (ERC Synergy Grant ImPACT).

REFERENCES

Parosh Aziz Abdulla, Kārlis Čerāns, Bengt Jonsson, and Yih-Kuen Tsay. 1996. General decidability theorems for infinite-state systems. In *Proceedings of the Eleventh Annual Symposium on Logic in Computer Science*. IEEE Computer Society Press, 313–321.

Krzysztof R. Apt and Ernst-Rüdiger Olderog. 1991. *Verification of Sequential and Concurrent Programs*. Springer-Verlag.

Mohamed Faouzi Atig, Ahmed Bouajjani, Michael Emmi, and Akash Lal. 2012a. Detecting Fair Non-termination in Multithreaded Programs. In *Proceedings of CAV 2012*. 210–226. https://doi.org/10.1007/978-3-642-31424-7_19

Mohamed Faouzi Atig, Ahmed Bouajjani, K. Narayan Kumar, and Prakash Saivasan. 2012b. Linear-Time Model-Checking for Multithreaded Programs under Scope-Bounding. In *Automated Technology for Verification and Analysis - 10th International Symposium, ATVA 2012, Thiruvananthapuram, India, October 3-6, 2012. Proceedings (Lecture Notes in Computer Science, Vol. 7561)*. Springer, 152–166.

Mohamed Faouzi Atig, Ahmed Bouajjani, K. Narayan Kumar, and Prakash Saivasan. 2017. Parity Games on Bounded Phase Multi-pushdown Systems. In *Networked Systems - 5th International Conference, NETYS 2017, Marrakech, Morocco, May 17-19, 2017, Proceedings (Lecture Notes in Computer Science, Vol. 10299)*. Springer, 272–287.

Mohamed Faouzi Atig, Ahmed Bouajjani, and Shaz Qadeer. 2009. Context-Bounded Analysis for Concurrent Programs with Dynamic Creation of Threads. In *Proceedings of TACAS 2009*. 107–123.

Mohamed Faouzi Atig, Ahmed Bouajjani, and Shaz Qadeer. 2011. Context-Bounded Analysis For Concurrent Programs With Dynamic Creation of Threads. *Log. Methods Comput. Sci.* 7, 4 (2011). https://doi.org/10.2168/LMCS-7(4:4)2011

Georg Bachmeier, Michael Luttenberger, and Maximilian Schlund. 2015. Finite Automata for the Sub- and Superword Closure of CFLs: Descriptional and Computational Complexity. In *9th International Conference on Language and Automata Theory and Applications, LATA 2015, Nice, France, March 2-6, 2015, Proceedings*. Springer, 473–485.

Pascal Baumann, Rupak Majumdar, Ramanathan S. Thinniyam, and Georg Zetzsche. 2020a. 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) (LIPIcs, Vol. 168)*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 111:1–111:16.

Pascal Baumann, Rupak Majumdar, Ramanathan S. Thinniyam, and Georg Zetzsche. 2020b. Context-Bounded Verification of Liveness Properties for Multithreaded Shared-Memory Programs. arXiv:2011.04581

Julius Richard Büchi. 1964. Regular canonical systems. *Archiv für mathematische Logik und Grundlagenforschung* 6, 3-4 (1964), 91–111.

Heino Carstensen. 1987. Decidability questions for fairness in Petri nets. In *Proceedings of STACS 1987*. Springer, 396–407.

Byron Cook, Andreas Podelski, and Andrey Rybalchenko. 2007. Proving thread termination. In *Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation, San Diego, California, USA, June 10-13, 2007*. ACM, 320–330.

Byron Cook, Andreas Podelski, and Andrey Rybalchenko. 2011. Proving program termination. *Commun. ACM* 54, 5 (2011), 88–98. https://doi.org/10.1145/1941487.1941509

Bruno Courcelle. 1991. On construction obstruction sets of words. *EATCS* 44 (1991), 178–185.

Wojciech Czerwiński, Slawomir Lasota, Ranko Lazic, Jérôme Leroux, and Filip Mazowiecki. 2019. The reachability problem for Petri nets is not elementary. In *Proceedings of STOC 2019*. 24–33.

Antoine Durand-Gasselin, Javier Esparza, Pierre Ganty, and Rupak Majumdar. 2017. Model checking parameterized asynchronous shared-memory systems. *Formal Methods Syst. Des.* 50, 2-3 (2017), 140–167. https://doi.org/10.1007/s10703-016-0258-3

Paul Erdős and Richard Rado. 1952. Combinatorial Theorems on Classifications of Subsets of a Given Set. *Proceedings of the London Mathematical Society* s3-2, 1 (01 1952), 417–439. https://doi.org/10.1112/plms/s3-2.1.417

Azadeh Farzan, Zachary Kincaid, and Andreas Podelski. 2016. Proving Liveness of Parameterized Programs. In *Proceedings of the 31st Annual ACM/IEEE Symposium on Logic in Computer Science, LICS '16, New York, NY, USA, July 5-8, 2016*. ACM, 185–196.

Alain Finkel and Philippe Schnoebelen. 2001. Well-structured transition systems everywhere! *Theor. Comput. Sci.* 256, 1-2 (2001), 63–92.

Marie Fortin, Anca Muscholl, and Igor Walukiewicz. 2017. Model-Checking Linear-Time Properties of Parametrized Asynchronous Shared-Memory Pushdown Systems. In *Computer Aided Verification - 29th International Conference, CAV 2017, Heidelberg, Germany, July 24-28, 2017, Proceedings, Part II (Lecture Notes in Computer Science, Vol. 10427)*. Springer, 155–175.

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

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

David Harel. 1986. Effective transformations on infinite trees, with applications to high undecidability, dominoes, and fairness. *J. ACM* 33 (1986), 224–248.

Rodney R Howell, Louis E Rosier, and Hsu-Chun Yen. 1991. A taxonomy of fairness and temporal logic problems for Petri nets. *Theoretical Computer Science* 82, 2 (1991), 341–372.

Petr Jančar. 1990. Decidability of a temporal logic problem for Petri nets. *Theoretical Computer Science* 74, 1 (1990), 71–93.

Vineet Kahlon. 2008. Parameterization as Abstraction: A Tractable Approach to the Dataflow Analysis of Concurrent Programs. IEEE Computer Society, 181–192.

Sambasiva Rao Kosaraju. 1982. Decidability of Reachability in Vector Addition Systems (Preliminary Version). In *STOC '82: Proc. of 14th ACM symp. on Theory of Computing*. ACM, 267–281.

Bernhard Kragl, Constantin Enea, Thomas A. Henzinger, Suha Orhun Mutluergil, and Shaz Qadeer. 2020. Inductive sequentialization of asynchronous programs. In *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2020, London, UK, June 15-20, 2020*. ACM, 227–242.

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

Akash Lal, Tayssir Touili, Nicholas Kidd, and Thomas W. Reps. 2008. Interprocedural Analysis of Concurrent Programs Under a Context Bound. In *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 (Lecture Notes in Computer Science, Vol. 4963)*. Springer, 282–298.

Jérôme Leroux, Grégoire Sutre, and Patrick Totzke. 2015. On the coverability problem for pushdown vector addition systems in one dimension. In *ICALP 2015*, Vol. 9135. 324–336. https://doi.org/10.1007/978-3-662-47666-6_26

Irina A. Lomazova and Philippe Schnoebelen. 1999. Some Decidability Results for Nested Petri Nets. In *Perspectives of System Informatics, Third International Andrei Ershov Memorial Conference, PSI'99, Akademgorodok, Novosibirsk, Russia, July 6-9, 1999, Proceedings (Lecture Notes in Computer Science, Vol. 1755)*. Springer, 208–220.

Ernst W. Mayr. 1981. An Algorithm for the General Petri Net Reachability Problem. In *Proceedings of STOC 1981*. 238–246.

Anca Muscholl, Helmut Seidl, and Igor Walukiewicz. 2017. Reachability for Dynamic Parametric Processes. In *Verification, Model Checking, and Abstract Interpretation - 18th International Conference, VMCAI 2017, Paris, France, January 15-17, 2017, Proceedings (Lecture Notes in Computer Science, Vol. 10145)*. Springer, 424–441.

Madanlal Musuvathi and Shaz Qadeer. 2007. 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*. ACM, 446–455.

Oded Padon, Jochen Hoenicke, Giuliano Losa, Andreas Podelski, Mooly Sagiv, and Sharon Shoham. 2018. Reducing liveness to safety in first-order logic. *Proc. ACM Program. Lang.* 2, POPL (2018), 26:1–26:33.

Rohit J. Parikh. 1966. On Context-Free Languages. *J. ACM* 13, 4 (1966), 570–581.

Shaz Qadeer and Jakob Rehof. 2005. Context-Bounded Model Checking of Concurrent Software. In *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 (Lecture Notes in Computer Science, Vol. 3440)*. Springer, 93–107.

Charles Rackoff. 1978. The covering and boundedness problems for vector addition systems. *Theoretical Computer Science* 6, 2 (1978), 223–231.

Ganesan Ramalingam. 2000. Context-sensitive synchronization-sensitive analysis is undecidable. *ACM TOPLAS* 22(2) (2000), 416–430.

Frank Plumpton Ramsey. 1930. On a Problem of Formal Logic. *Proceedings of the London Mathematical Society* s2-30, 1 (1930), 264–286. https://doi.org/10.1112/plms/s2-30.1.264

Salvatore La Torre, Margherita Napoli, and Gennaro Parlato. 2016. Scope-Bounded Pushdown Languages. *Int. J. Found. Comput. Sci.* 27, 2 (2016), 215–234. https://doi.org/10.1142/S0129054116400074

Moshe Y. Vardi. 1991. Verification of concurrent programs—the automata-theoretic framework. *Annals of Pure and Applied Logic* 51 (1991), 79–98.

Kumar Neeraj Verma and Jean Goubault-Larrecq. 2005. Karp-Miller Trees for a Branching Extension of VASS. *Discrete Mathematics & Theoretical Computer Science* Vol. 7 (2005).

Georg Zetzsche. 2013. Silent Transitions in Automata with Storage. (2013). arXiv:1302.3798 Full version of an article in Proceedings of ICALP 2013.