# 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.

## 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 $\Pi_1^1$-complete; in fact, the problem is already $\Pi_1^1$-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 $\Sigma_1^0$-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. 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 $\Pi_1^0$-complete when each thread is context-switched a finite number of times. (Note that this does not contradict the $\Pi_1^1$-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

---

[1]Recall that the class $\Pi_1^1$ 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.

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. [2020]. 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.

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 multi-threaded 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. [n. d.]].

**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

---

[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 [Czerwinski et al. 2019].

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 $\Sigma_1^1$-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 $a \in S\setminus\{a_1, a_3\}$. 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 $\leq$ on $\mathbb{M}[S]$ as follows: $\mathbf{m} \leq \mathbf{m}'$ 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 $\mathbf{m}' \leq \mathbf{m}$ 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 $\Sigma$, a finite alphabet of *stack symbols* $\Gamma$, 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 $(q_0, \gamma_0)$. The set of *final configurations* is $Q_F \times \Gamma^*$. For each $a \in \Sigma \cup \{\varepsilon\}$, the relation $\xRightarrow{a}$ on configurations of $\mathcal{P}$ is defined as follows: $(q, \gamma w) \xRightarrow{a} (q', w'w)$ 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 $c \xRightarrow{a} c'$ for some $a$. Furthermore, we write $c \xRightarrow{u}^* c'$ for some $u \in \Sigma^*$ if there is a sequence of configurations $c_0$ to $c_n$ with

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

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 $\text{Parikh}(u) : \Sigma \to \mathbb{N}$ such that, for every $a \in \Sigma$, we have $\text{Parikh}(u)(a) = |u|_a$, where $|u|_a$ denotes the number of occurrences of $a$ in $u$. We extend the definition to the Parikh image of a language $L \subseteq \Sigma^*$: $\text{Parikh}(L) = \{\text{Parikh}(u) \mid u \in L\}$. 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 $\text{Parikh}(L)$ is semi-linear. A representation of the semi-linear set $\text{Parikh}(L)$ 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* $\Gamma$, an *initial state* $g_0 \in G$, an *initial stack symbol* $\gamma_0 \in \Gamma$, and a finite set of *transition rules* $\Delta$. The set of transition rules $\Delta$ is partitioned into four kinds of rules: *creation rules* $\Delta_c$, *interruption rules* $\Delta_i$, *resumption rules* $\Delta_r$, and *termination rules* $\Delta_t$. Elements of $\Delta_c$ 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 $\gamma'$. Elements

of $\Delta_i$ 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 $\Delta_r$ have the form $g \mapsto g' \triangleleft \gamma$, where $g, g' \in G$ and $\gamma \in \Gamma$. Elements of $\Delta_t$ 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 $\Gamma$ and uses the transition rules in $\Delta_c$ to manipulate the global state and its stack. It can additionally spawn new threads using rules of type (2) in $\Delta_c$. 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 $\rightarrow$ on configurations of $\mathcal{A}$: we have $\langle g, (\gamma w, i), \mathbf{m} \rangle \rightarrow \langle g', (w'w, i), \mathbf{m}' \rangle$ for all $w \in \Gamma^*$ iff (1) there is a rule $g|\gamma \hookrightarrow g'|w'$ in $\Delta_c$ and $\mathbf{m}' = \mathbf{m}$ or (2) there is a rule $g|\gamma \hookrightarrow g'|w' \triangleright \gamma'$ in $\Delta_c$ and $\mathbf{m}' = \mathbf{m} + [[(\gamma', 0)]]$. We extend the *thread step* relation $\rightarrow^+$ to be the irreflexive-transitive closure of $\rightarrow$; thus $c \rightarrow^+ c'$ if there is a sequence $c \rightarrow c_1 \rightarrow \ldots c_k \rightarrow c'$ 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 $\mathcal{A}$:

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

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

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

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 $c_0 \Rightarrow c_1 \ldots$.

## 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 $\ell$ 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 $\gamma$. 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 $\Delta_c$, but additionally (1) makes visible as the input alphabet the initial symbols (from $\Gamma$) 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 $(g_1, \gamma, g_2)$ 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 $\gamma$ on top of its stack (without being active in the interim). A jump $(g, \perp)$ 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 $\Gamma \cup G \times \Gamma \times G \cup G \times \{\perp\}$.

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

(1) There is a transition $\text{init} \xrightarrow{\varepsilon|\perp/\gamma\perp} g$ 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 $\gamma_2 \in \Gamma_\perp$ there is a transition $g_1 \xrightarrow{\varepsilon|\gamma_1/w} (g_2, \perp)$, and a transition $(g_2, \perp) \xrightarrow{(g_2,\perp)|\gamma_2/\gamma_2} \text{end}$ in $E$,

(6) For every $g_1 \hookrightarrow g_2 \in \Delta_t$ there is a transition $g_1 \xrightarrow{(g_2,\perp)|\perp/\perp} \text{end}$ 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:

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

The language $L_{(g,\gamma)}^{(i)}$ is a context-free language. In the definition, we use the end of stack symbol $\perp$ 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. [2020]. 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 $\Delta_r$. 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 $\rho$ 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:

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

$$(\#, [[(\mathsf{foo}, 0)^1, (\mathsf{foo}, 0)^3]]) \overset{*}{\Rightarrow} ((\mathsf{foo}, 0)^3, [[(\mathsf{foo}, 0)^1]]) \overset{*}{\Rightarrow} \ldots$$

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 $\rho = c_0 \Rightarrow c_1 \Rightarrow \ldots$ 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* 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* FNTERM[$K$] *is decidable.*

THEOREM 2.4 (FAIR STARVATION). *For each $K \in \mathbb{N}$, the problem* 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 [Czerwinski 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. 2020]. We recall the main ideas as a step toward the more complex proof for *fair* termination.

### 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 $\Delta_c$. Informally, the "forgotten" spawned thread $\gamma'$ 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 $L_{(g,\gamma)}^{(i)}$ of each thread into a *regular* language by taking *downward closures*.

We need some definitions. For any alphabet $\Sigma$, 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 $L\downarrow := \{w' \in \Sigma^* \mid \exists w \in L : w' \sqsubseteq w\}$. 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:

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

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.

### 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 $c_0 \xrightarrow{\delta_0} c_1 \xrightarrow{\delta_1} \dots$. 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 $V(\mathcal{A}) = (G \times (\Gamma \times \{0, \dots, K\} \cup \{\#\}), (\Gamma \cup \{\varepsilon\}) \times \{0, \dots, K+1\}, E)$. 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 $\gamma$ and has been previously context switched $i$ times, and for each $\gamma' \in \Gamma$ and $i \in \{0, \dots, K+1\}$, the value $u(\gamma', i)$ represents the number of pending threads with stack $\gamma'$ 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 $i \in \{0, \dots, K\}$, the VASS has a transition that changes $(g, \gamma, i)$ to $(g', \gamma', i)$. For each transition $g|\gamma \hookrightarrow g'|\gamma' \triangleright \gamma'' \in \Delta_c$ and for each $i \in \{0, \dots, K\}$, the VASS has a transition that changes $(g, \gamma, i)$ to $(g', \gamma', i)$ and puts a token in $(\gamma'', 0)$. For each transition $g|\gamma \mapsto g'|\gamma' \in \Delta_i$ and for each $i \in \{0, \dots, K\}$, the VASS has a transition that changes $g$ to $(g', \#)$ while putting a token into $(\gamma', i+1)$. For each $g \mapsto g' \triangleleft \gamma \in \Delta_r$, there is a transition $(g, \#)$ to $(g', \gamma, i)$ that takes a token from $(\gamma, i)$. For each $g \mapsto g' \in \Delta_t$, there is a transition $(g, \varepsilon, i)$ 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 $((g_0, \#), [[\gamma_0, 0]])$. ∎

### 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. 2020] 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*, $\Omega$ is a finite set of *balloon states*, $\Phi$ 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) $op = \text{inflate}(\sigma, S)$, where $\sigma \in \Omega$ and $S \subseteq \mathbb{N}^\Phi$ is a semi-linear subset of $\mathbb{N}^\Phi$.
(3) $op = \text{deflate}(\sigma, \sigma', \pi, p)$, where $\sigma, \sigma' \in \Omega$, $\pi \in \Phi$, $p \in P$.
(4) $op = \text{burst}(\sigma)$, 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 $op = \text{inflate}(\sigma, S)$ 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 $\sigma$ and multiset $\mathbf{k}$ for some $\mathbf{k} \in S$.
(3) If $op = \text{deflate}(\sigma, \sigma', \pi, p)$ 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 $\pi$ from $\mathbf{k}$ to place $p$ in $\mathbf{m}$, and update the balloon state $\sigma$ to $\sigma'$. Here we say the balloon $(\sigma, \mathbf{k})$ was *deflated*.
(4) If $op = \text{burst}(\sigma)$ 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 $\sigma$ 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 $\rho$ 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 $\rho$ 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 $\rho$, there is a future point in $\rho$ 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 $\Gamma$ 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 $\Gamma$ 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 $\gamma$. 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 $\geq 1$ is then represented by a balloon where the number of tokens on balloon place $(\gamma, i)$ is equal to the number of threads with stack content $\gamma$ that $t$ will spawn during its $i$th segment. The spawns for segment 0 are transferred to the place set $\Gamma$ 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 $\gamma$ 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

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

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 $t = (g_0', \gamma_0', (g_1, \gamma_1, g_1') \ldots (g_K, \gamma_K, g_K'), g_{K+1}) \in \mathcal{T}(\mathcal{A}, K)$ 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 $\Gamma$ to the input alphabet of $\mathcal{P}_{(g_0', \gamma_0')}$. It is then redefined as $\Sigma = \Gamma \times \{0, \ldots, K\} \cup G \times \Gamma \times G \cup G \times \{\bot\}$, 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 $(\gamma, k)$.

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:

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

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 $\Gamma \times \{0, \ldots, K\}$, which we denote $\mathsf{sl}(t)$. We also define the set of all semi-linear sets that arise in this way as $\mathsf{SL}(\mathcal{A}, K) := \{\mathsf{sl}(t) \mid t \in \mathcal{T}(\mathcal{A}, K), L_t \neq \emptyset\}$.

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 $\gamma_0$ 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.

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 $q = q'$, $\mathbf{m} \preceq \mathbf{m}'$, and $\mathbf{n} \preceq \mathbf{n}'$. Moreover, we write $\mathrm{p}(c) \leq_{\mathrm{p}} \mathrm{p}(c')$ if $q = q'$, $\mathbf{m} \preceq \mathbf{m}'$, and $\partial\mathbf{n} \preceq \partial\mathbf{n}'$. 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 $c_1 \rightarrow c_1'$ and $c_1 \leq c_2$, then there is some $c_2'$ such that $c_1' \leq c_2'$ and $c_2 \rightarrow c_2'$.

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 $\rho$, converting it into a progressive run $\rho'$ with a special property: there exist infinitely many configurations in $\rho'$ 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 $\sigma$, 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 $\mathrm{seq}_i$ that it undergoes. Given a run $\rho$, we produce a corresponding *canonical run-with-id* $\tau$ 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 $\tau$ (which consist of sequences of configurations-with-id), we define the notion of a *progressive* run-with-id $\tau$, which is one such that the sequence $\mathrm{seq}_i$ associated with any id $i$ in $\tau$ is either infinite, or $\mathrm{seq}_i$ is finite with the last operation being a burst. If every id $i$ undergoes a burst operation in a run-with-id $\tau$, we say "$\tau$ bursts every balloon." We abuse terminology by saying "$\rho$ bursts every balloon" for a run $\rho$ to mean that there is a *corresponding* run-with-id $\tau$ which bursts every balloon. It is easy to see that the canonical run-with-id $\tau$ associated with a progressive run $\rho$ is "almost" progressive. By always picking the id which has been idle for the longest time, we can convert $\tau$ 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 $\pi_j$ 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.

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 $\tau''$. In order to convert the given $\tau''$ into a progressive $\tau'$ in which every balloon is burst, we need to burst those id's $i$ such that $\text{seq}_i$ is infinite in $\tau''$. Since only a finite number of non-trivial deflates can be performed by a given balloon $i$, this implies that $\text{seq}_i$ 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 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 $\tau'$ from a run $\tau$ 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 $\tau$. 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
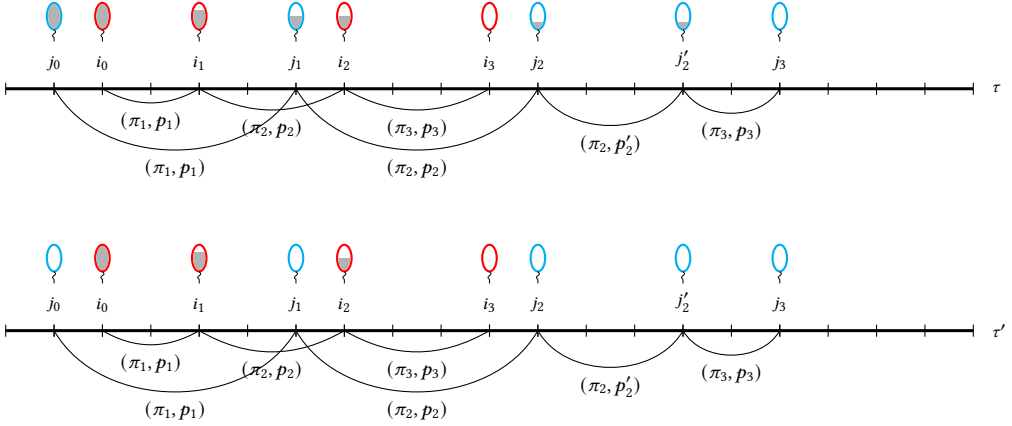
Fig. 1. Top: Initial run $\tau$ with two non-empty balloons which perform the same sequence of three non-trivial deflates $(\pi_1, p_1), (\pi_2, p_2), (\pi_3, p_3)$. Bottom: Modified run $\tau'$ after shifting tokens from the cyan balloon inflated at $j_0$ to the red balloon inflated at $i_0$.

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 $\tau'$, 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 $\tau$. 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 $\tau$ may be modified to give a *valid* run-with-id $\tau'$. Note that the configurations-with-id $d_j'$ of $\tau'$ 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 $\tau$ of $\mathcal{V}'$ such that every balloon is burst, we produce a run $\tau'$ of $\mathcal{V}'$ from which it will be easy to extract an $A, B$-witness. Let $T_\infty$ be the set of types of balloons which occur infinitely often in $\tau$. Since every balloon is eventually burst in $\tau$, there has to be a configuration $d_0$ such that after $d_0$, every occurring balloon has a type in $T_\infty$. 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 $T_\infty$ 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 $\tau$. The prefix $d_0'' \xrightarrow{*} d_0'$ of the modified run $\tau'$ may contain balloons of arbitrary type. Between $d_0'$ and $d_1'$, there are only balloons in $T_\infty$. After $d_2'$, we have an infinite suffix where all balloons are of a type from $T_\infty$ 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 $\leq_p$ is a well-quasi-ordering, the sequence $d'_2, d'_3, \ldots$ must contain configurations $d'_l$ and $d'_m$ with $d'_l \leq_p d'_m$. We obtain an $A, B$-witness as follows. We choose the set $P_\infty$ which is the set of places which are non-empty infinitely often along $\tau'$ 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 $\tau''$ of $\mathcal{V}'$. Furthermore, since the unrolling $\tau''$ 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 $p(c_1)$ 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 $\rho$. 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 $\Omega$ that store the number of balloons which were created empty for each balloon state $\sigma$ 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 $\sigma$, the number of balloons that are inflated in $\rho$ with state $\sigma$ is bounded by $N$, this implies a bound of $|\Omega|N$ on the total number of balloons inflated in $\rho$. 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 $\sigma$ are inflated in a run $\rho = s_1 \xrightarrow{*} s_2$, 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 $\sigma$.

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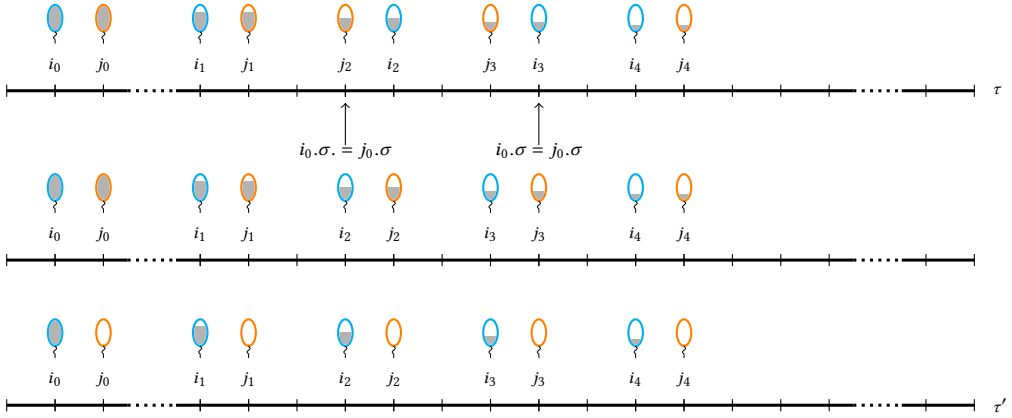
Fig. 2. Top: Initial run $\tau$ with two non-empty balloons of the same type: cyan balloon inflated at $i_0$ and orange balloon inflated at $j_0$. Middle: Switching cyan and orange balloons in the part of $\tau$ between $j_2$ and $i_3$. Bottom: Modified run $\tau'$ obtained by shifting token from orange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

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 $\tau'$ shown at the bottom of Figure 2, which contains one less non-empty balloon of type $t$ than $\tau$. It remains to show that such an id-switching surgery is always possible in a run $\tau$ 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 $R(r; n) \leq r^{r(n-2)+1}$.

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 $\tau$.

Formally, we construct a graph $G_t$ with vertex set $V_t$ of all id's in $\tau$ 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 $\pi_k$ 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
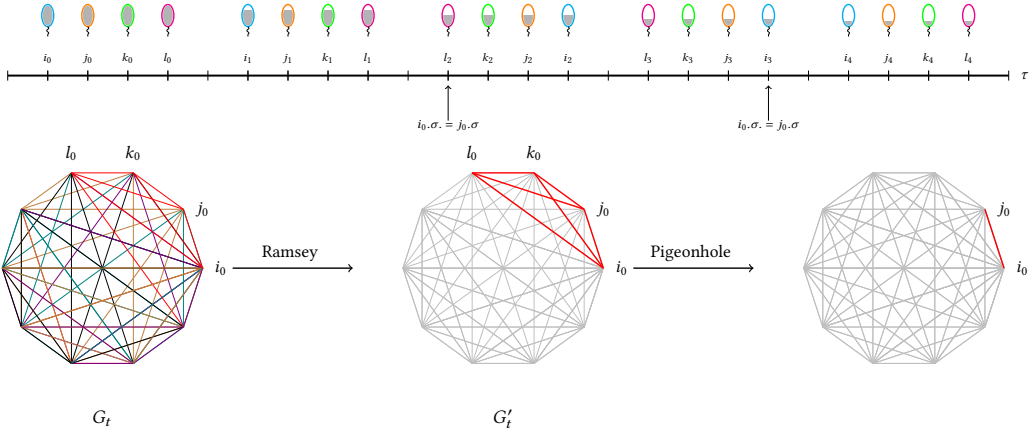
Fig. 3. Above:Four balloons inflated at $i_0 < j_0 < k_0 < l_0$ of the same type $t$ performing four deflate operations (subscripts denote deflate operations of the same balloon). Note that the ordering relationship of deflate operations between any pair of the four balloons is the same: between balloons $i_0$ 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 $\tau$.
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.

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.

### 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 $\gamma$ 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 $\rho$ 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 $\rho$ 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 $\rho$ 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 $\rho$ 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 $\rho$ 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 $(w, i)$.

### 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 $\text{supp}(\mathbf{m}) = \text{supp}(\mathbf{m}')$. Recall that $\text{supp}(\mathbf{m}) = \{x \in \Lambda \mid \mathbf{m}(x) > 0\}$ is the support of $\mathbf{m} \in \mathbb{M}[\Lambda]$. Here, the condition $\text{supp}(\mathbf{m}) = \text{supp}(\mathbf{m}')$ 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 $l \in [1, k]$.

**Starvation in terms of consistency**

This allows us to state the following reformulation of starvation, where $w$ does not appear explicitly. A progressive run $\rho$ 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 $\mathfrak{B} = (V_t)_{t \in \mathcal{T}}$ 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.*

**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 $\alpha_B$ to subsets of $\mathbb{M}[\Lambda]$ (point-wise) and to tuples of subsets of $\mathbb{M}[\Lambda]$ (component-wise). Note that for a tuple $\mathfrak{B} = (V_t)_{t \in \mathcal{T}}$ with $V_t \subseteq \mathbb{M}[\Lambda]$ for $t \in \mathcal{T}$, the tuple $\alpha_B(\mathfrak{B})$ 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* $\mathfrak{B} = (V_t)_{t \in \mathcal{T}}$ *is a tuple of* ***finite*** *subsets* $V_t \subseteq \mathbb{M}[\Lambda]$, *then* $\mathfrak{B}$ *is* $\mathfrak{S}_{\mathcal{A}}$-*consistent if and only if* $\alpha_B(\mathfrak{B})$ *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 $\alpha_B(\mathfrak{B})$. However, we may only conclude consistency of $\mathfrak{B}$ (and hence of the run) from consistency of $\alpha_B(\mathfrak{B})$ if $\mathfrak{B}$ 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 $\rho$ with consistent $\alpha_B(\mathfrak{B})$, then there also exists a run $\rho'$ with finite $\mathfrak{B}$ such that $\alpha_B(\mathfrak{B})$ is consistent, meaning by Theorem 6.2, $\rho'$ 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* $\mathfrak{B} = (V_t)_{t \in \mathcal{T}}$ *of finite subsets* $V_t \subseteq \mathbb{M}[\Lambda]$, *it is decidable whether* $\mathfrak{B}$ *is* $\mathfrak{S}_{\mathcal{A}}$-*consistent.*

**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 $\rho$ 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 $\mathcal{A}_{(i,\mathfrak{u})}$.

**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 $\Delta$ also contain a set $\Delta_{\mathfrak{u}}$ of *unfreezing rules* of the form $g \mapsto g' \triangleleft \gamma \circledast \gamma'$ and $\gamma_f$ 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 $\gamma$, while also freezing a thread with top of stack $\gamma'$. 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 $\Gamma$. In addition, there is one more rule:

Unfreeze

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

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 $g \mapsto g' \triangleleft \gamma \circledast \gamma'$ 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 $\gamma$. 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.

**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 $\mathfrak{u}$):

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

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 $c_1, c_2, \ldots$:

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

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 $i$th 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 $\alpha_B$). 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.

### 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. 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 $\Lambda$ 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* $\mathfrak{B} = (V_1, \ldots, V_k)$ *is a tuple of **finite** subsets* $V_j \subseteq \mathbb{M}[\Lambda]$, *then* $\mathfrak{B}$ *is* $\mathfrak{S}$-*consistent if and only if* $\alpha_B(\mathfrak{B})$ *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 $S = \{(a^n, n \cdot [[b]]) \mid n \in \mathbb{N}\}$ and $\mathfrak{S} = (S)$. Then a set $V \subseteq \mathbb{M}[\{b\}]$ 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 $\Theta$. Hence, for $q \in Q_j$ and $\Theta \subseteq \Lambda$, we have $(q, \Theta) \in \kappa(u)$ if and only if $q \xrightarrow{u \mid \mathbf{m}} q$ in $\mathcal{M}_j$ for some $\mathbf{m} \in \mathbb{M}[\Lambda]$ with $\text{supp}(\mathbf{m}) = \Theta$.

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 $\text{supp}(\mathbf{m}_h) = \text{supp}(\bar{\mathbf{m}})$. Then, in particular, $\text{supp}(\mathbf{m}_h) \subseteq \text{supp}(\mathbf{m}') = \text{supp}(\mathbf{m})$. 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 $\mathbf{m} \leq_1 \mathbf{m}' + (\ell - 1) \sum_{h=1}^n \mathbf{m}_h$, this implies $\mathbf{m} \in S_j \downarrow_{xy^\ell z}$. ∎

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* $\mathfrak{B} = (V_1, \ldots, V_k)$ *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 $\mathfrak{B}' = (V_1', \ldots, V_k')$ be obtained from $\mathfrak{B}$ 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 $\mathfrak{B}$ is $\mathfrak{S}$-consistent, there is a $w \in \Gamma^*$ that covers all elements of $\mathfrak{B}$. 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 $\mathfrak{B}$. 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 $\Pi_1^0$-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

# REFERENCES

P. A. Abdulla, K. Čerāns, B. Jonsson, and Yih-Kuan 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.

K.R. Apt and E.-R. 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 Thinniyam, and Georg Zetzsche. [n. d.]. Context-Bounded Verification of Liveness Properties for Multithreaded Shared-Memory Programs (full version). http://ramanathan.ts.gitlab.io/html-webpage/publications/POPL2021withAppendix.pdf.

Pascal Baumann, Rupak Majumdar, Ramanathan S. Thinniyam, and Georg Zetzsche. 2020. 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.

J 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 Czerwinski, 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

P. Erdős and R. 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.

S. 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.

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

F. P. 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.

## A STRENGTHENING FAIRNESS TO PROGRESSIVE RUNS

In this section we strengthen the notion of fairness for DCPS to progressiveness by proving Lemma 4.3.

*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".

*Formal construction.* Let $K \in \mathbb{N}$ and $\mathcal{A} = (G, \Gamma, \Delta, g_0, \gamma_0)$ be a DCPS. We construct the DCPS $\widetilde{\mathcal{A}} = (\widetilde{G}, \widetilde{\Gamma}, \widetilde{\Delta}, (g_0, G), \gamma_0)$, where

- $\widetilde{G} = (G \times \mathbb{P}(G)) \cup (G \times \{0, \ldots, K\} \times \mathbb{P}(G)) \cup (\bar{G} \times \{0, \ldots, K\} \times \mathbb{P}(G))$, where $\bar{G} = \{\bar{g} | g \in G\}$ and $\mathbb{P}(G)$ is the powerset of $G$, i.e. the set of all subsets of $G$,
- $\widetilde{\Gamma} = \Gamma_{\perp} \cup (\Gamma_{\perp} \times \{0, \ldots, K\}) \cup \bar{\Gamma}$, where $\Gamma_{\perp} = \Gamma \cup \{\perp\}$ and $\bar{\Gamma} = \{\bar{\gamma} | \gamma \in \Gamma\}$,
- $\widetilde{\Delta} = \widetilde{\Delta}_c \cup \widetilde{\Delta}_i \cup \widetilde{\Delta}_r \cup \widetilde{\Delta}_t$ consists of the following transitions rules:
  (1) $(g_1, G') \mapsto (g_2, G') \triangleleft \gamma \in \widetilde{\Delta}_r$ for all $G' \supseteq \{g_1\}$ iff $g_1 \mapsto g_2 \triangleleft \gamma \in \Delta_r$.
  (2) $(g_1, G') \mapsto (g_2, k, G') \triangleleft (\gamma, k) \in \widetilde{\Delta}_r$ for all $k \in \{1, \ldots, K\}$, $G' \supseteq \{g_1\}$ iff $g_1 \mapsto g_2 \triangleleft \gamma \in \Delta_r$.
  (3) $(g, G') | \gamma \hookrightarrow (g, 0, G') | (\gamma, 0).\perp \in \widetilde{\Delta}_c$ for all $g \in G$, $\gamma \in \Gamma$.
  (4) $(g, k, G') | \gamma \hookrightarrow (g, k, G') | (\gamma, k) \in \widetilde{\Delta}_c$ for all $k \in \{0, \ldots, K\}$, $g \in G$, $\gamma \in \Gamma_{\perp}$.
  (5) $(g_1, k, G') | (\gamma, k) \hookrightarrow (g_2, k, G') | w \in \widetilde{\Delta}_c$ for all $k \in \{0, \ldots, K\}$ iff $g_1 | \gamma \hookrightarrow g_2 | w \in \Delta_c$.
  (6) $(g_1, k, G') | (\gamma_1, k) \hookrightarrow (g_2, k, G') | w \triangleright \gamma_2 \in \widetilde{\Delta}_c$ for all $k \in \{0, \ldots, K\}$ iff $g_1 | \gamma_1 \hookrightarrow g_2 | w \triangleright \gamma_2 \in \Delta_c$.
  (7) $(g_1, k, G') | (\gamma_1, k) \mapsto (g_2, G') | (\gamma_2, k + 1) \gamma_3 \in \widetilde{\Delta}_i$ for all $k \in \{0, \ldots, K - 1\}$, $G' \supseteq \{g_2\}$ iff $g_1 | \gamma_1 \mapsto g_2 | \gamma_2 \gamma_3 \in \Delta_i$, where $\gamma_2 \in \Gamma$ and $\gamma_3 \in \Gamma \cup \{\varepsilon\}$.
  (8) $(g_1, k, G') | (\perp, k) \hookrightarrow (\bar{g}_2, k, G') | (\perp, k) \in \widetilde{\Delta}_c$ for all $k \in \{0, \ldots, K\}$ iff $g_1 \mapsto g_2 \in \Delta_t$
  (9) $(\bar{g}, k, G') | (\perp, k) \mapsto (\bar{g}, k + 1, G') | (\perp, k + 1) \in \widetilde{\Delta}_i$ for all $k \in \{0, \ldots, K - 1\}$, $g \in G' \subseteq G$.
  (10) $(\bar{g}, k, G') | (\perp, k) \hookrightarrow (\bar{g}, k, G') | (\perp, k) \in \widetilde{\Delta}_c$ for all $k \in \{0, \ldots, K - 1\}$, $g \in G' \subseteq G$.
  (11) $(\bar{g}, k, G') \mapsto (\bar{g}, k, G') \triangleleft (\perp, k) \in \widetilde{\Delta}_r$ for all $k \in \{0, \ldots, K\}$, $g \in G' \subseteq G$.
  (12) $(\bar{g}, K, G') | (\perp, K) \hookrightarrow (\bar{g}, K, G') | \varepsilon \in \widetilde{\Delta}_c$ for all $g \in G' \subseteq G$.
  (13) $(\bar{g}, K, G') \mapsto (g, G') \in \widetilde{\Delta}_t$ for all $g \in G' \subseteq G$.
  (14) $(g_1, k, G') | (\gamma, k) \hookrightarrow (\bar{g}_2, k, G') | \varepsilon \triangleright \bar{\gamma} \in \widetilde{\Delta}_c$ for all $k \in \{0, \ldots, K\}$, $g_2 \in G' \subseteq G$, $\gamma \in \Gamma$ iff $g_1 | \gamma_1 \mapsto g_2 | w \in \Delta_i$ for some $w \in \Gamma^*$ and $g' \mapsto g \triangleleft \gamma \in \Delta_r$ for some $g' \in G'$, $g \in G$.
  (15) $(g_1, k, G') | (\gamma, k) \hookrightarrow (\bar{g}_2, k, G') | \varepsilon \in \widetilde{\Delta}_c$ for all $k \in \{0, \ldots, K\}$, $g_2 \in G' \subseteq G$, $\gamma \in \Gamma$ iff $g_1 | \gamma_1 \mapsto g_2 | w \in \Delta_i$ for some $w \in \Gamma^*$ and $g' \mapsto g \triangleleft \gamma \notin \Delta_r$ for all $g' \in G'$, $g \in G$.
  (16) $(\bar{g}, k, G') | \gamma \hookrightarrow (\bar{g}, k, G') | \varepsilon \in \widetilde{\Delta}_c$ for all $k \in \{0, \ldots, K\}$, $g \in G' \subseteq G$, $\gamma \in \Gamma$.
  (17) $(\bar{g}, k, G') | \perp \hookrightarrow (\bar{g}, k, G') | (\perp, k) \in \widetilde{\Delta}_c$ for all $k \in \{0, \ldots, K\}$, $g \in G' \subseteq G$.
  (18) $(g, G') \mapsto (g, 0, G') \triangleleft \bar{\gamma} \in \widetilde{\Delta}_r$ for all $k \in \{1, \ldots, K\}$, $g \in G' \subseteq G$, $\gamma \in \Gamma$.
  (19) $(g, 0, G_1) | \bar{\gamma} \hookrightarrow (\bar{g}, 0, G_2) | (\perp, 0) \in \widetilde{\Delta}_c$ for all $g \in G_2 \subseteq G_1 \subseteq G$, $\gamma \in \Gamma$, where $G_2 = \{g_1 \in G_1 | \forall g_2 \in G: g_1 \mapsto g_2 \triangleleft \gamma \notin \Delta_r\}$.

We proceed by constructing a run of $\widetilde{\mathcal{A}}$ from a fair run of $\mathcal{A}$ and argue that this construction results in a progressive run and preserves starvation, which proves the if direction of the two points of Lemma 4.3. For the only if direction we argue that we can also do this backwards, starting with a progressive run from $\widetilde{\mathcal{A}}$ and constructing one from $\mathcal{A}$.

Let $\rho$ be a infinite fair run of $\mathcal{A}$. We begin by formalizing the notion threads reaching a certain local configuration and never progressing from there. Consider a local configuration $t = (w, i)$ of $\mathcal{A}$ that over the course of $\rho$ is only removed from the bag finitely often (via applications of Resume) and is added to the bag more often than it is removed (via applications of Swap). Let $n_t$ be the number of times $t$ is removed from the bag in this way. Then from the $(n + 1)$th applications of Swap and onwards adding $t$ to the bag over the course of $\rho$, we say the local configuration $t$ added to the bag is *stagnant*.

Now we can properly construct an infinite run $\widetilde{\rho}$ of $\widetilde{\mathcal{A}}$. Whenever $\rho$ reaches a configuration with global state $g$, $\widetilde{\rho}$ mimics it by reaching a global state that includes $g$ in the state tuple. Similarly any non-stagnant local configuration $(\gamma w, i)$ occurring on $\rho$ corresponds to a local configuration $((\gamma, i)w, i)$ occurring on $\widetilde{\rho}$, whereas the empty stack corresponds to stack content $\perp$ or $(\perp, i)$. Newly spawned local configurations $(\gamma, 0)$ look the same in both runs. For the initial configuration $\langle g_0, \perp, [[(\gamma_0, 0)]]\rangle$ of $\rho$ and $\langle (g_0, G), \perp, [[(\gamma_0, 0)]]\rangle$ of $\widetilde{\rho}$ this correspondence evidently holds. Let us now go over the consecutive thread step and scheduler step relations of $\rho$ and construct $\widetilde{\rho}$ appropriately:

In the following, for any step relation between two configurations of $\mathcal{A}$, we write a number 1 to 17 above the arrow to denote which rule of $\widetilde{\Delta}$ is being applied.

**Case** $\langle g_1, \#, \mathbf{m} + [[(\gamma w, i)]]\rangle \mapsto \langle g_2, (\gamma w, i), \mathbf{m}\rangle$ **due to** $g_1 \mapsto g_2 \triangleleft \gamma \in \Delta_r$:
If $i = 0$ (and therefore also $w = \varepsilon$) then

$$\langle (g_1, G'), \#, \widetilde{\mathbf{m}} + [[(\gamma, 0)]]\rangle \xrightarrow{1} \langle (g_2, G'), (\gamma, 0), \widetilde{\mathbf{m}}\rangle \xrightarrow{3} \langle (g_2, 0, G'), ((\gamma, 0)\perp, 0), \widetilde{\mathbf{m}}\rangle,$$

otherwise (if $i \geq 1$)

$$\langle (g_1, G'), \#, \widetilde{\mathbf{m}} + [[((\gamma, i)w\perp, i)]]\rangle \xrightarrow{2} \langle (g_2, i, G'), ((\gamma, i)w\perp, i), \widetilde{\mathbf{m}}\rangle.$$

**Case** $\langle g_1, (\gamma w, i), \mathbf{m}\rangle \rightarrow \langle g_2, (w'w, i), \mathbf{m}\rangle$ **due to** $g_1|\gamma \hookrightarrow g_2|w' \in \Delta_c$:

$$\langle (g_1, i, G'), ((\gamma, i)w\perp, i), \widetilde{\mathbf{m}}\rangle \xrightarrow{5} \langle (g_2, i, G'), (w'w\perp, i), \widetilde{\mathbf{m}}\rangle,$$

from here, if $w'w = \gamma'w'' \neq \varepsilon$ for some $\gamma' \in \Gamma$, $\widetilde{\rho}$ continues with

$$\langle (g_2, i, G'), (\gamma'w''\perp, i), \widetilde{\mathbf{m}}\rangle \xrightarrow{4} \langle (g_2, i, G'), ((\gamma', i)w''\perp, i), \widetilde{\mathbf{m}}\rangle,$$

otherwise (if $w'w = \varepsilon$) $\widetilde{\rho}$ continues with

$$\langle (g_2, i, G'), (\perp, i), \widetilde{\mathbf{m}}\rangle \xrightarrow{4} \langle (g_2, i, G'), ((\perp, i), i), \widetilde{\mathbf{m}}\rangle.$$

**Case** $\langle g_1, (\gamma w, i), \mathbf{m}\rangle \rightarrow \langle g_2, (w'w, i), \mathbf{m} + [[(\gamma', 0)]]\rangle$ **due to** $g_1|\gamma \hookrightarrow g_2|w' \triangleright \gamma' \in \Delta_c$:

$$\langle (g_1, i, G'), ((\gamma, i)w\perp, i), \widetilde{\mathbf{m}}\rangle \xrightarrow{6} \langle (g_2, i, G'), (w'w\perp, i), \widetilde{\mathbf{m}} + [[(\gamma', 0)]]\rangle,$$

if $w'w = \gamma''w'' \neq \varepsilon$ for some $\gamma'' \in \Gamma$ then $\widetilde{\rho}$ continues with

$$\langle (g_2, i, G'), (\gamma''w''\perp, i), \widetilde{\mathbf{m}} + [[(\gamma', 0)]]\rangle \xrightarrow{4} \langle (g_2, i, G'), ((\gamma'', i)w''\perp, i), \widetilde{\mathbf{m}} + [[(\gamma', 0)]]\rangle,$$

otherwise (if $w'w = \varepsilon$) $\widetilde{\rho}$ continues with

$$\langle (g_2, i, G'), (\perp, i), \widetilde{\mathbf{m}} + [[(\gamma', 0)]]\rangle \xrightarrow{4} \langle (g_2, i, G'), ((\perp, i), i), \widetilde{\mathbf{m}} + [[(\gamma', 0)]]\rangle.$$

**Case** $\langle g_1, (\gamma w, i), \mathbf{m}\rangle \mapsto \langle g_2, \#, \mathbf{m} + [[(w'w, i+1)]]\rangle$ **due to** $g_1|\gamma \mapsto g_2|w' \in \Delta_i$:
By definition of $\Delta_i$ we have $1 \leq |w'| \leq 2$ and therefore $w' = \gamma_1\gamma_2$ for some $\gamma_1 \in \Gamma, \gamma_2 \in \Gamma \cup \{\varepsilon\}$. If $(w'w, i+1)$ is not stagnant here, then

$$\langle (g_1, i, G'), ((\gamma, i)w\perp, i), \widetilde{\mathbf{m}}\rangle \xrightarrow{7} \langle (g_2, G'), \#, i), \widetilde{\mathbf{m}} + [[((\gamma_1, i+1)\gamma_2 w\perp, i+1)]]\rangle.$$

Otherwise (if $(w'w, i+1)$ is stagnant here), if $G'$ still contains states that allow RESUME rules on $\gamma_1$, we first save this top of stack symbol in the bag (as $\bar{\gamma}_1$):

$$\langle (g_1, i, G'), ((\gamma, i)w\bot, i), \widetilde{\mathbf{m}} \rangle \xrightarrow{14} \langle (\bar{g}_2, i, G'), (w\bot, i), \widetilde{\mathbf{m}} + [[(\bar{\gamma}_1, 0)]] \rangle.$$

If $G'$ does not allow for any RESUME rules on $\gamma_1$ we do not save this top of stack symbol:

$$\langle (g_1, i, G'), ((\gamma, i)w\bot, i), \widetilde{\mathbf{m}} \rangle \xrightarrow{15} \langle (\bar{g}_2, i, G'), (w\bot, i), \widetilde{\mathbf{m}} \rangle.$$

In both of these stagnant cases we continue by almost emptying the stack of the active thread in $\widetilde{\rho}$. Let $\widetilde{\mathbf{m}}' = \widetilde{\mathbf{m}} + [[(\bar{\gamma}_1, 0)]]$ in the first case and $\widetilde{\mathbf{m}}' = \widetilde{\mathbf{m}}$ in the second case:

$$\langle (\bar{g}_2, i, G'), (w\bot, i), \widetilde{\mathbf{m}}' \rangle \xrightarrow{16} \ldots \xrightarrow{16} \langle (\bar{g}_2, i, G'), (\bot, i), \widetilde{\mathbf{m}}' \rangle \xrightarrow{17} \langle (\bar{g}_2, i, G'), ((\bot, i), i), \widetilde{\mathbf{m}}' \rangle.$$

From here $\widetilde{\rho}$ continues with what we call an *extended thread termination*, where we force the active thread to make its remaining context switches (up to $K$) and then terminate:

$$\langle (\bar{g}_2, i, G'), ((\bot, i), i), \widetilde{\mathbf{m}}' \rangle \xmapsto{9} \langle (\bar{g}_2, i+1, G'), \#, \widetilde{\mathbf{m}}' + [[((\bot, i+1), i+1)]] \rangle$$

$$\xmapsto{11} \langle (\bar{g}_2, i+1, G'), ((\bot, i+1), i+1), \widetilde{\mathbf{m}}' \rangle$$

$$\xrightarrow{10} \langle (\bar{g}_2, i+1, G'), ((\bot, i+1), i+1), \widetilde{\mathbf{m}}' \rangle$$

$$\vdots \quad ((K-i) \text{ repetitions of the rule sequence } (9), (11), (10))$$

$$\xrightarrow{10} \langle (\bar{g}_2, K, G'), ((\bot, K), K), \widetilde{\mathbf{m}}' \rangle$$

$$\xrightarrow{12} \langle (\bar{g}_2, K, G'), (\varepsilon, K), \widetilde{\mathbf{m}}' \rangle$$

$$\xmapsto{13} \langle (g_2, G'), \#, \widetilde{\mathbf{m}}' \rangle.$$

Of course if $i = K$ only the last two steps are part of $\widetilde{\rho}$. We also note that rule (10) is only necessary to facilitate alternation between thread step and scheduler step relations, as required by runs of DCPS.

**Case** $\langle g_1, (\varepsilon, i), \mathbf{m} \rangle \mapsto \langle g_2, \#, \mathbf{m} \rangle$ **due to** $g_1 \mapsto g_2 \in \Delta_{\mathsf{t}}$**:**

$$\langle (g_1, i, G'), ((\bot, i), i), \widetilde{\mathbf{m}} \rangle \xrightarrow{8} \langle (\bar{g}_2, i, G'), ((\bot, i), i), \widetilde{\mathbf{m}}' \rangle,$$

from here $\widetilde{\rho}$ continues with an extended thread termination, as explained previously.

**Once a particular configuration** $\langle g, \#, \mathbf{m} \rangle$ **is reached:**

Let $G_{\mathsf{inf}} \subseteq G$ be the set of global states that occur infinitely often on configurations of $\rho$ with no active thread. Let $\langle g, \#, \mathbf{m} \rangle$ be a configuration upon which only global states in $G_{\mathsf{inf}}$ occur on configurations with no active thread on $\rho$ (including for this configuration itself, meaning $g \in G_{\mathsf{inf}}$). Furthermore, no new top of stack symbols appear among stagnant threads after this configuration has occurred. Now we handle all the threads with top of stack symbol in $\bar{\Gamma}$ spawned over the course of $\widetilde{\rho}$:

$$\langle (g, G_1), \#, \widetilde{\mathbf{m}} + [[(\bar{\gamma}, 0)]] \rangle \xmapsto{18} \langle (g, 0, G_1), (\bar{\gamma}, 0), \widetilde{\mathbf{m}} \rangle \xrightarrow{19} \langle (\bar{g}, 0, G_2), ((\bot, 0), 0), \widetilde{\mathbf{m}} \rangle.$$

Here $G_2 = \{g_1 \in G_1 | \forall g_2 \in G: g_1 \mapsto g_2 \lhd \gamma \notin \Delta_{\mathsf{r}}\}$, i.e. the subset of $G_1$ that no longer contains any global states that allow RESUME rules for $\gamma$. We continue with an extended thread termination, upon which we repeat these steps until no more local configurations with top of stack symbol in $\bar{\Gamma}$ are in the bag.

The run $\widetilde{\rho}$ is sound: Firstly, the correspondence of local configurations we mentioned earlier is upheld throughout the run. Secondly, we always save the cs-number of the active thread in the state tuple and we have no cs-number in the state tuple if there is no active thread, which we assumed to hold at the start of all step sequences we added to $\widetilde{\rho}$. Finally, the subset $G' \subseteq G$ in each state tuple restricts which global states can occur while there is no active thread, but we update it in such a way that none of the steps we want to make are disallowed: As long as we visit global states $\notin G_{\text{inf}}$ in configurations with no active thread $G' = G$ holds. After such visits stop happening, we update $G'$ based on all the stagnant threads that occurred. However, the top of stack symbols of stagnant threads cannot allow for the application of Resume rules from global states in $G_{\text{inf}}$, because that would violate the fairness condition of $\rho$. Therefore $G_{\text{inf}} \subseteq G'$ still holds after we finish updating $G'$, which means all future steps are still allowed by $G'$.

The run $\widetilde{\rho}$ is also progressive: Firstly, every local configuration on $\widetilde{\rho}$ that corresponds to a non-stagnant thread of $\rho$ is resumed at some point, because its correspondent was handled fairly on $\rho$, which implies resumption for non-stagnant threads. Secondly, the local configurations of $\widetilde{\rho}$ corresponding to stagnant threads of $\rho$ do not have to be resumed, since they always terminate. Thirdly, all threads with top of stack in $\bar{\Gamma}$ are resumed eventually: Since we only update $G'$ once all different types of these threads were spawned, the update disallows any Resume rules for all top of stack symbols of stagnant threads occurring afterwards. Therefore the only transition rule in $\widetilde{\Delta}_{\text{c}}$ that spawns these threads (rule 14) can no longer be applied (we apply rule 15 instead). Since these threads then no longer occur in the bag afterwards, they are of course do not have to be resumed. Finally, every thread that terminates does so after exactly $K$ context switches: This is due to the extended thread termination that is performed every time the bottom of stack symbol $\bot$ is reached.

A thread is starved by $\widetilde{\rho}$ iff a thread is starved by $\rho$: It is clear that stagnant threads cannot be starved, and non-stagnant threads of $\rho$ have their correspondents handled in same way on $\widetilde{\rho}$. The only mismatch in starvation can thus occur on the threads with top of stack symbol in $\bar{\Gamma}$. However, we already argued that after a certain point, none of these threads even occur as part of a configuration of $\widetilde{\rho}$ anymore. Therefore they cannot be starved either. This concludes the if-direction of the proof.

For the *only if* direction let $\widetilde{\rho}$ be an infinite progressive run of $\widetilde{\mathcal{A}}$. We observe that we can do most of the previous construction backwards to obtain an infinite run $\rho$ of $\mathcal{A}$. Whenever we guess a thread to be stagnant by using rule (14) or (15) on $\widetilde{\rho}$, we instead keep its correspondent around in $\rho$ forever. Since each local configuration with top of stack symbol in $\bar{\Gamma}$ has to have a Resume rule applied to it on $\widetilde{\rho}$, we eventually are restricted to global states that do not allow for Resume rules on the corresponding top of stack symbols in $\Gamma$. This means the threads we keep around forever on $\rho$ are "stuck" and therefore handled fairly. Restricting the global states based on some symbol $\bar{\gamma}$ also disallows new threads with the same symbol to spawn during $\widetilde{\rho}$ afterwards. This means at some point no new threads with top of stack symbol in $\bar{\Gamma}$ are newly added to the bag, but the old threads disappear eventually because of progressiveness. Thus these threads cannot be exhibit starvation that then would not be present in $\rho$. Finally, all remaining threads of $\rho$ are handled the same way as their correspondents of $\widetilde{\rho}$, meaning the progressiveness of the latter implies fairness of the former.

# B   PROOFS FROM SECTION 4.1

In this section we prove Theorem 4.4 formally, which involves constructing a VASSB $\mathcal{V}$ with the same progressiveness properties (when starting in configuration $c_0 = (q_0, \emptyset, \emptyset)$) as a given DCPS $\mathcal{A}$.

*Formal construction.* Let $K \in \mathbb{N}$ and $\mathcal{A} = (G, \Gamma, \Delta, g_0, \gamma_0)$ be a DCPS with $\Gamma = \{\gamma_0, \ldots, \gamma_l\}$. Construct a VASSB $\mathcal{V} = (Q, P, \bar{Q}, \bar{P}, E)$, where

- $\bar{Q} = \big(\mathcal{T}(\mathcal{A},K) \times \{0,\ldots,K\} \times \mathsf{SL}(\mathcal{A},K) \times (\Gamma \cup \{\varepsilon\})\big) \cup \{\bot\}$,
- $\tilde{P} = \Gamma \times \{0,\ldots,K\}$,
- $Q = G \cup (G \times \bar{Q} \times G) \cup \{q_0\}$,
- $P = \Gamma$,
- $E$ contains the following edges of the form $q \xrightarrow{op} q'$:

(1) $op = \delta \in \mathbb{Z}^P$ with $\delta(p) = 0$ for all $p \in P$, iff $q = q_0$, and $q' = (g_0, \bar{q}, g_1)$ with $\bar{q} = (t, 0, \mathsf{sl}(t), \varepsilon)$ where $t \in \mathcal{T}(\mathcal{A},K)$ with a first segment going from $g_0$ to $g_1$.

(2) $op = \delta \in \mathbb{Z}^P$ with $\delta(\gamma) = -1$ and $\delta(p) = 0$ for all $p \neq \gamma$, iff there is a rule $g_1 \mapsto g_2 \lhd \gamma \in \Delta_r$, $q = g_1 \in G$, and $q' = (g_2, \bar{q}, g_3)$ with $\bar{q} = (t, 0, \mathsf{sl}(t), \varepsilon)$ where $t \in \mathcal{T}(\mathcal{A},K)$ with a first segment going from $g_2$ to $g_3$.

(3) $op = \mathsf{inflate}(\bar{q}, S)$, iff $q = (g_1, \bar{q}, g_2)$ where $\bar{q} = (t, 0, S, \varepsilon)$, and $q' = (g_1, \bar{q}', g_2)$ where $\bar{q}' = (t, 0, S, \gamma_0)$, for some $t \in \mathcal{T}(\mathcal{A},K)$, $S \in \mathsf{SL}(\mathcal{A},K)$, $g_1, g_2 \in G$.

(4) $op = \mathsf{deflate}(\bar{q}, \bar{q}', \bar{p}, p)$, iff $p = \gamma_i$, $\bar{p} = (\gamma_i, j)$, $\bar{q} = (t, j, S, \gamma_i)$, $\bar{q}' = (t, j, S, \gamma_{i+1})$, $q = (g_1, \bar{q}, g_2)$, and $q' = (g_1, \bar{q}', g_2)$, for some $j \in \{0,\ldots,K\}$, $t \in \mathcal{T}(\mathcal{A},K)$, $S \in \mathsf{SL}(\mathcal{A},K)$, $i \in \{0,\ldots,l-1\}$, $g_1, g_2 \in G$.

(5) $op = \mathsf{deflate}(\bar{q}, \bar{q}', \bar{p}, p)$, iff $p = \gamma_l$, $\bar{p} = (\gamma_l, j)$, $\bar{q} = (t, j, S, \gamma_l)$, $\bar{q}' = (t, j+1, S, \varepsilon)$, $q = (g_1, \bar{q}, g_2)$, and $q' = g_2$, for some $j \in \{0,\ldots,K-1\}$, $t \in \mathcal{T}(\mathcal{A},K)$, $S \in \mathsf{SL}(\mathcal{A},K)$, $g_1, g_2 \in G$.

(6) $op = \mathsf{deflate}(\bar{q}, \bar{q}', \bar{p}, p)$, iff $p = \gamma_l$, $\bar{p} = (\gamma_l, K)$, $\bar{q} = (t, K, S, \gamma_l)$, $\bar{q}' = \bot$, $q = (g_1, \bar{q}, g_2)$, and $q' = (g_1, \bot, g_2)$, for some $t \in \mathcal{T}(\mathcal{A},K)$, $S \in \mathsf{SL}(\mathcal{A},K)$, $g_1, g_2 \in G$.

(7) $op = \mathsf{burst}(\bot)$ iff $q = (g_1, \bot, g_2)$ and $q = g_2$ for some $g_1, g_2 \in G$.

(8) $op = \delta \in \mathbb{Z}^P$ with $\delta(p) = 0$ for all $p \in P$, iff there is a rule $g_1 \mapsto g_2 \lhd \gamma \in \Delta_r$, $q = g_1 \in G$ and $q' = (g_2, \bar{q}, g_3)$ with $\bar{q} = (t, j, S, \gamma_0)$ where $t \in \mathcal{T}(\mathcal{A},K)$ with a $j$th context switch on symbol $\gamma$ followed by a $j$th segment going from $g_2$ to $g_3$, for some $S \in \mathsf{SL}(\mathcal{A},K)$, $j \in \{1,\ldots,K\}$.

Intuitively, the edge in (1) spawns the initial thread, the edges in (2) remove a thread with cs-number 0 when it becomes active for the first time and the edges in (3) convert it to a balloon, the edges in (4) apply the spawns during the $j$th segment of a thread by transferring them from the balloon to the regular places, the edges in (5) and (6) finalize this application process with the edges in (6) handling the special case of the $K$th segment of a thread, the edges in (7) burst balloons containing no more spawns, and the edges in (8) initialize the application of spawns for a thread with cs-number $\geq 1$ that is already being represented by a balloon.

LEMMA B.1. *The* DCPS *$\mathcal{A}$ has an infinite, progressive, $K$-context switch bounded run iff the* VASSB *$\mathcal{V}$ has an infinite progressive run.*

PROOF. For the *only if direction* let $\rho$ be an infinite, progressive, $K$-context switch bounded run of $\mathcal{A}$. We want to decompose $\rho$ into parts corresponding to a single thread each. To do this we look at the continuous *segments* of $\rho$ where a thread is active, starting from the first configuration with an active thread and ending each segment at a configuration with no active thread, upon which a new segment begins at the very next configuration. We can now group segments together into *executions* of single threads:

- Start with the earliest not yet grouped segment where the active thread has context switch number 0. Note the local configuration that gets added to the bag at the end of this segment.
- Take the very next ungrouped segment in $\rho$ where the active thread at the beginning matches the previously noted local configuration. Again, note the local configuration that gets added to the bag at the end of this segment.
- Repeat the previous step until an empty stack occurs.

Repeating this infinitely often decomposes $\rho$ into infinitely many thread executions of $K + 1$ segments each. This is because due to progressiveness all local configurations have to be resumed

and all thread terminations occur after exactly $K$ context switches. Each execution also corresponds to a thread type $t$, which can be determined by taking note of the global state and top of stack symbol at the start of each segment and the global state at the end of each segment of the execution. Since all executions end in the empty stack, $L_t \neq \varnothing$ for all thread types $t$ that occur on $\rho$ in this way. Let us now construct from $\rho$ an infinite run of $\mathcal{V}$ starting from $c_0 = (q_0, \emptyset, \emptyset)$:

- Start with the edge from (1) to create a token on the place corresponding to the initial stack symbol $\gamma_0$.
- Go over all consecutive segments of $\rho$. If a segment is the first segment of an execution:
  - Start with an edge from (2) to remove a token from place $\gamma$, which is the top of stack symbol this segment starts with. Use as $g_2$ the global state this segment starts with, as $g_3$ the global state this segment ends in, and as $t$ the thread type this segment's execution corresponds to. This leads to a state of $\mathcal{V}$ that contains the semi-linear set $s(t)$.
  - Next, take an edge from (3) to create a balloon whose contents characterize exactly the spawns made by this segment's execution. This is possible because we kept track of $s(t)$ in the state, and the appropriate balloon contents are in this semi-linear set by construction.
  - From here we need to deflate the balloon $|\Gamma|$ times to transfer all the spawns from this segment to the places of $\mathcal{V}$. This is done using the edges from (4) for each $i$ from 0 to $l-1$ on the balloon places $(\gamma_0, 0)$ to $(\gamma_{l-1}, 0)$. The 0 as the second component is because this is the first segment of an execution.
  - If this is not the last segment of its execution, we use an edge from (5) to perform the last deflation on the balloon place $(\gamma_l, 0)$ and go to the global state that this segment ends in, using that state as $g_2$.
  - If it is the last segment of its execution we instead use an edge from (6) perform the same deflation, but go to a state $(g_1, \perp, g_2)$ instead. From there we use the edge from (7) to burst the balloon and finally go to state $g_2$, which this segment ends in.
- If a segment is not the first segment of its execution:
  - Start with an edge from (8) using as $g_2$ the global state this segment starts with, as $g_3$ the global state this segment ends in, and as $\bar{q}$ the balloon state we ended in after handling the previous segment of this execution.
  - From here continue like for a first segment of an execution, deflating the balloon $|\Gamma|$ times in the next few steps. Note that we operate on balloon places $\Gamma \times \{j\}$, where this is the $j$th segment within its execution, instead of $\Gamma \times \{0\}$.

Since each thread execution of $\rho$ has exactly $K$ segments, any balloon will eventually reach the $K$th segment while going over $\rho$. This results in an edge from (7) being used to burst the balloon. Therefore any balloon state occurring in a configuration of $\mathcal{V}$ on the constructed run will be used for a deflate or burst later. Furthermore, each place $\gamma \in \Gamma$ corresponds to a local configuration $(\gamma, 0)$ of $\mathcal{A}$, and we remove a token from it whenever we switch in a thread with that local configuration. Since such local configurations always have to be resumed on $\rho$, the corresponding places will always have tokens removed from them as well. Thus the handling of all balloons and places fulfils the conditions for a progressive run.

The constructed run of $\mathcal{V}$ is also sound: For each segment of $\rho$ it has a sequence of transitions that make the same state change as that segment, and also apply the same spawns. The latter is due to initially choosing the contents during the creation of a balloon correctly. A token representing the initial thread is also created in the beginning of the run. Since the transitions from one segment to the next require the same resumption rules $\Delta_r$ as in $\rho$ (see edges (2) and (8)) these are also possible in $\mathcal{V}$.

For the *if direction* we do a very similar construction, but backwards. Starting with an infinite progressive run of $\mathcal{V}$ from $c_0$, we decompose it into segments that end in a state in $G$. Afterwards we group those segments together by matching the local balloon configurations in the same way we did for local thread configurations before. Then by construction of $L_t$ we know that there is a thread execution of type $t$ that makes the same spawns in each segment as the group of one balloon does. Those executions will all terminate after $K$ context switches, guaranteeing progressiveness, and we can interleave them in the same way as the balloon segments to form an infinite progressive run of $\mathcal{A}$. ∎

## C    PROOFS FOR SECTION 5

*Balloons with Identity.* In order to prove Lemma 5.1, we will work with runs which contain a unique identity for every balloon i.e. with *balloons-with-id*. Formally, a balloon-with-id is a tuple $(b, i)$ where $b$ is a balloon and $i \in \mathbb{N}$. A configuration-with-id $d$ is a tuple $d = (q, \mathbf{m}, v)$ where $v$ is a multiset of balloons-with-id. For a VASSB $\mathcal{V} = (Q, P, \Omega, \Phi, E)$, the edges in $E$ define a transition relation on configurations-with-id. For an edge $q \xrightarrow{op} q'$, and configurations-with-id $d = (q, \mathbf{m}, \mathbf{n})$ and $d' = (q', \mathbf{m}', \mathbf{n}')$, we define $d \xrightarrow{op} d'$:

- If $op = \delta \in \mathbb{Z}^P$ and $\mathbf{m}' = \mathbf{m} + \delta$ and $\mathbf{n}' = \mathbf{n}$.
- If $op = \text{inflate}(\sigma, S)$ and $\mathbf{m}' = \mathbf{m}$ and $\mathbf{n}' = \mathbf{n} + [[((\sigma, \mathbf{k}), i)]]$ for some $\mathbf{k} \in S$ and $i \in \mathbb{N}$. That is, we create a new balloon with state $\sigma$, multiset $\mathbf{k}$ for some $\mathbf{k} \in S$ and arbitrary id $i$.
- If $op = \text{deflate}(\sigma, \sigma', \pi, p)$ and there is a balloon-with-id $(b, i) = ((\sigma, \mathbf{k}), i) \in \Omega \times \mathbb{M}[\Phi]$ with $\mathbf{n}(b, i) \geq 1$ and $\mathbf{m}' = \mathbf{m} + \mathbf{k}(\pi) \cdot [[p]]$ and $\mathbf{n}' = \mathbf{n} - [[(b, i)]] + [[((\sigma', \mathbf{k}'), i)]]$, 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-with-id $((\sigma, \mathbf{k}), i)$ from $\mathbf{n}$, transfer the contents in place $\pi$ from $\mathbf{k}$ to place $p$ in $\mathbf{m}$, and update the state $\sigma$ to $\sigma'$ while retaining its id. Here we say the balloon-with-id $((\sigma, \mathbf{k}), i)$ was *deflated*.
- If $op = \text{burst}(\sigma)$ and there is a balloon-with-id $(b, i) = ((\sigma, \mathbf{k}), i) \in \Omega \times \mathbb{M}[\Phi] \times \mathbb{N}$ with $\mathbf{n}(b, i) \geq 1$ and $\mathbf{m}' = \mathbf{m}$ and $\mathbf{n}' = \mathbf{n} \ominus [[(b, i)]]$. This means we pick some balloon-with-id $(b, i)$ with state $\sigma$ from our multiset $\mathbf{n}$ of balloons-with-id and remove it, making any tokens still contained in its balloon places disappear as well. Here we say the balloon-with-id $(b, i)$ is *burst*.

We often simply say 'the balloon $i$' was burst (or deflated) when balloon identities are unique. A *run-with-id* $\tau = d_0 \xrightarrow{op_1} d_1 \xrightarrow{op_1} d_2 \cdots$ is a finite or infinite sequence of configurations-with-id. We note that a semiconfiguration-with-id is just a semiconfiguration since there are no balloons.

*Definition C.1.* We associate a *canonical run-with-id* $\tau = d_0 \xrightarrow{op_1} d_1 \xrightarrow{op_1} d_2 \cdots$ to any given run $\rho = c_0 \xrightarrow{op_1} c_1 \xrightarrow{op_2} \cdots$ as follows:

- If $op_i$ creates a balloon $b$ in $\rho$, then it creates $(b, i)$ in $\tau$. This implies that balloons are assigned unique id's since every operation creates only one balloon.
- If $op_i$ deflates (resp. bursts) a balloon $b$ in $\rho$, then it deflates (resp. bursts) the balloon-with-id $(b, j)$ in $\tau$ where $j = min\{k \mid d_{i-1}.v(b, k) \geq 1\}$.

We observe that the multiset $d_i.v$ of any configuration-with-id $d_i$ in a canonical run-with-id $\tau$ is infact a set. The set of id's in $\tau$ is denoted $I(\tau)$. A collection of balloons-with-id with the same id $i_0$ may be viewed as a particular balloon which undergoes a sequence of operations $\text{seq}_{i_1} = op_{i_0}, op_{i_1}, op_{i_2}, \cdots$. The balloons $(b_j, i_0)$ resulting from the operations $op_{i_j}$ are associated with the id $i_0$ and we will perform surgery on a run-with-id $\tau$ resulting in a modified run $\tau'$ by replacing the balloon $(b_1, i_0)$ at its point of inflation $op_{i_0}$ by $(b'_1, i_0)$ in $\tau'$, with the implicit assumption that the sequence of operations $\text{seq}'_{i_0}$ in the modified run-with-id $\tau'$ now act on $(b'_1, i_0)$ instead.

Note that we use the notation $\mathrm{seq}_i'$ to represent the sequence of operations on id $i$ occuring in the modified run $\tau'$. The sequences $\mathrm{seq}_{i_0}$ and $\mathrm{seq}_{j_0}$ are disjoint for $i_0 \neq j_0$ and thus the set of sequences $\{\mathrm{seq}_i \mid i \in \tau\}$ form a partition of the indices $I_B(\tau) = \{i \mid op_i \text{ is a balloon operation }\}$. We will write $i.\sigma, i.\mathbf{k}$ in short for $(b, i).\sigma$ and $(b, i).\mathbf{k}$ where $(b, i)$ is created by $op_i$. The linear set used to create a balloon with the id $i$ is denoted $L_i$. A run-with-id $\tau$ *corresponds* to a run $\rho$ if $\rho$ is obtained from $\tau$ by removing id's. For a balloon $b$ in $\rho$, the set $\{(b, i) \mid i \in I(\tau)\}$ of balloons-with-id in a corresponding $\tau$ is called the *balloon class* of $b$.

The fact that reachability remains preserved whether considering runs or runs-with-id immediately follows from the definition of a canonical run-with-id:

PROPOSITION C.2. *Given $\mathcal{V}$ and its semiconfigurations $s_1, s_2$, there exists a run $\rho = s_1 \xrightarrow{*} s_2$ of $\mathcal{V}$ iff there exists a corresponding run-with-id $\tau = s_1 \xrightarrow{*} s_2$.*

The fact that progressivenss is also preserved is also not difficult to see. We first define the obvious notion of progressiveness for a run-with-id.

*Definition C.3.* A progressive run-with-id $\tau = d_0 \xrightarrow{op_1} d_1 \xrightarrow{op_1} d_2 \cdots$ is one such that:

- for every balloon-with-id $(b, i) \in d_j$ for some $d_j \in \tau$, there exists $d_k$ with $k > j$ such that $d_k \xrightarrow{op_k} d_{k+1}$ where $op_k$ is a deflate or burst operation on id $i$, and
- for every $p \in P$ if $d_i.\mathbf{m}(p) > 0$ then there exists $j > i$ such that $d_j \xrightarrow{\delta} d_{j+1}$ where $\delta(p) < 0$.

The following proposition follows from the above definition.

PROPOSITION C.4. *In a progressive run-with-id $\tau$, for any id $i$, one of the following is true:*

(1) $\mathrm{seq}_i$ *is finite and the final operation in $\mathrm{seq}_i$ is a burst operation, or*
(2) $\mathrm{seq}_i$ *is infinite.*

If $\mathrm{seq}_i$ is finite for all $i$, we say 'every balloon is burst in $\tau'$'. The existence of progressive runs and progressive runs-with-id also coincide.

PROPOSITION C.5. *For any given $\mathcal{V}$ and its semiconfiguration $s$, there exists a progressive run $\rho$ from $s$ iff there exists a corresponding progressive run-with-id $\tau'$ from $s$.*

PROOF. Clearly a progressive run $\rho$ can be obtained from a progressive run-with-id $\tau$ by id-removal. For the converse direction, consider the canonical $\tau = d_0 \xrightarrow{op_1} d_1 \xrightarrow{op_1} d_2 \cdots$ corresponding to $\rho = c_0 \xrightarrow{op_1} c_1 \xrightarrow{op_2} c_2 \cdots$. The canonical $\tau$ satisfies all progressiveness conditions except for one special case: a balloon $b$ in $\rho$ which undergoes an infinite sequence of trivial deflates at $j_1, j_2, \cdots$ may result in $\tau$ which contains multiple balloons-with-id in the balloon class of $b$ at every configuration $d_{j_1}, d_{j_2}, \cdots$. By construction, $\tau$ always chooses the least id and hence $\tau$ is not progressive for other id's in the balloon class of $b$. The progressive $\tau'$ is obtained by modifying $\tau$ to 'dovetail' through all possible choices in the balloon class of $b$ (i.e. if the id's in the balloon class are $i_1 < i_2 < i_3 \cdots$, then the choices made are $i_1, i_1, i_2, i_1, i_2, i_3, i_1 \cdots$). We note that $\tau'$ agrees with $\tau$ on all inflate and burst operations. ∎

We now consider the two properties assumed in the proof of Lemma 5.1, namely that of being zero-base and being typed and show that it is possible to construct a VASSB $\mathcal{V}'$ with these properties from a given VASSB $\mathcal{V}$ while preserving reachability and progressiveness.

*Balloon Types and Surgeries on Balloons.* A deflate operation transferring a non-zero number of tokens is called a *non-trivial* deflate, otherwise it is a trivial deflate. Clearly, a non-trivial deflate moving tokens from a balloon place $\pi$ must be the first deflate which transfers tokens from $\pi$, motivating the following definition. For an id $i$, define

$$\text{typeseq}_i = \{(\pi, p, j) \mid \exists \sigma, \sigma' \in \Omega \; \exists j \in \text{seq}_j : op_j = \text{deflate}(\sigma, \sigma', \pi, p) \text{ and}$$

$$\forall \sigma'', \sigma''' \in \Omega \; \forall p' \in P \; \forall j' \in \text{seq}_i, j' < j : op_{j'} \neq \text{deflate}(\sigma'', \sigma''', \pi, p')\}$$

We write $i <_t j$ for id's $i, j$ if $i < j$, $L_i = L_j$ where $L_i$ (resp. $L_j$) is the linear set used to inflate balloon $i$ (resp. $j$) and for every $(\pi, p) \in \Phi \times P$, there exists $k \in \text{seq}_i$ such that $(\pi, p, k) \in \text{typeseq}_i$ iff there exists $k' \in \text{seq}_j$ with $k' > k$ such that $(\pi, p, k') \in \text{typeseq}_j$.

A *deflate-sequence* $S = (\pi_1, p_1), (\pi_2, p_2), \cdots, (\pi_n, p_n)$ is a finite sequence of elements from $\Phi \times P$ which satisfies the property that for all $i, j \in [1, n]$, we have $\pi_i \neq \pi_j$. We write $n = |S|$ and also write $\text{set}(S)$ for the set of tuples $(\pi_i, p_i)$ obtained from $S$ by ignoring the order. A marked deflate-sequence $M = (S, i)$ is a tuple consisting of a deflate-sequence $S$ and $i \in [0, \cdots, |S|]$. We write $M.S$ and $M.i$ for the two components of $M$. If $M' = (S, i+1)$ and $M = (S, i)$, we write $M' = M + 1$. The set of all marked deflate-sequences is denoted by $\mathcal{M}$.

Let $\text{typeseq}_i \! \downarrow = (\pi_1, p_1), (\pi_2, p_2), \cdots, (\pi_n, p_n)$ be the deflate-sequence obtained by projecting $\text{typeseq}_i$ to the first two components, in increasing order of the third component. We write $k = \text{typeseq}_i(j)$ if $(\pi_j, p_j, k) \in \text{typeseq}_i$. We write $i \sim_t j$ if $\text{typeseq}_i \! \downarrow = \text{typeseq}_j \! \downarrow$ and $L_i = L_j$. A tuple $t = (L_i, \text{typeseq}_i \! \downarrow)$ is called a *type* and only depends on $\mathcal{V}$. The set of finitely many types associated with $\mathcal{V}$ is denoted $\mathcal{T}_{\mathcal{V}}$; we drop the subscript when $\mathcal{V}$ is clear from the context.

Given any VASSB $\mathcal{V} = (Q, P, \Omega, \Phi, E_p \cup E_n \cup E_d \cup E_b)$, its *typed extension* $\mathcal{V}' = (Q, P, \Omega', \Phi, E_p \cup E_n' \cup E_d' \cup E_b')$ is given by $\Omega' = \Omega \times \mathcal{M}$ and set of balloon edges is given as:

(1) For each $e = q \xrightarrow{\text{inflate}(\sigma, L)} q'$ in $E_n$, for each $M \in \mathcal{M}$ such that $M.i = 0$, add
$q \xrightarrow{\text{inflate}((\sigma, M), L)} q'$ to $E_n'$,

(2) for each $e = q \xrightarrow{\text{deflate}(\sigma, \sigma', \pi, p)} q'$ in $E_d$ and for each $M \in \mathcal{M}$ such that $(\pi, p) = (\pi_j, p_j) \in M$ for some $j \in [1, \cdots, |M.S|]$
  (a) if $j \leq M.i$ then add
$q \xrightarrow{\text{deflate}((\sigma, M), (\sigma', M), \pi, p)} q'$ to $E_d'$,
  (b) else if $j = (M.i) + 1$ then add $q \xrightarrow{\text{deflate}((\sigma, M), (\sigma', M+1), \pi, p)} q'$ to $E_d'$, and

(3) for each $e = q \xrightarrow{\text{burst}(\sigma)} q'$ in $E_b$ and for each $M \in \mathcal{M}$ with $M.i = |M.S|$, add $q \xrightarrow{\text{burst}(\sigma, M)} q'$ to $E_b'$.

LEMMA C.6. *Given any VASSB $\mathcal{V}$ and two semiconfigurations $s_1, s_2$ of $\mathcal{V}$, its typed extension VASSB $\mathcal{V}'$ satisfies the following properties:*

*(1) $\mathcal{V}$ has a progressive run starting from $s_1$ iff $\mathcal{V}'$ has a progressive run starting from $s_1$ and*
*(2) $(\mathcal{V}, s_1, s_2) \in \text{REACH}$ iff $(\mathcal{V}', s_1, s_2) \in \text{REACH}$.*

PROOF. Given a run $\rho' = c_0' \xrightarrow{op_1'} c_1' \xrightarrow{op_2'} \cdots$ of $\mathcal{V}'$ starting from a semiconfiguration $c_0'$, it is clear that there exists a run $\rho = c_0' \xrightarrow{op_1} c_1 \xrightarrow{op_2} \cdots$ of $\mathcal{V}$ where for each $k \geq 1$,

- if $op_k' \in E_p$ then $op_k = op_k'$,
- else if $op_k'$ creates a balloon $b'$ with state $(\sigma, M)$, then $op_k$ creates a balloon $b$ such that $b.\mathbf{k} = b'.\mathbf{k}$ and $b.\sigma = \sigma$,
- otherwise $op_k'$ is a deflate or burst operation on a balloon $b'$ with state $(\sigma, M)$ and $op_k$ applies the corresponding deflate or burst operation to a balloon $b$ with $b.\mathbf{k} = b'.\mathbf{k}$ and $b.\sigma = \sigma$.

By induction on length of the run, we see that $c_k.\mathbf{m} = c'_k.\mathbf{m}$ for each $k \geq 0$ and further, there is a bijection between $c_k.v$ and $c'_k.v$ which preserves the balloon contents with the state of a balloon in $c_k$ obtained by projecting the state of the corresponding balloon to the first component. If $\rho'$ is a finite run ending in a semiconfiguration, this implies that $\rho$ also ends in the same semiconfiguration. Since $\rho'$ is progressive with respect to balloon states of the form $(\sigma, M)$, clearly $\rho$ is progressive with respect to the balloon states $\sigma$.

For the converse direction, we argue using runs-with-id. Let $\tau = d_0 \xrightarrow{op_1} d_1 \xrightarrow{op_2} \cdots$ be the canonical run-with-id of $\mathcal{V}$ starting from a semiconfiguration $d_0$ for a given $\rho$ of $\mathcal{V}$. We define the run-with-id $\tau' = d_0 \xrightarrow{op'_1} d_1 \xrightarrow{op'_2} \cdots$ of $\mathcal{V}'$ where for each $k \geq 1$,

- if $op_k \in E_p$ then $op'_k = op_k$,
- if $op_k \in E_n$ creates $(b, i)$ then $op'_k$ creates $(b', i)$ where $b'.\sigma = (b.\sigma, M)$ with $M = (\text{typeseq}_i \downarrow, 0)$ and $b'.\mathbf{k} = b.\mathbf{k}$, and
- if $op_k \in E_d \cup E_b$ and $op_k \in \text{seq}_i$ for some id $i$, then $op'_k \in \text{seq}'_i$. Here $\text{seq}'_i$(resp. $\text{seq}_i$) refers to the sequence of operations on id $i$ in $\tau'$ (resp. $\tau$).

If $\tau$ is a finite run between semiconfigurations $s_1$ and $s_2$, then so is $\tau'$ and thus we have (2) of the lemma. If $\tau$ is a progressive run-with-id from $s_1$ then so is $\tau'$ and by Proposition C.5, we conclude (1) of the lemma. ∎

In the remainder of this section, we assume that any VASSB is typed.
A VASSB $\mathcal{V}$ is said to be *zero-base* if every linear set in $\mathcal{V}$ has base vector $\emptyset$.

LEMMA C.7. *Given any VASSB $\mathcal{V}$, we can construct a zero-base VASSB $\mathcal{V}'$ such that for any two semiconfigurations $s_1, s_2$ of $\mathcal{V}$,*

*(1) $\mathcal{V}$ has a progressive run starting from $s_1$ iff $\mathcal{V}'$ has a progressive run starting from $s_1$ and*
*(2) $(\mathcal{V}, s_1, s_2) \in \text{REACH}$ iff $(\mathcal{V}', s_1, s_2) \in \text{REACH}$.*

PROOF. We may assume that there exists a unique linear set $L_\sigma$ associated with any given balloon state $\sigma$ of $\mathcal{V}$ such that any balloon $b$ inflated with state $\sigma$ has $b.\mathbf{k} \in L_\sigma$, by applying the following modification to $\mathcal{V}$. Let $\mathcal{L}$ be the finite set of linear sets used in $\mathcal{V}$. We replace the set of balloon states $\Omega$ by the cartesian product $\Omega \times \mathcal{L}$ and for each inflate operation $\text{inflate}(\sigma, L)$ we use $\text{inflate}((\sigma, L), L)$ in the modified VASSB. This modification is easily seen to preserve progressiveness and reachability. We also assume that the given $\mathcal{V}$ is typed and instead of writing balloon states as tuples $(\sigma, M)$, we assume that every state $\sigma$ comes with an associated $M_\sigma$.

Let $\mathcal{V} = (Q, P, \Omega, \Phi, E_p \cup E_n \cup E_d \cup E_b)$ and $\mathbf{b}_\sigma$ be the base vector of a balloon state $\sigma \in \Omega$. $\mathcal{V}' = (Q', P, \Omega', \Phi, E'_p \cup E'_n \cup E'_d \cup E_b)$ is constructed from $\mathcal{V}$ as follows:

(1) $Q' = Q \cup E_d \times P$,
(2) $\Omega' = \Omega$ contains a state $\sigma'$ for each $\sigma$ such that $L_{\sigma'}$ is obtained from $L_\sigma$ by removing the base vector $\mathbf{b}_\sigma$,
(3) $E'_p = E_p \cup E''_p$, where $E''_p$ contains for each edge $e = q \xrightarrow{\text{deflate}(\sigma_1, \sigma_2, \pi, p)} q'$ in $E_d$ with $M_{\sigma_1} \neq M_{\sigma_2}$, the edge $(e, p) \xrightarrow{\delta} q'$ where $\delta(p) = \mathbf{b}_\sigma(p)$ and $\delta(p') = 0$ for $p' \neq p$,
(4) $E'_n$ contains an edge $q \xrightarrow{\text{inflate}(\sigma', L_{\sigma'})} q'$ for each edge $q \xrightarrow{\text{inflate}(\sigma, L_\sigma)} q'$ in $E_n$, and
(5) $E'_d$ contains the following edges for each $e = q \xrightarrow{\text{deflate}(\sigma_1, \sigma_2, \pi, p)} q'$ in $E_d$:
  (a) if $M_{\sigma_1} \neq M_{\sigma_2}$ then we add $q \xrightarrow{\text{deflate}(\sigma'_1, \sigma'_2, \pi, p)} (e, p)$ to $E'_d$,
  (b) else we add $q \xrightarrow{\text{deflate}(\sigma'_1, \sigma'_2, \pi, p)} q'$ to $E'_d$.

Given a finite run $\rho = c_0 \rightarrow c_1 \rightarrow c_2 \cdots c_k$ of $\mathcal{V}$ between semiconfigurations, there exists a finite run $\rho' = c_0' \xrightarrow{*} c_1' \xrightarrow{*} c_2' \cdots c_k'$ where for every inflate$(\sigma, L)$ operation in $\rho$, we perform a inflate$(\sigma', L)$ operation in $\rho'$, creating a balloon which is identical except for the base vector. The missing base vector tokens are then transferred to the appropriate place via edges of the form $(e, p) \xrightarrow{\delta} q'$ from (2). Note that the extra base-vector tokens are only added during non-trivial deflates, which is information that can be obtained from the type information in $M_\sigma$.

Conversely if $\rho'$ is a a finite run of $\mathcal{V}'$ between semiconfigurations of $\mathcal{V}$, we observe that the states use to add the missing tokens are sandwiched between the states of a deflate operation and we obtain a run $\rho$ from $\rho'$ by replacing the inflate$(\sigma', L)$ operations by inflate$(\sigma, L)$ operations and replacing every chain of edges $q \rightarrow (e, p) \rightarrow q'$ by the corresponding deflate edge $q \rightarrow q'$ of $\mathcal{V}$. Thus reachability is preserved. It is easy to see that the construction also preserves progressiveness since any infinite run of $\mathcal{V}'$ can also be decomposed into segments which are runs between semiconfigurations of $\mathcal{V}$. ∎

Henceforth we assume that any VASSB is zero-base.

*Token-shifting Surgery.* Fix a run-with-id $\tau = d_0 \xrightarrow{op_1} d_1 \xrightarrow{op_2} d_2 \cdots$ of a zero-base VASSB $\mathcal{V}$. Let $i$ be an id and $I$ a finite set of id's with the property that for each $j \in I$ we have $i <_t j$. Let $N = \sum_{\mathbf{k} \in K} \mathbf{k}$ where $K = \{\mathbf{k} \mid \exists j \in I, op_j \text{ creates balloon } j \text{ with } \mathbf{k} = j.\mathbf{k}\}$. The token-shifting surgery $\mathcal{S}_{i \leftarrow I}$ on $\tau$ yielding $\mathcal{S}_{i \leftarrow I}(\tau) = \tau' = d_0' \xrightarrow{op_1'} d_1' \xrightarrow{op_2'} d_2' \cdots$ is defined as, for each $k \geq 1$:

(1) If $k \notin I \cup \{i\}$, then $op_k' = op_k$,
(2) else if $k = i$ then if $op_i$ creates $(b, i)$ then $op_i'$ creates $(b', i)$ where $b'.\sigma = b.\sigma$ and $b'.\mathbf{k} = b.\mathbf{k} + N$,
(3) else $k \in I$ and if $op_k$ creates $(b_k, k)$ then $op_k'$ creates $(b', k)$ where $b'.\sigma = b.\sigma$ and $b'.\mathbf{k} = \emptyset$.

In other words, the only difference between $\tau'$ and $\tau$ is the content of balloons created at $I \cup \{i\}$ and the resulting changes in the configurations.

PROPOSITION C.8. *For any $\tau = d_0 \xrightarrow{op_1} d_1 \xrightarrow{op_2} d_2 \cdots$ and any $\mathcal{S}_{i \leftarrow I}$, the run $\mathcal{S}_{i \leftarrow I}(\tau) = \tau' = d_0' \xrightarrow{op_1'} d_1' \xrightarrow{op_2'} d_2' \cdots$ is a valid run. Further, if $\tau$ is a progressive run from $s$ then so is $\mathcal{S}_{i \leftarrow I}(\tau)$. If $\tau = s \xrightarrow{*} s'$ is a finite run between two semiconfigurations, then $\mathcal{S}_{i \leftarrow I}(\tau)$ is also a run from $s$ to $s'$.*

PROOF. The fact that $\mathcal{V}$ is zero-base ensures that $b'.\mathbf{k}$ as defined in (2) belongs to the linear set $L_i$. The condition $i <_t j$ for each $j \in I$ implies that the balloon $i$ makes its non-trivial deflates before any of the balloons with an id from $I$. This implies that for each $k \geq 1$ we have $d_k.\mathbf{m} \leq d_k'.\mathbf{m}$ since the total number of tokens transferred from $\pi$ to $p$ for any $\pi \in \Phi, p \in P$ in $\tau'$ remains greater than that in $\tau$ for every prefix $\tau[0, k]$. By monotonicity, every operation remains enabled in $\tau'$. Progressiveness is preserved since the only change is in the content of some of the newly created balloons. Since the total number of tokens transferred in the entire run is the same in $\tau$ and $\tau'$ for each $(\pi, p)$ in a finite run between two semiconfigurations $s, s'$, and there are no balloons in $s'$, this implies that $\tau'$ reaches the same semiconfiguration $s'$. ∎

## C.1 Proof of Lemma 5.1

We have now introduced all of the machinery required to prove Lemma 5.1. We will construct $\mathcal{V}'$ from $\mathcal{V}$ such that the following are equivalent:

(S1) there exists a progressive run-with-id $\tau = s \rightarrow d_1 \rightarrow d_2 \cdots$ of $\mathcal{V}$,
(S2) there exists a progressive run-with-id $\tau' = s \rightarrow d_1' \rightarrow d_2' \cdots$ of $\mathcal{V}'$ which bursts every balloon, and
(S3) there exists an $A, B$-witness $\rho_{A,B}' = s \xrightarrow{*} c \xrightarrow{*} c'$ of $\mathcal{V}'$ for some $A \subseteq P', B \subseteq \Omega'$.

We will first show (S1) iff (S2) and then show (S2) iff (S3). By Lemmas C.7 and C.6 we assume that VASSB $\mathcal{V}$ is zero-base and typed, and that every $\sigma \in \Omega$ comes with an associated marked deflate-sequence $M_\sigma$ and do not write balloon states explicitly as $(\sigma, M_\sigma)$. From $\mathcal{V} = (Q, P, \Omega, \Phi, E_p \cup E_n \cup E_d \cup E_b)$, we construct $\mathcal{V}' = (Q', P', \Omega, \Phi, (E_p \cup E_p'') \cup E_n \cup E_d' \cup E_b')$ as follows:

- $Q' = Q \cup (E \times \{0, 1\})$,
- $P' = P \cup (\Omega \times \{0, 1\})$,
- $E'$ is defined as follows:
  (1) For each $e = q \xrightarrow{\mathsf{deflate}(\sigma, \sigma', \pi, p)} q'$ in $E_d$ where $(\pi, p) = (\pi_j, p_j) \in M_\sigma$,
      (a) if $M_\sigma.i \le |M_\sigma.S| - 1$ then add,
          $q \xrightarrow{\mathsf{deflate}(\sigma, \sigma', \pi, p)} q'$ to $E_d'$,
      (b) else if $M_\sigma.i = |M_\sigma.S|$,
          (i) $q \xrightarrow{\mathsf{deflate}(\sigma, \sigma', \pi, p)} q'$ to $E_d'$,
          (ii) $q \xrightarrow{\mathsf{deflate}(\sigma, \sigma', \pi, p)} (e, 0)$ to $E_d'$,
          (iii) $(e, 0) \xrightarrow{\mathsf{burst}(\sigma')} (e, 1)$ to $E_b'$,
          (iv) $(e, 1) \xrightarrow{\sigma'^+} q'$ to $E_p''$.
          (v) if $\sigma = \sigma'$, for $(i, j) \in \{(0, 1), (1, 0)\}$ add $q \xrightarrow{(\sigma, i)^+ (\sigma, j)^-} q'$ to $E_p''$,
          (vi) else for each $i, j \in \{0, 1\}$ add $q \xrightarrow{(\sigma, i)^- (\sigma', j)^+} q'$ to $E_p''$
  (2) For each $e = q \xrightarrow{\mathsf{burst}(\sigma)} q'$ in $E_b$, add $q \xrightarrow{\mathsf{burst}(\sigma)} q'$ to $E_b'$.

For $k, k' \in \mathsf{seq}_i$, we write $k \lessdot_i k'$ to denote the successor relation in $\mathsf{seq}_i$. Let $m$ be the smallest id in $\tau$ such that $\mathsf{seq}_m$ is infinite. Since there can be at most finitely many indices in $\mathsf{seq}_m$ where a non-trivial deflate occurs, there exists $n \in \mathsf{seq}_m$ which is the last such index. Intuitively, $\mathcal{V}'$ uses the edges in (1 b ii) to (1 b vi) to burst the balloon with id $m$ at operation $n$ where the last non-trivial deflate occurs and uses a VASS token to simulate the effect of the infinite suffix of $\mathsf{seq}_m$ which performs only trivial deflates. We use the notation $p^+$(resp. $p^-$) to indicate that the place $p$ is incremented (resp. decremented). For example, for $p_1, p_2 \in P$, we write $p_1^+$, $p_2^-$, and $p_1^- p_2^+$ for the vectors mapping $p_1$ to +1 and all other places to 0, $p_2$ to −1 and all other places to 0, and $p_1$ to −1, $p_2$ to +1, and all other places to 0, respectively.

From the given progressive $\tau = s \xrightarrow{op_1} d_1 \xrightarrow{op_2} d_2 \cdots$ of $\mathcal{V}$, we construct a progressive $\tau_1' = d_0' \xrightarrow{*} d_{i_1}' \xrightarrow{*} d_{i_2}' \cdots$ for $\mathcal{V}'$. We set $d_0' = s$ and for every $k$, the run $d_{i_{k-1}}' \xrightarrow{*} d_{i_k}'$ is defined as follows:

(I) If $k \notin \mathsf{seq}_m$ or $k < n$ then
    (a) if $op_k = \delta$ then $i_k = i_{k-1} + 1$ and $d_{i_{k-1}}' \xrightarrow{\delta} d_{i_k}'$,
    (b) if $op_k = \mathsf{inflate}(\sigma, L)$ creates balloon $(b, k)$, then $i_k = i_{k-1} + 1$ and
        $d_{i_{k-1}}' \xrightarrow{\mathsf{inflate}(\sigma, L)} d_{i_k}'$ creates $(b', i_k)$ with $b'.k = b.k$,
    (c) else if $op_k = \mathsf{deflate}(\sigma, \sigma', \pi, p)$ deflates balloon $k'$, then $i_k = i_{k-1} + 1$ and
        $d_{i_{k-1}}' \xrightarrow{\mathsf{deflate}(\sigma, \sigma', \pi, p)} d_{i_k}'$ deflates balloon $i_{k'}$,
    (d) else $op_k = \mathsf{burst}(\sigma)$ bursts balloon $k'$, then $i_k = i_{k-1} + 1$ and
        $d_{i_{k-1}}' \xrightarrow{\mathsf{burst}(\sigma)} d_{i_k}'$ bursts balloon $i_{k'}$,
(II) otherwise we have $k \in \mathsf{seq}_m$ and $k \ge n$, in which case,
    (a) if $k = n$ and $op_k = \mathsf{deflate}(\sigma, \sigma', \pi, p)$ deflates $m$, then $i_n = i_{n-1} + 3$ and
        $d_{i_{n-1}}' \xrightarrow{op_{i_{n-1}+1}'} d_{i_{n-1}+1}' \xrightarrow{op_{i_{n-1}+2}'} d_{i_{n-1}+2}' \xrightarrow{op_{i_{n-1}+3}'} d_{i_{n-1}+3}'$ where

- $op'_{i_{n-1}+1} = \mathrm{deflate}(\sigma, \sigma', \pi, p)$ deflates balloon $i_m$ where $n' <_m n$,
- $op'_{i_{n-1}+2} = \mathrm{burst}(\sigma)$, and
- $op'_{i_{n-1}+3} = (\sigma, 0)^+$,

(b) else if $k > n$ and $op_k = \mathrm{deflate}(\sigma, \sigma', \pi, p)$ then $i_k = i_{k-1} + 1$ and

   (i) if $\sigma = \sigma'$ and $\exists k' <_m k$ such that $op'_{i_{k'}} = (\sigma, 1)^+(\sigma, 0)^-$ then
   $op'_{i_k} = (\sigma, 0)^+(\sigma, 1)^-$ else
   $op'_{i_k} = (\sigma, 1)^+(\sigma, 0)^-$,

   (ii) else if $\sigma \neq \sigma'$ and $k' <_m k$ is such that $op'_{i_{k'}} = (\sigma, 0)^+(\sigma', 0)^-$ then
   $op'_{i_k} = (\sigma, 0)^+(\sigma', 0)^-$ else
   $op'_{i_k} = (\sigma, 1)^+(\sigma', 1)^-$.

We repeat the construction starting with $\tau_1$, picking the next higher index $m_1 > m$ such that $\mathrm{seq}(i_1)$ is infinite and obtain a sequence of runs $\tau_1, \tau_2, \tau_3 \cdots$ which agree on increasingly larger prefixes (since the id's are chosen in order). Thus we may define $\tau'$ as the limit of this sequence having the property that every balloon is burst in $\tau'$. This implies that $\tau'$ satisfies the progressiveness conditions for all places $p \in P$ and all balloons. Our choices in (II b i) and (II b ii) imply that $\tau'$ also satisfies progressiveness conditions for the places $\Omega \times \{0, 1\}$.

Conversely, consider an arbitrary progressive run-with-id $\tau'$ of $\mathcal{V}'$ which bursts all balloons. It may be decomposed as $\tau' = d'_0 \xrightarrow{*} d'_{i_1} \xrightarrow{*} d'_{i_2} \cdots$ where each segment $d'_{i_{k-1}} \xrightarrow{*} d'_{i_k}$ is either an edge from $E_p$, (1 a),(1 b i),(1 b v),(1 b vi) or (2), or a sequence of edges (1 b ii,1 b iii,1 b iv). Choosing the balloon with least index $m$ such that there exists a burst operation (1 b iii), we know there exists an infinite sequence of corresponding token transfers (1 b v,1 b vi) by progressiveness. By construction, there exist trivial deflates (1 b i) corresponding to these token transfers and by replacing the transfers by (1 b i) edges, we obtain $\tau'_1$.

Repeating the construction with $\tau'_1$ we obtain a sequence of runs $\tau'_1, \tau'_2, \cdots$ which agree on increasingly larger prefixes and we obtain a progressive run $\tau''$ of $\mathcal{V}'$ as the limit of these runs. Since no edges from (1 b ii) to (1 b vi) exist in $\tau''$, this implies that $\tau''$ is infact a progressive $\tau$ of $\mathcal{V}$. This concludes the proof that (S1) iff (S2).

We now take up (S2) iff (S3), beginning with the only-if direction. Let $\tau' = d'_0 \to d'_1 \to d'_2 \cdots$ be a progressive run of $\mathcal{V}'$ which bursts every balloon. We explain the idea behind constructing the required witness for (S3): We pick a set of id's $I_1$ which contains one balloon for each $\sigma' \in \Omega'$. We then inductively pick sets $I_k$ for each $k > 1$ such that all balloons in $I_{k-1}$ have been burst before the creation of any balloon in $I_k$. This divides $\tau'$ into segments such that the tokens of all balloons created in the $k^{th}$ segment excluding those from $I_k$ can be shifted to the balloons in $I_{k-1}$. In particular, this creates a infinite sequence of configurations which contain only empty balloons, which can be used to construct the required witness. Recall that $I(\tau)$ denotes the set of id's in $\tau$ while $I_B(\tau)$ denotes the indices of the operations in $\tau$ which are balloons operations. Formally, for each $\sigma' \in \Omega'$ we define
$i_{1,\sigma'} = \min\{i \in I(\tau') \mid op_i = \mathrm{inflate}(\sigma', L_{\sigma'})\}$,
$I_1 = \{i_{1,\sigma'} \mid \sigma' \in \Omega'\}$ and
$B_1 = \{j_{1,\sigma'} \in I_B(\tau') \mid \exists i_{1,\sigma'} \in I_1, op_{j_{1,\sigma'}} = \mathrm{burst}(\sigma'), j_{1,\sigma'} \in \mathrm{seq}_{i_{1,\sigma'}}\}$.
Inductively, we define for $k > 1$:
$i_{k,\sigma'} = \min\{l \in I(\tau') \mid op_l = \mathrm{inflate}(\sigma', L_{\sigma'}), l > \max(B_{k-1})\}$,
$I_k = \{i_{k,\sigma'} \mid \sigma' \in \Omega'\}$ and
$B_k = \{j_{k,\sigma'} \in I_B(\tau') \mid \exists i_{k,\sigma'} \in I_k, op_{j_{k,\sigma'}} = \mathrm{burst}(\sigma'), j_{k,\sigma'} \in \mathrm{seq}_{i_{k,\sigma'}}\}$.
We also define, for $k \geq 1$,
$I_{k,k+1,\sigma'} = \{l \in I(\tau') \mid l \notin I_{k+1}, op_l = \mathrm{inflate}(\sigma', L_{\sigma'})$ and $\max(B_k) < l < \max(B_{k+1})$ and
$I_{k,k+1} = \bigcup_{\sigma' \in \Omega'} I_{k,k+1,\sigma'}$.

We note that by construction, for each $k \geq 1$, the id $i_{k,\sigma'}$ and the set of id's $I_{k,k+1,\sigma'}$ satisfy the requirements for a token-shifting surgery. Applying the surgeries $\mathcal{S}_{i_{k,\sigma'} \leftarrow I_{k,k+1,\sigma'}}$ to $\tau'$, we obtain $\tau'' = d_0'' \rightarrow d_1'' \rightarrow d_2'' \cdots$ which bursts every balloon. Therefore there exists $n \in \mathbb{N}$ such that for any id $i$ satisfying $i < \max(B_1)$, balloon $i$ is burst before $d_n''$. By construction, for every $k > n$ the configuration $d_{\max(B_k)}''$ contains only empty balloons i.e. for every $(b, j)$ such that $d_{\max(B_k)}''.v(b, j) = 1$, we have $(b, j).\mathbf{k} = \emptyset$.

Let $S = d_{j_1}'', d_{j_2}'', \cdots$ be the infinite sequence of configurations in $\tau''$ such that they only contain empty balloons i.e. satisfy condition (3) of an $A, B$-witness. There exists an infinite subsequence $S' = d_{k_1}'', d_{k_2}'', \cdots$ of $S$ such that the corresponding id-removals $c_{k_1}'', c_{k_2}'', \cdots$ satisfy $c_{k_1}'' \leq c_{k_2}'' \leq \cdots$ due to the fact that $\leq$ is a Well-Quasi-Order. Further, by the Pigeonhole principle, we may assume that they all have the same set of occupied places and same set of balloon states $B$, thus satisfying condition (5) of an $A, B$-witness. We pick $A = P_\infty = \{p \in P \mid \forall i \geq 0 \exists j > i : d_j''.\mathbf{m}(p) > 0\}$, ensuring that $\mathrm{supp}(c_{k_l}'') \subseteq A$ as required. By progressiveness, there exists $k_m, k_n$ with $k_m < k_n$ such that conditions (2) and (4) of an $A, B$-witness are satisfied for $c_{k_m}$ and $c_{k_n}$. Thus the run $\rho'' = c_0 \xrightarrow{*} c_{k_m}'' \xrightarrow{*} c_{k_n}''$ is the required $A, B$-witness.

Conversely, given an $A, B$-witness $\rho'' = c_0 \xrightarrow{*} c \xrightarrow{*} c'$ let $\rho' = c_0 \xrightarrow{*} c_1 \xrightarrow{*} c_2 \xrightarrow{*} c_3 \cdots$ be its 'unrolling' where $c_1 = c, c_2 = c'$ and for each $k \geq 3$, the configuration $c_k$ is obtained from $c_{k-1}$ by applying the sequence of operations $c_1 \xrightarrow{*} c_2$. Clearly the unrolling satisfies all the progressiveness conditions. Consider a corresponding progressive run-with-id $\tau' = d_0' \xrightarrow{*} d_1' \xrightarrow{*} d_2' \xrightarrow{*} d_3' \cdots$ given by Proposition C.5. By Proposition C.4, every id created is either burst or, $\mathrm{seq}_i$ is infinite with $\mathrm{seq}_i$ containting an infinite suffix of trivial deflates. By applying the construction in (S1) iff (S2), we replace every id $i$ which has an infinite $\mathrm{seq}_i$ in $\tau'$ by a VASS token to obtain a progressive $\tau''$ where every balloon is burst. This concludes the proof of Lemma 5.1.

## C.2 Proof of Lemma 5.3

The detailed construction of the VASSB $\mathcal{V}(T)$ corresponding to an $A, B$-witness is given below. The VASSB $\mathcal{V}(T)$ consists of five total stages broken up into three main stages and two auxiliary stages. *First Main Stage: Simulating two copies.* We define a VASSB $\mathcal{V}_1$ that simulates two identical copies of the run of $\mathcal{V}$. The two copies share the global state in VASSB $\mathcal{V}_1$. It maintains two copies of the places, one for each run, and in addition, uses a pair of places $\Omega \times \{1, 2\}$ for each balloon state in order to count the total number of balloons of a given balloon state. An extra pair of places $\Omega \times \{3, 4\}$ for each balloon state remain unused in this stage and are used by later stages for checking the emptiness of balloons. Each step of $\mathcal{V}$ is simulated by $\mathcal{V}_1$ by updating two copies of the places and conceptually maintaining two copies of the balloons. The only non-trivial point is that we cannot maintain two balloons separately, because two different steps executing $\mathrm{inflate}(\sigma, L)$ may pick two different contents from the linear set $L$. We avoid this by maintaining one balloon with two sets of balloon places and extend the linear set $L$ so that it has two identical copies of the same multiset for each set of places.

We introduce some notation. For a $v \in \mathbb{N}^P$ (resp. $\delta \in \mathbb{Z}^P$), we write the "doubling" $v \odot v$ (resp. $\delta \odot \delta$) for the function in $\mathbb{N}^{P \times \{1,2\}}$ such that $v \odot v(p, 1) = v \odot v(p, 2) = v(p)$ (resp. $\mathbb{Z}^{P \times \{1,2\}}$ such that $\delta \odot \delta(p, 1) = \delta \odot \delta(p, 2) = \delta(p)$). For a linear set $L \subseteq \mathbb{N}^P$, we similarly write $L \odot L \subseteq \mathbb{N}^{P \times \{1,2\}}$ for the "doubling" of $L$: $v \odot v \in L \odot L$ iff $v \in L$. A representation of $L \odot L$ can be obtained from the representation of $L$ by doubling the base and period vectors. We also write $\mathbf{0} \circ v$ (resp. $\mathbf{0} \circ L$) to denote the extension of $v \in \mathbb{N}^P$ (resp. $L \subseteq \mathbb{N}^P$) to $\mathbb{N}^{P \times \{1,2\}}$: we define $\mathbf{0} \circ v(p, 1) = 0$ and $\mathbf{0} \circ v(p, 2) = v(p)$, and $\mathbf{0} \circ v \in \mathbf{0} \circ L$ iff $v \in L$. Finally, we write a function $\delta \in \mathbb{Z}^P$ by explicitly mentioning the components that change.

Formally, $\mathcal{V}_1 = (Q \cup E \times \{1, 2\}, P \times \{1, 2\} \cup \Omega \times \{1, 2, 3, 4\}, \Omega \times \Omega \cup \Omega, \Phi \times \{1, 2\}, E_1)$, where we add the following edges to $E_1$:

(1) For each $e = q \xrightarrow{\delta} q'$ in $E$ there is $q \xrightarrow{\delta \odot \delta} q'$ in $E_1$.

(2) For each $e = q \xrightarrow{\text{inflate}(\sigma, L)} q'$ in $E$ the edges $q \xrightarrow{\text{inflate}((\sigma, \sigma), L \odot L)} (e, 1)$ and $(e, 1) \xrightarrow{(\sigma, 1)^+ (\sigma, 2)^+} q'$ are in $E_1$. The second edge keeps track of the number of balloons in state $\sigma$. Note that the first stage only creates balloons with states in $\Omega \times \Omega$.

(3) For each $e = q \xrightarrow{\text{deflate}(\sigma, \sigma', \pi, p)} q'$ in $E$, the following edges are in $E_1$:
   $q \xrightarrow{\text{deflate}((\sigma, \sigma), (\sigma', \sigma), (\pi, 1), (p, 1))} (e, 1)$, $(e, 1) \xrightarrow{\text{deflate}((\sigma', \sigma), (\sigma', \sigma'), (\pi, 2), (p, 2))} (e, 2)$,
   $(e, 2) \xrightarrow{(\sigma, 1)^- , (\sigma', 1)^+ , (\sigma, 2)^- , (\sigma', 2)^+} q'$. The first two edges transfer the balloon tokens to $(p, 1)$ and $(p, 2)$, respectively. The last edge tracks the new number of balloons in each balloon state.

(4) For each $e = q \xrightarrow{\text{burst}(\sigma)} q'$ in $E$, the edges $q \xrightarrow{\text{burst}(\sigma, \sigma)} (e, 1)$ and $(e, 1) \xrightarrow{(\sigma, 1)^- (\sigma, 2)^-} q'$ are in $E'$. The second edge maintains the count of the number of balloons with balloon state $\sigma$.

For an initial (semi-)configuration $c_0 = (q_0, \mathbf{m}_0, \emptyset)$ of $\mathcal{V}$, one can construct a configuration of $\mathcal{V}_1$ that makes two identical copies on to the places and initializes the places $\sigma \times \{1, 2, 3, 4\}$ to zero. A run of $\mathcal{V}_1$ decomposes into two identical runs of $\mathcal{V}$, and for any reachable configuration, the places $P \times \{1\}$ and $P \times \{2\}$ have identical number of tokens. So do the places $\Omega \times \{1\}$ and $\Omega \times \{2\}$ which track the number of balloons in each copy for a given balloon state. The places in $\Omega \times \{3, 4\}$ remain empty. Each balloon can be divided into two identical balloons by focusing on the two copies of the places.

*First Auxilliary Stage: Checking Emptiness of Balloons in $c_1$*. The first auxiliary stage uses a VASSB $\mathcal{V}_{1 \to 2}$ which is used to check balloons in $c_1$ for emptiness while at the same time transferring tokens from the copy $\Omega \times \{1\}$ to the copy $\Omega \times \{3\}$. The only places used in this stage are those in $\Omega \times \{1, 3\}$. The states used by $\mathcal{V}_{1 \to 2}$ are two copies of the states of $\mathcal{V}$ together with states of the form $(q, \sigma, \pi)$ used to perform the emptiness check. We also have two marked copies of balloon states $\tilde{\Omega} \times \{1, 2\}$. The VASSB $\mathcal{V}_{1 \to 2}$ picks a balloon $b$ from $c_1$ and performs one deflate operation for each $\pi \in \Phi$ on $b$, sending all tokens to a special place $p_{\text{check}}$. During the check, the balloon is put into the first special marked copy of its state to ensure that all of the checks are performed on the balloon $b$ picked. At the end of the check, a token is transferred indicating that $b$ has been checked for emptiness and the balloon state is put into the second marked copy. The series of checks forms a loop starting and ending at a state $(q, 1)$ while passing through states of the form $(q, \sigma, \pi)$, with one loop per balloon checked. In the event of a correct check on all balloons in $c_1$, the place $p_{\text{check}}$ remains empty, all the tokens have been transferred and control is passed to the second main stage. In checking emptiness only the first copy of $Q$ is used. $\mathcal{V}_{1 \to 2}$ then non-deterministically guesses that all the balloons have been checked for emptiness and moves from a state $(q, 1)$ to the state $(q, 2)$ in the second copy of $Q$. It then converts the balloon states of all the balloons from the second marked copy to the normal balloon state. Note that this conversion cannot be done before all the balloons are checked for emptiness since we could otherwise make the mistake of checking only one balloon repeatedly for emptiness.

Let $\Phi = \{\pi_1, \pi_2, \cdots, \pi_n\}$ be an arbitrary enumeration. Formally, $\mathcal{V}_{1 \to 2} = (Q_{1 \to 2}, p_{\text{check}} \cup \Omega \times \{1, 2, 3, 4\}, ((\Omega \times \Omega) \cup (\tilde{\Omega} \times \tilde{\Omega} \times \{1, 2\})), \Phi \times \{1, 2\}, E_{1 \to 2})$ where $\tilde{\Omega}$ is a decorated copy of $\Omega$. The set global of states is $Q_{1 \to 2} = (Q \times \{1, 2\}) \cup (Q \times \Omega \times \Phi)$ and we add the following edges to $E_{1 \to 2}$ for each $q \in Q$ and $\sigma \in \Omega$:

(1) $(q, 1) \xrightarrow{\text{deflate}((\sigma, \sigma), (\tilde{\sigma}, \tilde{\sigma}, 1), (\pi_1, 1), p_{\text{check}})} (q, \sigma, \pi_1)$,

(2) for each $k \in [1, n-2]$ add
$$(q, \sigma, \pi_k) \xrightarrow{\text{deflate}((\tilde{\sigma}, \tilde{\sigma}, 1), (\tilde{\sigma}, \tilde{\sigma}, 1), (\pi_{k+1}, 1), p_{\text{check}})} (q, \sigma, \pi_{k+1}),$$

(3) $(q, \sigma, \pi_{n-1}) \xrightarrow{\text{deflate}((\tilde{\sigma}, \tilde{\sigma}, 1), (\tilde{\sigma}, \tilde{\sigma}, 2), (\pi_n, 1), p_{\text{check}})} (q, \sigma, \pi_n)$ and

(4) $(q, \sigma, \pi_n) \xrightarrow{(\sigma, 1)^-, (\sigma, 3)^+} (q, 1)$. A token is transferred from $(\Omega, 1)$ to $(\Omega, 3)$.

(5) $(q, 1) \xrightarrow{\mathbf{0}} (q, 2)$,

(6) $(q, 2) \xrightarrow{\text{deflate}((\tilde{\sigma}, \tilde{\sigma}, 2), (\sigma, \sigma), (\pi_n, 1), p_{\text{check}})} (q, 2)$.

*Second Main Stage: Simulating $c_1 \xrightarrow{*} c_2$.* At this point, we note that the first copy of $c_1$ resides in the first copy of places and the third copy of balloon states. The second main stage is given by a VASSB $\mathcal{V}_2$ which keeps the first copy of $c_1$ frozen and simulates the VASSB $\mathcal{V}$ on the second copy. It additionally performs verification of the progress constraints. Thus, a state consists of a triple $(q, q', M) \in Q \times Q \times 2^{A \cup B}$, where $q$ keeps the state of $c_1$, $q'$ is the current state of the simulation, and $M$ is a subset of $A \cup B$. When control moves from state $q$ in $\mathcal{V}_{1 \to 2}$ to $\mathcal{V}_2$, we start at $(q, q, \emptyset)$, where the $\emptyset$ denotes none of the progress constraints have been checked.

The simulation only updates the second copy of the places. New balloons now use the states of $\mathcal{V}$ and we continue to track the number of balloons for each balloon state but only in the second copy i.e. the places $\Omega \times \{2\}$. The other copies $\Omega \times \{1, 3, 4\}$ remain untouched. A deflate or a burst operation may be performed on double-balloons with state $\sigma \times \sigma$ or on normal balloons with state $\sigma$. On double-balloons, deflate works on the second component only.

Formally, $\mathcal{V}_2 = (Q_2, P \times \{1, 2\} \cup \Omega \times \{1, 2, 3, 4\}, \Omega \times \Omega \cup \Omega, \Phi \times \{1, 2\}, E_2)$ consists of states $Q_2 = \{(q, q', M) \mid q \in Q, q' \in Q \cup E, M \subseteq A \cup B\}$, and the following edges in $E_2$:

(1) Let $e = q \xrightarrow{\delta} q'$. Let $P_\delta = \{p \in A \mid \delta(p) < 0\}$. If $\text{supp}(\delta) \subseteq A$, then for each $q_1 \in Q, M \subseteq A \cup B$, and $p \in P$, add $(q_1, q, M) \xrightarrow{\mathbf{0} \circ \delta} (q_1, q', M)$ if $p \in P \setminus P_\delta$ and $(q_1, q, M) \to (q_1, q', M \cup P_\delta)$ if $p \in P_\delta$. The $\delta$ edges are restricted to those whose support is in $A$. We track the place progress condition.

(2) For each $e = q \xrightarrow{\text{inflate}(\sigma, L)} q'$ in $E$, for each $q_1 \in Q, M \subseteq A \cup B$, add $(q_1, q, M) \xrightarrow{\text{inflate}(\sigma, \mathbf{0} \circ L)} (q_1, e, M)$ and $(q_1, e, M) \xrightarrow{(\sigma, 2)^+} (q_1, q', M)$.

(3) For each $e = q \xrightarrow{\text{deflate}(\sigma, \sigma', \pi, p)} q'$ in $E$, for each $q_1 \in Q, M \subseteq A \cup B, \sigma'' \in \Omega$ add:
$(q_1, q, M) \xrightarrow{\text{deflate}((\sigma'', \sigma), (\sigma'', \sigma'), (\pi, 2), (p, 2))} (q_1, e, M')$, $(q_1, q, M) \xrightarrow{\text{deflate}(\sigma, \sigma', (\pi, 2), (p, 2))} (q_1, e, M)$,
$(q_1, e, M) \xrightarrow{(\sigma, 2)^-, (\sigma', 2)^+} (q_1, q', M')$, where $M' = M \cup \{\sigma\}$ if $\sigma \in B$ and $M' = M$ otherwise. The first edge deflates doubled balloons left over from the first main stage, but only to the second copy of $P$. We track the balloon progress condition only for balloons from the first main stage.

(4) For each $e = q \xrightarrow{\text{burst}(\sigma)} q'$ in $E$, for each $q_1 \in Q, M \subseteq A \cup B, \sigma' \in \Omega$, add $(q_1, q, M) \xrightarrow{\text{burst}((\sigma', \sigma))} (q_1, e, M')$, $(q_1, q, M) \xrightarrow{\text{burst}(\sigma)} (q_1, e, M)$, and $(q_1, e, M) \xrightarrow{(\sigma, 2)^-} (q_1, q', M)$, where $M' = M \cup \{\sigma\}$ if $\sigma \in B$ and $M' = M$ otherwise.

*Second Auxiliary Stage: Checking Emptiness of Balloons in $c_2$.* The difference between the first auxiliary stage and second auxiliary stage is that the emptiness must be checked for both the balloons produced in the first main stage as well as the second main stage in the latter, since there may be a double-balloon that continues to exist at $c_2$. The emptiness check transfers tokens from the copy $\Omega \times \{2\}$ to the copy $\Omega \times \{4\}$. As in the first auxiliary stage, we pick an enumeration $\{\pi_1, \pi_2, \cdots, \pi_n\}$ of $\Phi$. Formally, $\mathcal{V}_{2 \to 3} = (Q_{2 \to 3}, p_{\text{check}} \cup \Omega \times \{1, 2, 3, 4\}, (\Omega \times \Omega \cup \Omega) \cup ((\tilde{\Omega} \times \tilde{\Omega} \cup \tilde{\Omega}) \times \{1, 2\}), \Phi \times \{1, 2\}, E_{2 \to 3})$

where $\tilde{\Omega}$ is a decorated copy of $\Omega$. The set of global states is $Q_{2\to3} = (Q\times\{1,2\})\cup Q\times\Phi\times(\Omega\cup\Omega\times\Omega)$ and we add the following edges to $E_{2\to3}$ for each $q \in Q$, $(\sigma,\sigma') \in \Omega\times\Omega$ and $\sigma \in \Omega$:

(1) $(q,1) \xrightarrow{\mathsf{deflate}((\sigma,\sigma'),(\tilde{\sigma},\tilde{\sigma}',1),\pi_1,p_{\mathsf{check}})} (q,\pi_1,\sigma,\sigma')$,

   $q \xrightarrow{\mathsf{deflate}(\sigma,(\tilde{\sigma},1),\pi_1,p_{\mathsf{check}})} (q,\pi_1,\sigma)$,

(2) for each $k \in [1, n-2]$ add

   $(q,\pi_k,\sigma,\sigma') \xrightarrow{\mathsf{deflate}((\tilde{\sigma},\tilde{\sigma}',1),(\tilde{\sigma}',\tilde{\sigma}',1),\pi_{k+1},p_{\mathsf{check}})} (q,\pi_{k+1},\sigma,\sigma')$,

   $(q,\pi_k,\sigma) \xrightarrow{\mathsf{deflate}((\tilde{\sigma},1),(\tilde{\sigma},1),\pi_{k+1},p_{\mathsf{check}})} (q,\pi_{k+1},\sigma)$,

(3) $(q,\pi_{n-1},\sigma,\sigma') \xrightarrow{\mathsf{deflate}((\tilde{\sigma},\tilde{\sigma}',1),\tilde{\sigma},\tilde{\sigma}',2),\pi_n,p_{\mathsf{check}})} (q,\pi_n,\sigma,\sigma')$,

   $(q,\pi_{n-1},\sigma) \xrightarrow{\mathsf{deflate}((\tilde{\sigma},1)(\tilde{\sigma},2),\pi_n,p_{\mathsf{check}})} (q,\pi_n,\sigma)$ and

(4) $(q,\pi_n,\sigma,\sigma') \xrightarrow{(\sigma',2)^-,(\sigma',4)^+} (q,1)$, $(q,\pi_n,\sigma) \xrightarrow{(\sigma,2)^-,(\sigma,4)^+} (q,1)$ . A token is transferred from $(\Omega,2)$ to $(\Omega,4)$.

(5) $(q,1) \xrightarrow{\mathbf{0}} (q,2)$,

(6) $(q,2) \xrightarrow{\mathsf{deflate}((\tilde{\sigma},\tilde{\sigma}',2),(\sigma,\sigma'),\pi_n,p_{\mathsf{check}})} (q,2)$,

   $(q,2) \xrightarrow{\mathsf{deflate}((\tilde{\sigma},2),\sigma,\pi_n,p_{\mathsf{check}})} (q,2)$.

*Third Main Stage: Verification.* At this point, the first copy of $c_1$ uses balloon places $\Omega\times\{3\}$ while $c_2$ uses the balloon places $\Omega\times\{4\}$. In the third main stage, verification non-deterministically removes the same number of tokens from the two copies for each place in $A$ and for each place in $B$ in the copies $\Omega\times\{3,4\}$. Additionally, all balloons are burst. Finally, all places in $P\times\{2\}$ and $\Omega\times\{4\}$ are emptied. Formally, VASSB $\mathcal{V}_3 = (\{q_{\mathsf{ver}}, q_f\}, P\times\{1,2\}\cup\Omega\times\{1,2,3,4\}, \Omega\cup\Omega\times\Omega, \Phi\times\{1,2\}, E_3)$ contains the following edges:

(1) For each $p \in A$, there is an edge $q_{\mathsf{ver}} \xrightarrow{(p,1)^-(p,2)^-} q_{\mathsf{ver}}$ and for each $\sigma \in B$, the edge $q_{\mathsf{ver}} \xrightarrow{(\sigma,3)^-(\sigma,4)^-} q_{\mathsf{ver}}$. For each $\sigma \in \Omega$, the edge $q_{\mathsf{ver}} \xrightarrow{\mathsf{burst}(\sigma)} q_{\mathsf{ver}}$ and for each $(\sigma,\sigma') \in \Omega\times\Omega$ the edge $q_{\mathsf{ver}} \xrightarrow{\mathsf{burst}(\sigma,\sigma')} q_{\mathsf{ver}}$

(2) $q_{\mathsf{ver}} \xrightarrow{\mathbf{0}} q_f$ and for each $p \in A$, the edge $q_f \xrightarrow{(p,2)^-} q_f$ and for each $\sigma \in B$, the edge $q_f \xrightarrow{(\sigma,4)^-} q_f$.

The edges in (1) ensure $\mathsf{p}(c_1) \leq_p \mathsf{p}(c_2)$ in the places $A\cup B$ and that all remaining balloons are removed. The edges in (2) remove extra tokens in case $\mathsf{p}(c_1) <_p \mathsf{p}(c_2)$.

*Overall VASSB and Transitions between Stages.* The VASSB $\mathcal{V}(T)$ is a composition of the five stages $\mathcal{V}_1$, $\mathcal{V}_{1\to2}$, $\mathcal{V}_2$, $\mathcal{V}_{2\to3}$ and $\mathcal{V}_3$. A set of glue transitions connect the five stages: these non-deterministically guess when to move from $\mathcal{V}_1$ to $\mathcal{V}_{1\to2}$, $\mathcal{V}_{1\to2}$ to $\mathcal{V}_2$, $\mathcal{V}_2$ to $\mathcal{V}_{2\to3}$ and from $\mathcal{V}_{2\to3}$ to $\mathcal{V}_3$. Transfers between stages do not perform any operations and use the following edges:

- From $\mathcal{V}_1$ to $\mathcal{V}_{1\to2}$, for each $q \in Q$ of $\mathcal{V}_1$ to the corresponding first copy $(q,1)$ in $\mathcal{V}_{1\to2}$,
- from $\mathcal{V}_{1\to2}$ to $\mathcal{V}_2$, for each $(q,2) \in Q\times\{2\}$ of $\mathcal{V}_{1\to2}$ to the corresponding $(q,q,\emptyset)$ of $\mathcal{V}_2$,
- from $\mathcal{V}_2$ to $\mathcal{V}_{2\to3}$, for each $(q,q,A\cup B)$ of $\mathcal{V}_2$ to the corresponding $(q,1)$ of $\mathcal{V}_{2\to3}$, and
- from $\mathcal{V}_{2\to3}$, for each $(q,2) \in Q\times\{2\}$ of $\mathcal{V}_{2\to3}$ to $q_{\mathsf{ver}}$ of $\mathcal{V}_3$.

*Correctness.* A correct simulation of the progressiveness witness results in $\mathcal{V}(T)$ reaching the configuration $(q_f, \emptyset, \bar{\mathbf{0}})$. Conversely, suppose $\mathcal{V}(T)$ reaches $(q_f, \emptyset, \bar{\mathbf{0}})$. This implies that it reaches a configuration $(q_{\mathsf{ver}}, \mathbf{m}, \nu)$ where $\mathbf{m}$ only contains tokens in the second copy of $A$ and the fourth copy of $B$. This implies that all the balloons of $c_2$ were correctly checked for emptiness in $\mathcal{V}_{2\to3}$ since no tokens remain in the second copy of $\Omega$ and there are no tokens in $p_{\mathsf{check}}$. Similarly, the emptiness check performed by $\mathcal{V}_{1\to2}$ is also correct. Thus, the configuration $c_1$ at the end of $\mathcal{V}_1$

and $c_2$ at the end of $\mathcal{V}_2$ satisfy $p(c_1) \leq p(c_2)$ and they both only contain empty balloons with state $B$ and have non-zero tokens in places in $A$. Further, the transition between $\mathcal{V}_2$ and $\mathcal{V}_{2\rightarrow 3}$ ensures the progressivenes conditions on $A$ and $B$ have been checked.

## D  PROOFS FOR SECTION 5.3

The proofs of Lemmas 5.4 and 5.5 both assume that the VASSB considered is both zero-base (by Lemma C.7) and typed (by Lemma C.6). Further, the notion of runs-with-id introduced in Appendix C is used in the proof of Lemma 5.4.

### D.1  Proof of Lemma 5.4

In any finite run-with-id $\tau$, let $I_\sigma(\tau)$ be the set of id's of non-empty balloons inflated with state $\sigma$. Note that by definition, $M_\sigma.i = 0$ for any newly created balloon. Let $\tau' = s_1 \xrightarrow{*} s_2 = d'_0 \xrightarrow{op'_1} d'_1 \xrightarrow{op'_2} d'_2 \cdots$ be an arbitrary canonical run-with-id which is $N$-balloon bounded i.e. inflates at most $N$ non-empty balloons. We equivalently assume that the bound applies to each balloon state since this implies a bound of $|\Omega|N$ on the total number of balloons. Suppose there exists a state $\sigma_0$ with $|I_{\sigma_0}(\tau')| > N$. We will show that there exists $\tau = s_1 \xrightarrow{*} s_2$ of $\mathcal{V}$ with $|I_\sigma(\tau)| = |I_\sigma(\tau')|$ for $\sigma \neq \sigma_0$ and $|I_{\sigma_0}(\tau)| < |I_{\sigma_0}(\tau')|$. Clearly, this suffices to prove the lemma.

For any deflate sequence $S = (\pi_1, p_1), \cdots, (\pi_n, p_n)$, let $s(S)$ be the set $\{(\pi_1, p_1), \cdots, (\pi_n, p_n)\}$. For id's $i, j \in I_{\sigma_0}(\tau')$ with $i < j$, let $\text{color}_{i,j} : (s(M_{\sigma_0}.S)) \rightarrow \{\text{green}, \text{red}\}$ be defined by

$$\text{color}_{i,j}(\pi, p) = \begin{cases} \text{green} & \text{if } \exists k < k', (\pi, p, k) \in \text{typeseq}_i, (\pi, p, k') \in \text{typeseq}_j \\ \text{red} & \text{otherwise} \end{cases}$$

The color green (resp. red) indicates that a deflate $(\pi, p)$ of the balloon $i$ occurs before (resp. after) the corresponding deflate of balloon $j$. We write $k = \text{typeseq}_i(\pi, p)$ if $(\pi, p, k) \in \text{typeseq}_i$. The set $C_{\sigma_0} = \{\text{color}_{i,j} \mid i, j \in I_{\sigma_0}(\tau')\}$ of functions is finite since both the domain and range of the functions is finite. Let $G_{\sigma_0}$ be the graph with $I_{\sigma_0}$ as the set of vertices and edges colored by $C_{\sigma_0}$. Note that the color $\text{color}_{i,j}$ of the edge between vertices $i$ and $j$ is a finite word from $\{\text{green}, \text{red}\}^*$. Let $r = |C_{\sigma_0}| \leq 2^{|\Phi|}$ and $n = |\Omega|^{|\Phi|} + 1$. Then by Ramsey's theorem [Ramsey 1930, Theorem B] there exists $R(r; n) \in \mathbb{N}$ such that for any $r$-colored graph with at least $R(r; n)$ vertices, there exists a monochromatic subgraph of size at least $n$. Further, by the result of Erdős and Rado [Erdős and Rado 1952, Theorem 1] we know that $R(r; n) \leq r^{r(n-2)+1} \leq O(\exp_4(|\mathcal{V}|))$. Choosing $N = r^{r(n-2)+1}$, this implies that $|I_{\sigma_0}| > r^{r(n-2)+1}$ and therefore there exists a monochromatic subgraph $G'$ of $G_{\sigma_0}$ with at least $n$ many vertices. Let $c$ be the color of every edge in the graph induced by $G'$. A red-*block* $c[i, j]$ of $c$ is a maximal contiguous subsequence $(\pi_i, p_i), (\pi_{i+1}, p_{i+1}), \cdots, (\pi_j, p_j)$ of $M_{\sigma_0}.S$ such that $c$ takes value red on each $(\pi_k, p_k)$ for $i \leq k \leq j$ and value green on $(\pi_{j+1}, p_{j+1})$ (if $j < n$) and $(\pi_{i-1}, p_{i-1})$ (if $i > 1$). Let $c[j_{1,1}, j_{1,2}], c[j_{2,1}, j_{2,2}], \cdots, c[j_{l,1}, j_{l,2}]$ be the red-blocks of $c$ with $j_{k,2} < j_{k+1,1}$ for each $k$.

Intuitively, deflates of balloons from $G'$ in a particular red-block happen in the reverse order as compared to their id's. That is, if $i_0 < j_0 < k_0 < l_0$ are id's which are present in $G'$, then in a red-block, $l_0$ is deflated first, followed by $k_0$, then $j_0$ and finally $i_0$. In Figure 3, one such red-block (called 1-block in the main body) is formed by the second and third deflates where the magenta outlined balloon corresponding to $l_0$ is deflated first at $l_2$ and $l_3$, followed by the green, orange and cyan outlined balloons.

Formally, let $\min(G')$ (resp. $\max(G')$) be the least (resp. greatest) id in $G'$. For each $i \in G'$ and for each $k$ with $1 \leq k \leq l$, we have $\text{typeseq}_{\max(G')}(j_{k,1}) \leq \text{typeseq}_i(j_{k,1})$ and $\text{typeseq}_i(j_{k,2}) \leq \text{typeseq}_{\min(G')}(j_{k,2})$ by construction. In other words, for any red-block $c[j_{k,1}, j_{k,2}]$ of $c$, all non-trivial deflates made by balloons in $G'$ happen in $\tau'[m_{k,1}, m_{k,2}]$ where $m_{k,1} = \text{typeseq}_{\max(G')}(j_{k,1}) - 1$ and

$m_{k,2} = \text{typeseq}_{\min(G')}(j_{k,2})$ and further, no other non-trivial deflates of balloons in $G'$ other than the ones in $c[j_{k,1}, j_{k,2}]$ occur in $\tau'[m_{k,1}, m_{k,2}]$. Let $\tau'[m_{1,1}, m_{1,2}], \tau'[m_{2,1}, m_{2,2}], \cdots, \tau'[m_{l,1}, m_{l,2}]$ be the infixes of $\tau'$ corresponding to $c[j_{1,1}, j_{1,2}], c[j_{2,1}, j_{2,2}], \cdots, c[j_{l,1}, j_{l,2}]$. We observe that the total number of points $j_{1,1}, j_{1,2}, j_{2,1}, \cdots j_{l,2}$ is at most $|\Phi|$ since the $\pi$ values at all these points is distinct and therefore, the same bound applies to $m_{1,1}, m_{1,2}, m_{2,1}, m_{2,2}, \cdots, m_{l,1}, m_{l,2}$. Let $M = \bigcup_{1 \le k \le l}[m_{k,1}, m_{k,2}]$.

By construction, $|G'| = n \ge |\Omega|^{|\Phi|} + 1$, which implies that there exist $i, j \in G'$ with $i < j$ such that for each $k \ge 1$ and each $l \in \{1, 2\}$ we have $d'_{m_{k,l}}.i.\sigma = d'_{m_{k,l}}.j.\sigma$ i.e. the balloons $i$ and $j$ have the same balloon state at each of the configurations $d'_{m_{k,l}}$. Let $op'_i$ (resp. $op'_j$) in $\tau'$ create a balloon $(b', i)$ (resp. $(b'', j)$) with $b'.\mathbf{k} = \mathbf{k}_i$ (resp. $b''.\mathbf{k} = \mathbf{k}_j$). We now construct $\tau = d_0 \xrightarrow{op_1} d_1 \xrightarrow{op_2} d_2 \cdots$ from $\tau'$ by an *id-switching surgery* in which all operations in $\tau$ are retained by $\tau'$ with the exception of those in $\text{seq}'_i$ and $\text{seq}'_j$, which are replaced by $\text{seq}_i$ and $\text{seq}_j$ as follows:

(1) $op_i$ creates $(b_i, i)$ where $b_i.\mathbf{k} = \mathbf{k}_i + \mathbf{k}_j$ and $b_i.\sigma = b'.\sigma$ and $op_j$ creates $(b_j, j)$ where $b_j.\mathbf{k} = \emptyset$ and $b_j.\sigma = b''.\sigma$,
(2) for each $k \in \text{seq}'_i$ (resp. $k \in \text{seq}'_j$) where $k \in M$, $k \in \text{seq}_j$ (resp. $k \in \text{seq}_i$) and
(3) for each $k \in \text{seq}'_i$ (resp. $k \in \text{seq}'_j$) where $k \notin M$, $k \in \text{seq}_i$ (resp. $k \in \text{seq}_j$).

Intuitively, all operations in $\tau'$ belonging to the segments in $M$ on balloon $i$ are reinterpreted as being performed on balloon $j$ in $\tau$ and vice versa, while operations outside of the segments in $M$ are retained as before. First we observe that this reinterpretation results in a valid sequence of operations $\text{seq}_i$ and $\text{seq}_j$ in $\tau$ since we have chosen $i, j$ such that the balloon states match at every boundary point of $M$. This implies that $i <_t j$ in $\tau$ and hence (1) which is a token-shifting operation ensures that $\tau$ is valid by Proposition C.8 and is a run from $s_1$ to $s_2$. Since balloon $j$ is empty in $\tau$, this implies $|I_{\sigma_0}(\tau)| < |I_{\sigma_0}(\tau')|$.

## D.2 Proof of Lemma 5.5

Given VASSB $\mathcal{V} = (Q, P, \Omega, \Phi, E)$ and its semiconfigurations $s_1, s_2$, we construct a VASS $\mathcal{V}' = (Q', P', E')$ and its configurations $s'_1, s'_2$ such that there exists an $N$-balloon-bounded run $\rho = s_1 \xrightarrow{*} s_2$ of $\mathcal{V}$ (where $N$ is given by Lemma 5.4) iff $(\mathcal{V}', s'_1, s'_2) \in \text{REACH}$.

We assume that $\mathcal{V}'$ can have edges of the form $e = q \xrightarrow{+\Delta_L} q'$ where $+\Delta_L$ is an operation which non-determinstically adds an element from a linear set $L$ to the set of places, since reducing such an extension to a normal VASS is a standard construction. We similarly have operations $-\Delta_L$ which remove a vector $\mathbf{v} \in L$ from the set of places. For sets $\Phi = \{\pi_1, \cdots, \pi_n\}$ and $\Phi' = \{\pi'_1, \cdots, \pi'_n\}$, for $L \subseteq \mathbb{M}[\Phi]$, we write $L(\Phi \leftarrow \Phi') \subseteq \mathbb{M}[\Phi']$ for the corresponding linear set of vectors $\mathbf{v}'$ where $\mathbf{v}'(\pi'_i) = \mathbf{v}(\pi_i)$ for each $i$ where $\mathbf{v} \in L$. For $L \subseteq \mathbb{M}[\{\pi\}]$ and $p \in P$, we also define the multiset $L(\pi \odot (-p)) \subseteq \mathbb{M}[\{\pi, p\}]$ consisting of vectors $\mathbf{v}^- \odot \mathbf{v}(p)$ obtained from $\mathbf{v} \in L$ as $(\mathbf{v}^- \odot \mathbf{v}(p))(\pi) = -(\mathbf{v}^- \odot \mathbf{v}(p))(p) = -\mathbf{v}(\pi)$. The projection $L \!\downarrow_\chi$ of $L$ to a subset $\chi \subseteq \Phi$ is obtained by retaining the values of vectors for the $\chi$ component and setting the rest to 0. All of the above defined operations enable us to simulate balloon operations by addition and subtraction operations on VASS places.

Let $H$ be the set of all functions $\eta : \Omega \times \{1, \cdots, N\} \to \Omega \cup \{\#, \dagger\}$. A function $\eta \in H$ is used in the following construction to track the state of a non-empty balloon of $\mathcal{V}$ in the global state of $\mathcal{V}'$. The # symbol indicates that a balloon has non yet been inflated and $\dagger$ indicates that it has been burst. Let $G$ be the set of vectors $\mathbf{w} : \Omega \to \{0, 1, \cdots, N\}$. A vector $\mathbf{w} \in G$ is used to store the number of non-empty balloons of a particular balloon state in $\mathcal{V}$ in the global state of $\mathcal{V}'$. Increments $p^+$ (resp. decrements $p^-$) to a place $p$ are used as before. For a type sequence $S = (\pi_1, p_1), \cdots, (\pi_n, p_n)$, we write $\text{set}_1(S)$ for the set $\{\pi_1, \cdots, \pi_n\}$.

$\mathcal{V}'$ is defined as:

- $Q' = Q \times G \times H \cup \{q_f\}$,
- $P' = P \cup \Omega \times \{1, \cdots, N\} \times \Phi \cup \Omega$,
- $E'$ contains the following edges:

(1) for each $q \xrightarrow{\delta} q'$ in $E$, for each $\mathbf{w} \in G$ and $\eta \in H$ add

     $(q, \mathbf{w}, \eta) \xrightarrow{\delta} (q', \mathbf{w}, \eta)$ to $E'$,

(2) for each $q \xrightarrow{\text{inflate}(\sigma_1, L)} q'$ in $E$,

     (a) for each $\mathbf{w} \in G$ and $\eta \in H$ add $(q, \mathbf{w}, \eta) \xrightarrow{\sigma_1^+} (q', \mathbf{w}, \eta)$ to $E'$,

     (b) for each $\eta \in H$ and for each $\mathbf{w} \in G$ such that $\mathbf{w}(\sigma_1) < N$, add

         $(q, \mathbf{w}, \eta) \xrightarrow{\Delta_{\mathbf{w}, \sigma_1}} (q', \mathbf{w}', \eta')$ to $E'$ where
         $\Delta_{\mathbf{w}, \sigma_1}$ adds a vector from $L_{\sigma_1}(\Phi \leftarrow \{\sigma_1\} \times \{\mathbf{w}(\sigma_1) + 1\} \times \Phi)$,
         the vector $\mathbf{w}'$ satisfies $\mathbf{w}'(\sigma_1) = \mathbf{w}(\sigma_1) + 1$ and $\mathbf{w}'(\sigma) = \mathbf{w}(\sigma)$ for $\sigma \neq \sigma_1$, and
         $\eta'$ satisfies $\eta'(\sigma_1, \mathbf{w}(\sigma_1) + 1) = \sigma_1$ and for all $(\sigma, i) \neq (\sigma_1, \mathbf{w}(\sigma_1) + 1)$ we have $\eta'(\sigma, i) = \eta(\sigma, i)$,

(3) for each $q \xrightarrow{\text{deflate}(\sigma, \sigma', \pi, p)} q'$ in $E$,

     (a) for each $\mathbf{w} \in G$ and $\eta \in H$ add $(q, \mathbf{w}, \eta) \xrightarrow{\sigma'^+, \sigma^-} (q', \mathbf{w}, \eta)$ to $E'$,

     (b) for each $\mathbf{w} \in G$, for each $\eta \in H$ and $(\sigma_1, i) \in \Omega \times \{1, \cdots, N\}$ such that $\eta(\sigma_1, i) = \sigma$,

         (i) if $M_{\sigma'}.i = M_\sigma.i + 1$ then add

             $(q, \mathbf{w}, \eta) \xrightarrow{\Delta_{\sigma_1, i, \pi, p}} (q', \mathbf{w}, \eta')$ to $E'$ where
             $\Delta_{\sigma_1, i, \pi, p}$ adds a vector $\mathbf{v}^- \odot \mathbf{v}(p)$ corresponding to some $\mathbf{v} \in L_{\sigma_1} \mathord{\downharpoonright}_\pi (\pi \leftarrow \{\sigma_1\} \times \{i\} \times \pi)$
             and
             $\eta'$ satisfies $\eta'(\sigma_1, i) = \sigma'$ and $\eta'(\sigma, j) = \eta(\sigma, j)$ for all $(\sigma, j) \neq (\sigma_1, i)$ ,

         (ii) else add

             $(q, \mathbf{w}, \eta) \xrightarrow{\mathbf{0}} (q', \mathbf{w}, \eta')$ to $E'$ where
             $\eta'$ satisfies $\eta'(\sigma_1, i) = \sigma'$ and $\eta'(\sigma_2, j) = \eta(\sigma_2, j)$ for all $(\sigma_2, j) \neq (\sigma_1, i)$ ,

(4) for each $q \xrightarrow{\text{burst}(\sigma)} q'$ in $E$,

     (a) for each $\mathbf{w} \in G$ and $\eta \in H$ add $(q, \mathbf{w}, \eta) \xrightarrow{\sigma^-} (q', \mathbf{w}, \eta)$ to $E'$,

     (b) for each $\mathbf{w} \in G$, for each $\eta \in H$ and $(\sigma_1, i) \in \Omega \times \{1, \cdots, N\}$ such that $\eta(\sigma_1, i) = \sigma$, add

         $(q, \mathbf{w}, \eta) \xrightarrow{-\Delta_{\sigma_1, i, (\Phi \setminus \text{set}_1(M_{\sigma_1}.S))}} (q', \mathbf{w}, \eta')$ to $E'$ where
         $\Delta_{\sigma_1, i, (\Phi \setminus \text{set}_1(M_{\sigma_1}.S))}$ adds a vector from $L_{\sigma_1} \mathord{\downharpoonright}_{(\Phi \setminus \text{set}_1(M_{\sigma_1}.S))} ((\Phi \setminus \text{set}_1(M_{\sigma_1}.S)) \leftarrow \{\sigma_1\} \times$
         $\{i\} \times (\Phi \setminus \text{set}_1(M_{\sigma_1}.S)))$ and
         $\eta'$ satisfies $\eta'(\sigma_1, i) = \dagger$ and $\eta'(\sigma_2, j) = \eta(\sigma_2, j)$ for all $(\sigma_2, j) \neq (\sigma_1, i)$,

(5) for each $\eta$ such that for all $(\sigma, i) \in \Omega \times \{1, \cdots, N\}$ we have $\eta(\sigma, i) = \#$ or $\eta(\sigma, i) = \dagger$, add

     $(s_2.q, \mathbf{w}, \eta) \xrightarrow{\mathbf{0}} q_f$.

The configuration $s_1'$ is given as $s_1'.q = (s_1.q, \mathbf{0}, \eta_0)$ where $\eta_0(\sigma, i) = \#$ for all $(\sigma, i) \in \Omega \times \{1, \cdots, N\}$, and $s_1'.\mathbf{m}(p) = s_1.\mathbf{m}(p)$ for $p \in P$ and $s_1'.\mathbf{m}(p) = 0$ otherwise. The configuration $s_2'$ is given as $s_2'.q = q_f$, and $s_2'.\mathbf{m}(p) = s_2.\mathbf{m}(p)$ for $p \in P$ and $s_2'.\mathbf{m}(p) = 0$ otherwise.

$\mathcal{V}'$ contains extra places $\Omega$ which are used to store the number of balloons of a given state $\sigma$ which were created empty and remain empty throughout a run, as well as places $\Omega \times \{1, \cdots, N\} \times \Phi$ which are used to store the contents of non-empty balloons created during an $N$-balloon-bounded run of $\mathcal{V}$. The $G$-component of the global state is used to track the total number of non-empty balloons created while the $H$-component tracks the state changes of every balloon which was created non-empty. The initial configuration $s_1'$ of $\mathcal{V}$ contains $s_1$ in its $P$-places while other places

remain empty. The $G$-component of the global state is set to the vector $\mathbf{0}$ since no non-empty balloons have been created and the $H$-component is set to the function $\eta_0$ which is the constant function taking value #. The edges in (1) are used to simulate the place edges of $\mathcal{V}$ and hence do not modify either the $G$ or the $H$-component. Similarly, the simulation of operations on balloons which were created empty use edges from $(2a)$, $(3a)$ and $(4a)$ which also do not modify either the $G$ or the $H$-component and simply increment or decrement places from $\Omega$ as appropriate.

An inflate balloon operation in the run $\rho$ of $\mathcal{V}$ which creates the $i^{th}$ non-empty balloon (upto the bound $N$) with state $\sigma$ is simulated by using edges from $(2b)$ which increment $\mathbf{w} \in G$ from $i-1$ to $i$ and set the value of $\eta(\sigma, i)$ to $\sigma$ from its original value of #, while simultaneously populating the places in $(\sigma, i, \Phi)$ with a vector from $L_\sigma$. Deflate operations on non-empty balloons similarly are simulated by updating $\eta$ and removing tokens from the appropriate place using the edges in $(3b)$. Here, we make a distinction between the first time that a place $\pi$ undergoes a deflate, in which case $(3bi)$ is used and for every later deflate on the place $\pi$, we use $(3bii)$ which does not transfer any tokens. This implies that that for every non-empty balloon created, there is only one opportunity for $\mathcal{V}'$ to transfer tokens away from places in $\Omega \times \{1, \cdots, N\} \times \Phi$.

The edges in $(4b)$ simulate a burst operation by allowing the removal of tokens from places which have not been transferred away, while setting the value of $\eta$ to † at the appropriate place to indicate that the balloon has been burst.

Finally, we only allow a move to the state $q_f$ if the $H$-component indicates that all non-empty balloons produced have been burst. A correct simulation ensures that no tokens remain in any of the places $\Omega \times \{1, \cdots, N\} \times \Phi \cup \Omega$. Thus $\mathcal{V}'$ can reach $s_2'$ from $s_1'$ if $\mathcal{V}$ can reach $s_2$ from $s_1$. Conversely, we note that the sequence of deflates on each non-empty balloon is tracked in the global state and as mentioned, there is only one opportunity for $\mathcal{V}'$ to correctly transfer tokens from the places $\Omega \times \{1, \cdots, N\} \times \Phi$ for each non-empty balloon. Thus, from any run $s_1' \xrightarrow{*} s_2'$ of $\mathcal{V}'$ we obtain a run $s_1 \xrightarrow{*} s_2$ of $\mathcal{V}$.

# E PROOFS FROM SECTION 6

## E.1 Proof of Lemma 6.1

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

PROOF. Clearly, a starving run of a DCPS has to be consistent. Suppose a DCPS has a consistent run $\rho$. Then there are configurations $c_1, c_2, \ldots$ and thread executions $e_1, e_2, \ldots$ that produce $\mathbf{m}_1, \mathbf{m}_2, \ldots$ so that the consistency conditions are met. In particular, for $V_p = \{\mathbf{m}_j \mid e_j \text{ has type } t\}$, the tuple $\mathfrak{V} = (V_t)_{t \in \mathcal{T}}$ is $\mathfrak{S}_{\mathcal{A}}$-consistent. This means, there exists a stack content $w \in \Gamma^*$ such that we can choose for each $j \geq 1$, a thread execution $e_j'$ (not necessarily occurring in $\rho$) that produces a multiset $\mathbf{m}_j'$ such that:

(1) $e_j'$ has the same type as $e_j$
(2) $\mathbf{m}_j \leq_1 \mathbf{m}_j'$, and
(3) $e_j'$ arrives in $w$ after executing $i$ segments.

The idea is now to replace executions $e_j$ with $e_j'$ to obtain a starving run. Replacing any individual $e_j$ by $e_j'$ would yield an infinite run, because $e_j'$ has the same type (so that state transition would still be possible), and $\mathbf{m}_j' \geq_1 \mathbf{m}_j$, so that the bag contents are supersets of the bag contents of the old run.

However, the resulting run might not be progressive: $e_j'$ might spawn an additional thread $(\gamma, 0)$ such that $(\gamma, 0)$ becomes active only finitely many times during $\rho$. In that case, the extra $(\gamma, 0)$ would be in the bag forever, but never scheduled.

To remedy this, we only replace all but finitely many of the $e_j$. Let $\Gamma_{\inf}$ be the set of those $\gamma \in \Gamma$ such that $(\gamma, 0)$ is scheduled infinitely many times during $\rho$. Since $\rho$ is progressive, we know that for some $k \geq 1$, each $e_j$ with $k \geq j$ only spawns threads from $\Gamma_{\inf}$. Now since $\mathbf{m}'_j \geq_1 \mathbf{m}_j$, this implies that for $j \geq k$, $\mathbf{m}'_j$ only contains threads $\gamma$ (together with the context-switch number when they are produced) such that $\gamma \in \Gamma_{\inf}$.

Therefore, we obtain a new run $\rho'$ of our DCPS by replacing each $e_j$ by $e'_j$ for all $j \geq k$. Then, the resulting run is progressive, because the additionally spawned threads all belong to $\Gamma_{\inf}$. Moreover, $\rho'$ is starving. $\blacksquare$

## E.2  Proof of Lemma 6.4

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.*

PROOF. If $\mathcal{A}$ has a starving run $\rho$, then it also has a consistent run by Lemma 6.1. Hence, there are a number $i \in [0, K]$, configurations $c_1, c_2, \ldots$ and thread executions $e_1, e_2, \ldots$ that produce $\mathbf{m}_1, \mathbf{m}_2, \ldots$ such that the conditions for a consistent run are satisfied. In particular, with $V_t = \{\mathbf{m}_j \mid e_j \text{ has type } t\}$, the tuple $\mathfrak{B} = (V_t)_{t \in \mathcal{T}}$ is $\mathfrak{S}_{\mathcal{A}}$-consistent. Then, by definition of $\mathfrak{S}_{\mathcal{A}}$-consistency, the tuple $\mathfrak{m} = \alpha_B(\mathfrak{B})$ is also $\mathfrak{S}_{\mathcal{A}}$-consistent. Therefore, the run $\rho$ is $(i, \mathfrak{m})$-starving.

Conversely, suppose $\rho$ is a shallow $(i, \mathfrak{m})$-starving run for some $i \in [0, K]$ and some $\mathfrak{S}$-consistent tuple $\mathfrak{m} \in \mathbb{P}([0, B]^\Lambda)^{\mathcal{T}}$. Then we have configurations $c_1, c_2, \ldots$, and thread executions $e_1, e_2, \ldots$ that produce $\mathbf{m}_1, \mathbf{m}_2, \ldots$ such that the conditions of $(i, \mathfrak{m})$-starvation are satisfied.

Let $V_t = \{\mathbf{m}_j \mid e_j \text{ has type } t\}$ and $\mathfrak{B} = (V_t)_{t \in \mathcal{T}}$. Since $\rho$ is shallow, we know that each $V_t$ is a finite set. Moreover, since $\mathfrak{m} = \alpha_B(\mathfrak{B})$ is $\mathfrak{S}_{\mathcal{A}}$-consistent, Theorem 6.2 tells us that $\mathfrak{B}$ is $\mathfrak{S}_{\mathcal{A}}$-consistent as well. Thus, $\rho$ is consistent. Therefore $\mathcal{A}$ also has a starving run by Lemma 6.1. $\blacksquare$

## E.3  Freezing DCPS

In this section we prove Lemma 6.5, which states that it is decidable whether a freezing DCPS $\mathcal{A}$ has a $K$-context bounded progressive run, and if such a run exists then we can always assume it to be shallow. To this end we construct a DCPS $\mathcal{A}$ that simulates $\mathcal{A}$ in a way that preserves progressiveness and shallowness, while using at most $2K + 1$ context switches. The stated properties for freezing DCPS then follow from our results on DCPS (Theorem 2.3, Lemma 4.3, and Corollary 4.5).

*Idea.* The only differences between DCPS and freezing DCPS are the presence of frozen threads (marked with a top of stack symbol in $\Gamma^\circledast$) in configurations, and the rule UNFREEZE. To simulate the former we just add $\Gamma^\circledast$ to the stack alphabet and introduce a few new transition rules that spawn the initial frozen thread at the start of each run. To simulate UNFREEZE for a rule $g \mapsto g' \lhd \gamma \circledast \gamma'$, we follow these steps starting in global state $g$:

- Resume a thread with $\gamma'$ as top of stack symbol, change it to $\gamma'^\circledast$ then swap out this thread.
- Resume a thread with $\gamma^\circledast$ as top of stack symbol, change it to $\gamma$ and go to $g'$ with this thread staying active.

While these steps change the stack symbols correctly and make the right thread active, there are two things of note here: Firstly, if $\gamma = \gamma'$ then the very thread swapped out at the end of the first step could be resumed at the start of the second step, which is undesired behavior. To remedy this, we add a second copy of $\Gamma^\circledast$ to our stack alphabet, which we call $\bar{\Gamma}^\circledast$. We modify the simulation so that each time a thread with top of stack symbol in $\Gamma^\circledast$ is swapped out in the first step, a thread with top of stack symbol in $\bar{\Gamma}^\circledast$ is resumed in the second step and vice-versa. Thus the two steps can no longer act on the very same thread, which fixes the issue.

Secondly, *freezing* a thread in the first step causes it to make a context switch, which does not match the specification of UNFREEZE. However, *unfreezing* a thread makes it active and therefore adds a context switch once the thread is swapped out again. Thus, a thread can be frozen and unfrozen a total number of $K + 1$ times each in the freezing DCPS $\mathcal{A}$. In the DCPS $\mathcal{A}'$ such a thread would then make $2K + 2$ context switches, with the last one happening right after it was active from being unfrozen the $(K + 1)$th time. This means that the highest context switch number it reaches while being active is $2K + 1$, which is exactly what we chose as the context switch bound for $\mathcal{A}'$.

With this increased context switch bound, we need to make sure that threads do not make more than $K + 1$ context switches in $\mathcal{A}'$, if they are resumed without being frozen. To this end we just artificially increase the context switch number by forcing an additional context switch every time a non-frozen thread is resumed. This mirrors the extra context switch caused by freezing, and therefore makes these threads behave correctly with a context switch bound of $2K + 1$ as well.

*Formal construction.* Let $\mathcal{A} = (G, \Gamma, \Delta, g_0, \gamma_0, \gamma_f)$ with $\Delta = \Delta_c \cup \Delta_i \cup \Delta_r \cup \Delta_t \cup \Delta_u$ be a freezing DCPS. We construct the DCPS $\mathcal{A}' = (G', \Gamma', \Delta', g_0', \gamma_0')$, where:

- $\Gamma' = \{\gamma_0'\} \cup \Gamma \cup \bar{\Gamma} \cup \Gamma^{\circledast} \cup \bar{\Gamma}^{\circledast}$ with $\bar{\Gamma} = \{\bar{\gamma} | \gamma \in \Gamma\}$, $\Gamma^{\circledast} = \{\gamma^{\circledast} | \gamma \in \Gamma\}$, and $\bar{\Gamma}^{\circledast} = \{\bar{\gamma}^{\circledast} | \gamma \in \Gamma\}$.
- $G' = G \cup \bar{G} \cup (G \times \Gamma^{\circledast}) \cup (G \times \bar{\Gamma}^{\circledast})$ with $\bar{G} = \{\bar{g} | g \in G\}$
- $\Delta' = \Delta_c' \cup \Delta_i' \cup \Delta_r' \cup \Delta_t'$ consists of the following transition rules:
 (1) For the initial configuration:
    (a) $g_0' \mapsto g_0' \triangleleft \gamma_0' \in \Delta_r'$.
    (b) $g_0' | \gamma_0' \hookrightarrow g_0' | \gamma_f^{\circledast} \triangleright \gamma_0 \in \Delta_c'$.
    (c) $g_0' | \gamma_f^{\circledast} \mapsto g_0 | \gamma_f^{\circledast} \in \Delta_i'$.
 (2) For each rule $g_1 \mapsto g_2 \triangleleft \gamma \in \Delta_r$:
    (a) $g_1 \mapsto \bar{g}_2 \triangleleft \gamma \in \Delta_r'$.
    (b) $\bar{g}_2 | \gamma \hookrightarrow \bar{g}_2 | \bar{\gamma} \in \Delta_c'$.
    (c) $\bar{g}_2 | \bar{\gamma} \mapsto \bar{g}_2 | \bar{\gamma} \in \Delta_i'$.
    (d) $\bar{g}_2 \mapsto g_2 \triangleleft \bar{\gamma} \in \Delta_r'$.
    (e) $g_2 | \bar{\gamma} \hookrightarrow g_2 | \gamma \in \Delta_c'$.
 (3) For each rule $g_1 \mapsto g_2 \triangleleft \gamma_1 \circledast \gamma_2 \in \Delta_u$:
    (a) $g_1 \mapsto (g_2, \gamma_1^{\circledast}) \triangleleft \gamma_2 \in \Delta_r'$.
    (b) $(g_2, \gamma_1^{\circledast}) | \gamma_2 \hookrightarrow (g_2, \gamma_1^{\circledast}) | \gamma_2 \in \Delta_c'$.
    (c) $(g_2, \gamma_1^{\circledast}) | \gamma_2 \mapsto (g_2, \gamma_1^{\circledast}) | \bar{\gamma}_2^{\circledast} \in \Delta_i'$.
    (d) $(g_2, \gamma_1^{\circledast}) \mapsto (g_2, \gamma_1^{\circledast}) \triangleleft \gamma_1^{\circledast} \in \Delta_r'$.
    (e) $(g_2, \gamma_1^{\circledast}) | \gamma_1^{\circledast} \hookrightarrow g_2 | \gamma_1 \in \Delta_c'$.
 (4) Furthermore, for each rule $g_1 \mapsto g_2 \triangleleft \gamma_1 \circledast \gamma_2 \in \Delta_u$:
    (a) $g_1 \mapsto (g_2, \bar{\gamma}_1^{\circledast}) \triangleleft \gamma_2 \in \Delta_r'$.
    (b) $(g_2, \bar{\gamma}_1^{\circledast}) | \gamma_2 \hookrightarrow (g_2, \bar{\gamma}_1^{\circledast}) | \gamma_2 \in \Delta_c'$.
    (c) $(g_2, \bar{\gamma}_1^{\circledast}) | \gamma_2 \mapsto (g_2, \bar{\gamma}_1^{\circledast}) | \gamma_2^{\circledast} \in \Delta_i'$.
    (d) $(g_2, \bar{\gamma}_1^{\circledast}) \mapsto (g_2, \bar{\gamma}_1^{\circledast}) \triangleleft \bar{\gamma}_1^{\circledast} \in \Delta_r'$.
    (e) $(g_2, \bar{\gamma}_1^{\circledast}) | \bar{\gamma}_1^{\circledast} \hookrightarrow g_2 | \gamma_1 \in \Delta_c'$.
 (5) Finally, $\Delta'$ contains some unaltered rules of $\Delta$:
    (a) $r \in \Delta_c'$ for each rule $r \in \Delta_c$.
    (b) $r \in \Delta_i'$ for each rule $r \in \Delta_i$.
    (c) $r \in \Delta_t'$ for each rule $r \in \Delta_t$.

LEMMA E.1. *The freezing DCPS $\mathcal{A}$ has a (shallow) progressive run that respects the context switch bound $K$ iff the DCPS $\mathcal{A}'$ has a (shallow) progressive run that respects the context switch bound $2K + 1$.*

PROOF. From a given infinite run of $\mathcal{A}$ we construct an infinite run of $\mathcal{A}'$ in such a way that progressiveness and shallowness are preserved. Then we argue that this construction can also be done backwards, starting with a run of $\mathcal{A}'$.

Let $\rho$ be an infinite run of $\mathcal{A}$. We construct the infinite run $\rho'$ of $\mathcal{A}'$ by induction on the length of a prefix of $\rho$. Say $\rho$ reaches the configuration $c = \langle g, t_a, \mathbf{m} \rangle$ after finitely many steps. Intuitively we want $\rho'$ to reach a configuration $c'$ that contains threads with the same stack contents as $c$, except for the top of stack symbol of the frozen thread, but with different context switch numbers. Any thread with cs-number $k$ in $c$ has cs-number $2k + 1$ in $c'$ if it was an active or frozen thread, and cs-number $2k$ otherwise. Formally $c' = \langle g, t_a', \mathbf{m} \rangle$, where:

- If $t_a = \#$ then $t_a' = \#$. If $t_a = (w, k)$ then $t_a' = (w, 2k + 1)$.
- For all $k \in \mathbb{N}$, $w \in \Gamma^*$: $\mathbf{m}'(w, 2k) = \mathbf{m}(w, k)$.
- For all $k \in \mathbb{N}$, $w \in \Gamma^*$, $\gamma \in \Gamma$: $\mathbf{m}'(\gamma^{\circledast} w, 2k + 1) = \mathbf{m}(\gamma^{\circledast} w, k)$ and $\mathbf{m}'(\bar{\gamma}^{\circledast} w, 2k + 1) = 0$, or $\mathbf{m}'(\bar{\gamma}^{\circledast} w, 2k + 1) = \mathbf{m}(\gamma^{\circledast} w, k)$ and $\mathbf{m}'(\gamma^{\circledast} w, 2k + 1) = 0$.
- For all $k \in \mathbb{N}$, $w \in \Gamma^*$, $\gamma \in \Gamma$: $\mathbf{m}'(w, 2k + 1) = \mathbf{m}'(\gamma^{\circledast} w, 2k) = \mathbf{m}'(\bar{\gamma}^{\circledast} w, 2k) = 0$.

The second to last bullet point means that for a frozen thread in $c$ with top of stack symbol $\gamma^{\circledast}$, the corresponding thread in $c'$ has either $\gamma^{\circledast}$ or $\bar{\gamma}^{\circledast}$ as its top of stack symbol. The last bullet point simply ensures that $c'$ contains no additional threads (that do not correspond to a thread of $c$).

Now to construct $\rho'$ inductively. If $c = \langle g_0, \#, [[(\gamma_0, 0)]] + [[(\gamma_f^{\circledast}, 0)]] \rangle$ is the initial configuration of $\mathcal{A}$ then we use the following transitions in $\rho'$ starting with the initial configuration of $\mathcal{A}'$:

$$\langle g_0', \#, [[(\gamma_0', 0)]] \rangle \overset{1a}{\longmapsto} \langle g_0', (\gamma_0', 0), \emptyset \rangle$$
$$\overset{1b}{\longrightarrow} \langle g_0', (\gamma_0', 0), [[(\gamma_0, 0)]] \rangle$$
$$\overset{1c}{\longmapsto} \langle g_0, \#, [[(\gamma_0, 0)]] + [[(\gamma_f^{\circledast}, 1)]] \rangle = c'.$$

Here and for all following transitions of $\mathcal{A}'$ we label each arrow with the corresponding transition rule of $\Delta'$. Since we reach the correct $c'$, this concludes the base case.

For the inductive case assume that we reach $c$ after at least one step on $\rho$. We make a case distinction regarding the transition $\widetilde{c} \to c$ or $\widetilde{c} \mapsto c$ of $\rho$ that reaches $c$:

**Case** $\langle g_1, \#, \mathbf{m} + [[(\gamma w, i)]] \rangle \mapsto \langle g_2, (\gamma w, i), \mathbf{m} \rangle$ **due to** $g_1 \mapsto g_2 \lhd \gamma \in \Delta_{\mathrm{r}}$**:**
We add the following transitions to $\rho'$, starting from the configuration obtained by applying the induction hypothesis to $\widetilde{c}$:

$$\langle g_1, \#, \mathbf{m}' + [[(\gamma w, 2i)]] \rangle \overset{2a}{\longmapsto} \langle \bar{g}_2, (\gamma w, 2i), \mathbf{m}' \rangle$$
$$\overset{2b}{\longrightarrow} \langle \bar{g}_2, (\bar{\gamma} w, 2i), \mathbf{m}' \rangle$$
$$\overset{2c}{\longmapsto} \langle \bar{g}_2, \#, \mathbf{m}' + [[(\bar{\gamma} w, 2i + 1))]] \rangle$$
$$\overset{2d}{\longmapsto} \langle g_2, (\bar{\gamma} w, 2i + 1), \mathbf{m}' \rangle$$
$$\overset{2e}{\longrightarrow} \langle g_2, (\gamma w, 2i + 1), \mathbf{m}' \rangle = c'.$$

**Case** $\langle g_1, \#, \mathbf{m} + [[\gamma_1^{\circledast} w_1, i]] + [[\gamma_2 w_2, j]] \rangle \mapsto \langle g_2, (\gamma_1 w_1, i), \mathbf{m} + [[\gamma_2^{\circledast} w_2, j]] \rangle$
**due to** $g_1 \mapsto g_2 \lhd \gamma_1 \circledast \gamma_2 \in \Delta_{\mathrm{u}}$**:**
We again start from the configuration obtained by applying the induction hypothesis to $\widetilde{c}$. However, we need to make a case distinction based on the top of stack symbol of the thread

in this configuration that corresponds to the frozen thread of $\widetilde{c}$. If this symbol is in $\Gamma^{\circledast}$, we add the following transitions to $\rho'$:

$$\langle g_1, \#, \mathbf{m}' + [[\gamma_1^{\circledast} w_1, 2i+1]] + [[(\gamma_2 w_2, 2j)]]\rangle$$

$$\overset{3a}{\longmapsto} \langle(g_2, \gamma_1^{\circledast}), (\gamma_2 w_2, 2j), \mathbf{m}' + [[\gamma_1^{\circledast} w_1, 2i+1]]\rangle$$

$$\overset{3b}{\longrightarrow} \langle(g_2, \gamma_1^{\circledast}), (\bar{\gamma}_2^{\circledast} w_2, 2j), \mathbf{m}' + [[\gamma_1^{\circledast} w_1, 2i+1]]\rangle$$

$$\overset{3c}{\longmapsto} \langle(g_2, \gamma_1^{\circledast}), \#, \mathbf{m}' + [[\gamma_1^{\circledast} w_1, 2i+1]] + [[(\bar{\gamma}_2^{\circledast} w_2, 2j+1)]]\rangle$$

$$\overset{3d}{\longmapsto} \langle(g_2, \gamma_1^{\circledast}), (\gamma_1^{\circledast} w_1, 2i+1), \mathbf{m}' + [[(\bar{\gamma}_2^{\circledast} w_2, 2j+1)]]\rangle$$

$$\overset{3e}{\longrightarrow} \langle g_2, (\gamma_1 w_1, 2i+1), \mathbf{m}' + [[(\bar{\gamma}_2^{\circledast} w_2, 2j+1)]]\rangle = c'.$$

Otherwise, if the symbol is in $\bar{\Gamma}^{\circledast}$, we add the following transitions to $\rho'$:

$$\langle g_1, \#, \mathbf{m}' + [[\bar{\gamma}_1^{\circledast} w_1, 2i+1]] + [[(\gamma_2 w_2, 2j)]]\rangle$$

$$\overset{4a}{\longmapsto} \langle(g_2, \bar{\gamma}_1^{\circledast}), (\gamma_2 w_2, 2j), \mathbf{m}' + [[\bar{\gamma}_1^{\circledast} w_1, 2i+1]]\rangle$$

$$\overset{4b}{\longrightarrow} \langle(g_2, \bar{\gamma}_1^{\circledast}), (\gamma_2^{\circledast} w_2, 2j), \mathbf{m}' + [[\bar{\gamma}_1^{\circledast} w_1, 2i+1]]\rangle$$

$$\overset{4c}{\longmapsto} \langle(g_2, \bar{\gamma}_1^{\circledast}), \#, \mathbf{m}' + [[\bar{\gamma}_1^{\circledast} w_1, 2i+1]] + [[(\gamma_2^{\circledast} w_2, 2j+1)]]\rangle$$

$$\overset{4d}{\longmapsto} \langle(g_2, \bar{\gamma}_1^{\circledast}), (\bar{\gamma}_1^{\circledast} w_1, 2i+1), \mathbf{m}' + [[(\gamma_2^{\circledast} w_2, 2j+1)]]\rangle$$

$$\overset{4e}{\longrightarrow} \langle g_2, (\gamma_1 w_1, 2i+1), \mathbf{m}' + [[(\gamma_2^{\circledast} w_2, 2j+1)]]\rangle = c'.$$

The remaining four cases use the same transition rules for both $\rho$ and $\rho'$, as the rules defined in (5) are ones already present in $\Delta$:

**Case** $\langle g_1, (\gamma w, i), \mathbf{m}\rangle \rightarrow \langle g_2, (w_2 w_1, i), \mathbf{m}\rangle$ **due to** $g_1 | \gamma \hookrightarrow g_2 | w_2 \in \Delta_c$**:**

$$\langle g_1, (\gamma w, 2i+1), \mathbf{m}'\rangle \overset{5a}{\longrightarrow} \langle g_2, (w_2 w_1, 2i+1), \mathbf{m}'\rangle = c'.$$

**Case** $\langle g_1, (\gamma_1 w_1, i), \mathbf{m}\rangle \rightarrow \langle g_2, (w_2 w_1, i), \mathbf{m} + [[(\gamma_2, 0)]]\rangle$ **due to** $g_1 | \gamma_1 \hookrightarrow g_2 | w_2 \triangleright \gamma_2 \in \Delta_c$**:**

$$\langle g_1, (\gamma_1 w_1, 2i+1), \mathbf{m}'\rangle \overset{5a}{\longrightarrow} \langle g_2, (w_2 w_1, 2i+1), \mathbf{m}' + [[(\gamma_2, 0)]]\rangle = c'.$$

**Case** $\langle g_1, (\gamma w_1, i), \mathbf{m}\rangle \mapsto \langle g_2, \#, \mathbf{m} + [[(w_2 w_1, i+1)]]\rangle$ **due to** $g_1 | \gamma \mapsto g_2 | w_2 \in \Delta_i$**:**

$$\langle g_1, (\gamma w_1, 2i+1), \mathbf{m}'\rangle \overset{5b}{\longmapsto} \langle g_2, \#, \mathbf{m}' + [[(w_2 w_1, 2i+2)]]\rangle = c'.$$

**Case** $\langle g_1, (\varepsilon, i), \mathbf{m}\rangle \mapsto \langle g_2, \#, \mathbf{m}\rangle$ **due to** $g_1 \mapsto g_2 \in \Delta_t$**:**

$$\langle g_1, (\varepsilon, 2i+1), \mathbf{m}'\rangle \overset{5c}{\longmapsto} \langle g_2, \#, \mathbf{m}'\rangle = c'.$$

Each $c'$ reached in any of the cases correctly corresponds to $c$ as described above. Spawn boundedness is equivalent for $\rho$ and $\rho'$ since corresponding segments where a thread is active make the same amount of spawns in both runs. Progressiveness is also equivalent: Every transition that makes a thread active in $\rho$ (either by resuming or unfreezing) results in a corresponding thread becoming active in $\rho'$. Conversely every two resume transitions due to rules from (2) in $\rho'$ correspond to a single resume transition in $\rho$, and each resume transition due to (3d) or (4d) corresponds to an unfreezing transition in $\rho$. Resume transitions due to (3a) or (4a) cause a thread to gain a top of stack symbol from $\Gamma^{\circledast} \cup \bar{\Gamma}^{\circledast}$, which can be the case for only one thread at a time. Thus, if $\rho'$ is progressive, this very thread has to be resumed, causing a later resume transition due to (3d) or

(4d), which we already discussed. Finally, termination occurs at context switch number $k$ in $\rho$ iff it occurs at $2k + 1$ in $\rho'$, which matches the relationship between the two context switch bounds for these runs.

For the other direction we start with an infinite run $\rho'$ of $\mathcal{A}'$. Observe that $\rho'$ decomposes into infixes that have the same form as the transition sequences obtained during the construction for the other direction. This is because for each of the alphabets $\bar{\Gamma}$, $\Gamma^{\circledast}$, and $\bar{\Gamma}^{\circledast}$ there is always at most one thread that has a top of stack symbol in that alphabet; which causes the transition rules defined for $\mathcal{A}'$ to not allow for any other types of behavior. Thus, we can do the previous construction backwards and obtain a run $\rho$ of $\mathcal{A}$ that has equivalent progressiveness and shallowness properties when compared to $\rho'$. ∎

### E.4 Detailed construction of $\mathcal{A}_{(i,\mathfrak{u})}$

We construct a freezing DCPS $\mathcal{A}_{(i,\mathfrak{u})} = (G', \Gamma', \Delta', g_0', \gamma_0', \gamma_\dagger)$. Here, $g_0'$ is a fresh state and $\gamma_0'$ are fresh stack symbols, which are both used for initialization. Moreover, $\gamma_\dagger$ is a fresh stack symbol that will be the top of stack of the initially frozen thread.

In addition to the stack of a thread in $\mathcal{A}$, a thread in $\mathcal{A}_{(i,\mathfrak{u})}$ tracks some extra information $(t, j, \bar{\mathbf{m}}, \bar{\mathbf{n}})$, where

- $t$ is the type of the current thread execution,
- $j$ is the number of segments that have been completed,
- $\bar{\mathbf{m}}$ is the guess for $\alpha_B(\mathbf{m})$, where $\mathbf{m}$ is the production of the thread execution,
- $\bar{\mathbf{n}}$ is $\alpha_B(\mathbf{n})$, where $\mathbf{n} \in \mathbb{M}[\Lambda]$ is the multiset that has been produced so far.

When a thread is inactive, this extra information is stored on its top of stack symbol. For this, we need the new alphabet $\Gamma' = \Gamma \cup \tilde{\Gamma} \cup \{\gamma_0', \gamma_\dagger\}$, where

$$\tilde{\Gamma} = \{(\gamma, t, j, \bar{\mathbf{m}}, \bar{\mathbf{n}}) \mid \gamma \in \Gamma, \ t \text{ is a type}, \ \bar{\mathbf{m}}, \bar{\mathbf{n}} \in [0, B]^\Lambda\}.$$

While a thread is active, this extra information is stored in the global state. This makes it easier to update it, e.g. when the stack is popped. Moreover, we need a global state $\widehat{g}$ for each $g$, which enforce that the information is transfered from the stack to the global state. In order to execute the initially frozen thread with top-of-stack $\gamma_\dagger$, we need global states $g^{\dagger,j}$ for $g \in G$ and $j \in [0, K]$.

Finally, we need special global states $g_j'$ for $j \in [0, K]$ to execute an initial helper thread with top-of-stack $\gamma_0'$. Thus, we have $G' = G \cup \tilde{G} \cup \widehat{G} \cup \{g_j', g^{\dagger,j} \mid j \in [0, K]\}$, where

$$\tilde{G} = \{(g, t, j, \bar{\mathbf{m}}, \bar{\mathbf{n}}) \mid g \in G, \ t \text{ is a type}, \ j \in [0, K], \ \bar{\mathbf{m}}, \bar{\mathbf{n}} \in [0, B]^\Lambda\},$$

$$\widehat{G} = \{\widehat{g} \mid g \in G\}.$$

We now describe the transition rules of $\mathcal{A}_{(i,\mathfrak{u})}$.

*Rules for initialization.* We begin the description of a few rules that serve to initialize our freezing DCPS. The initial configuration of our freezing DCPS is $\langle g_0', \bot, [[(\gamma_0', 0), (\gamma_\dagger^{\circledast}, 0)]] \rangle$. Since in our simulation, we need that each thread in the bag is already annotated with its extra information, we use a helper thread with top-of-stack $\gamma_0'$ to (i) spawn a thread simulating $\gamma_0$ and also (ii) guess its extra information. Thus we have a resume rule

$$g_0' \mapsto g_1' \lhd \gamma_0'$$

to resume $(\gamma_0', 0)$. Then in this new thread, we spawn $(\gamma_0, 0)$, but with extra information. Thus, we have a creation rule

$$g_0' | \gamma_0' \hookrightarrow g_1' | \gamma_0' \rhd (\gamma_0, t, 0, \bar{\mathbf{m}}, \mathbf{0})$$

for every type $t$ and $\bar{\mathbf{m}} \in [0, B]^{\Lambda}$. After this, our helper thread has to complete $K$ context-switches. This means, it has a interrupt rules

$$g'_j|\gamma'_0 \mapsto g'_{j+1}|\gamma'_0$$

for $j \in [1, K-1]$ and a rule to remove $\gamma'_0$:

$$g'_K|\gamma'_0 \hookrightarrow g'_K|\varepsilon$$

Then, from $g'_K$ with an empty stack, we can only use the termination rule

$$g'_K \mapsto g_0$$

which enters the global state that corresponds to the initial global state $g_0$ of $\mathcal{A}$.

*Creation rules.* An internal action of a thread is simulated in the obious way. For each rule $g|\gamma \hookrightarrow g'|w'$, we have a rule

$$(g, t, j, \bar{\mathbf{m}}, \bar{\mathbf{n}})|\gamma \hookrightarrow (g, t, j, \bar{\mathbf{m}}, \bar{\mathbf{n}})|w'$$

for every type $t$, $j \in [0, K]$, and $\bar{\mathbf{m}}, \bar{\mathbf{n}} \in [0, B]^{\Lambda}$.

When our DCPS spawns a new thread, it immediately guesses its type and production abstraction $\bar{\mathbf{m}}$. Moreover, it sets its segment counter to $0$ and sets $\bar{\mathbf{n}}$ to $\mathbf{0}$. Hence, for every creation rule $g|\gamma \hookrightarrow g'|w' \rhd \gamma'$, we have a rule

$$(g, t, j, \bar{\mathbf{m}}, \bar{\mathbf{n}})|\gamma \hookrightarrow (g, t, j, \bar{\mathbf{m}}, \alpha_B(\bar{\mathbf{n}} + [[(\gamma, j)]]))|w' \rhd (\gamma', p', 0, \bar{\mathbf{m}}', \mathbf{0})$$

for types $t, t'$, $j \in [0, K]$, and $\bar{\mathbf{m}}, \bar{\mathbf{m}}', \bar{\mathbf{n}} \in [0, B]^{\Lambda}$. Note that changing $\bar{\mathbf{n}}$ to $\alpha_B(\bar{\mathbf{n}} + [[(\gamma, j)]])$ records that the current thread has spawned a thread $\gamma$ in segment $j$. Note that we perform this guessing of extra information with every newly spawned thread, not just those that will be frozen and hence yield the executions $e_1, e_2, \ldots$. Therefore, there is no requirement here that $\bar{\mathbf{m}}$ belong to $U_p$ where $\mathfrak{u} = (U_t)_{t \in \mathcal{T}}$.

*Interruption rules.* When we interrupt a thread, then its extra information is transferred from the global state to the top of stack and the segment counter $j$ is incremented. Thus, for every interrupt rule $g|\gamma \mapsto g'|w'$ of $\mathcal{A}$, we write $w' = \gamma''w''$ (recall that $1 \leq |w'| \leq 2$) and include rules

$$(g, t, j, \bar{\mathbf{m}}, \bar{\mathbf{n}})|\gamma \mapsto g|(\gamma'', t, j+1, \bar{\mathbf{m}}, \bar{\mathbf{n}})w''$$

for every type $t$, $j \in [0, K-1]$, and $\bar{\mathbf{m}}, \bar{\mathbf{n}} \in [0, B]^{\Lambda}$.

*Resumption rules.* In order to simulate a resumption rule $g \mapsto g' \lhd \gamma$, we resume some thread with $\gamma$ (and extra information) as top of stack. The transfer of the extra finformation cannot be done in the same step, so we have an additional state $\widehat{g'}$ in which this transfer is carried out. Hence, for every resumption rule $g \mapsto g' \lhd \gamma$, we have rules

$$g \mapsto \widehat{g'} \lhd (\gamma, p, j, \bar{\mathbf{m}}, \bar{\mathbf{n}})$$

for each type $t$, $j \in [0, K]$, $\bar{\mathbf{m}}, \bar{\mathbf{n}} \in [0, B]^{\Lambda}$. In $\widehat{g'}$, we then transfer the extra information into the global state. Thus, we have

$$\widehat{g}|(\gamma, t, j, \bar{\mathbf{m}}, \bar{\mathbf{n}}) \hookrightarrow (g, t, j, \bar{\mathbf{m}}, \bar{\mathbf{n}})|\gamma,$$

which exist for each $g \in G$, $\gamma \in \Gamma$, $\bar{\mathbf{m}}, \bar{\mathbf{n}} \in [0, B]^{\Lambda}$, $t \in \mathcal{T}$, and $j \in [0, K]$.

*Termination rules.* When we terminate a thread, we check that the two components $\bar{\mathbf{m}}$ and $\bar{\mathbf{n}}$ in the extra information match. Hence, for each termination rule $g \mapsto g'$, we include a rule

$$(g, t, j, \bar{\mathbf{m}}, \bar{\mathbf{m}}) \mapsto g'$$

for each type $t$, $j \in [0, K]$, and $\bar{\mathbf{m}} \in [0, B]^{\Lambda}$.

*Unfreezing rules.* Using freezing and unfreezing, we make sure that there are thread executions $e_1, e_2, \ldots$ that satisfy the conditions of a $(i, \mathfrak{u})$-starving run. This works as follows. Initially, we have the frozen thread $\gamma_\dagger$. To satisfy progressiveness, a run of $\mathcal{A}_{(i,\mathfrak{u})}$ must at some point unfreeze (and switch to) $\gamma_\dagger$. During this unfreeze, we make sure hat there exists a thread that can play the role of $e_1$.

Thus, to unfreeze $\gamma_\dagger$, we have to freeze a thread of some type $t$ where $\bar{\mathbf{m}}$ belongs to $U_t$:

$$g \mapsto g^{\dagger,0} \lhd \gamma_\dagger \circledast (\gamma, t, i, \bar{\mathbf{m}}, \bar{\mathbf{n}})$$

for every $g \in G$, $t \in \mathcal{T}$, $\bar{\mathbf{m}} \in U_p$. The state $g^{\dagger,0}$ is a copy of $g$ in which we can only complete the execution of the $\gamma_\dagger$ thread and then return to $g$. This means, we have interrupt rules

$$g^{\dagger,j}|\gamma_\dagger \mapsto g^{\dagger,j+1}|\gamma_\dagger$$

for $j \in [0, K-1]$ and $g \in G$, and resume rules

$$g^{\dagger,j}|\gamma_\dagger \mapsto g^{\dagger,j}|\gamma_\dagger$$

for $j \in [1, K]$ and $g \in G$, a rule to empty the stack in the last segment:

$$g^{\dagger,K}|\gamma_\dagger \hookleftarrow g^{\dagger,K}|\varepsilon$$

for $g \in G$ and finally a termination rule

$$g^{\dagger,K} \mapsto g$$

so that the simulation of $\mathcal{A}$ can continue.

After this, the new frozen thread with top of stack $(\gamma, t, i, \bar{\mathbf{m}}, \bar{\mathbf{n}})$ has to be resumed (and thus unfrozen) at some point. To make sure that at that point, there is a thread that can play the role of $e_2$. Therefore, to unfreeze (and thus resume) a thread with top of stack $(\gamma, t, i, \bar{\mathbf{m}}, \bar{\mathbf{n}})$, we freeze a thread $(\gamma', t', i, \bar{\mathbf{m}}', \bar{\mathbf{n}}')$ with $\bar{\mathbf{m}}' \in U_{t'}$. Note that unfreezing always happens with context-switch number $i$, because the executions $e_1, e_2, \ldots$ have to be in their $i$-th segment in the configurations $c_1, c_2, \ldots$. Thus, we have

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

for each resume rule $g \mapsto g' \lhd \gamma$, type $t$, $\bar{\mathbf{m}} \in U_p$, and $\bar{\mathbf{n}} \in [0, B]^\Lambda$, provided that $g$ is the state specified in $t$ to be entered in the $i$-th segment.

### E.5  Proof of Lemma 6.7

In this section, we prove Lemma 6.7. It will be convenient to use a slightly modified definition of pushdown automata for this.

*Pushdown automata with output.* If $\Gamma$ is an alphabet, we define $\bar{\Gamma} = \{\bar{\gamma} \mid \gamma \in \Gamma\}$. Moreover, if $x = \bar{\gamma}$, then we define $\bar{x} = \gamma$. For a word $v \in (\Gamma \cup \bar{\Gamma})^*$, $v = v_1 \cdots v_n$, $v_1, \ldots, v_n \in \Gamma \cup \bar{\Gamma}$, we set $\bar{v} = \bar{v}_n \cdots \bar{v}_1$. A *pushdown automaton with output* is a tuple $\mathcal{A} = (Q, \Gamma, \Lambda, E, q_0, q_f)$, where $Q$ is a finite set of *states*, $\Gamma$ is its *stack alphabet*, $\Lambda$ is its *output alphabet*, $E \subseteq Q \times (\Gamma \cup \bar{\Gamma} \cup \{\varepsilon\}) \times \mathbb{M}[\Lambda] \times Q$ is a finite set of *edges*, $q_0 \in Q$ is its *initial state*, and $F \subseteq Q$ is its set of *final states*. A *configuration* of $\mathcal{A}$ is a triple $(q, w, \mathbf{m})$ with $q \in Q$, $w \in \Gamma^*$, and $\mathbf{m} \in \mathbb{M}[\Lambda]$. For configurations $(q, w, \mathbf{m})$ and $(q', w', \mathbf{m}')$, we write $(q, w, \mathbf{m}) \to (q', w', \mathbf{m}')$ if there is an edge $(q, u, \mathbf{n}, q')$ in $\mathcal{A}$ such that $\mathbf{m}' = \mathbf{m} + \mathbf{n}$ and (i) if $v = \varepsilon$, then $w' = w$, (ii) if $v \in \Gamma$, then $w' = wv$ and (iii) if $v = \bar{\gamma}$ for $\gamma \in \Gamma$, then $w = w'\gamma$. By $\to^*$, we denote the reflexive transitive closure of $\to$.

For a pushdown automata with output $\mathcal{A}$ and a state $q$, we define

$$S_{\mathcal{A},q} = \{(w, \mathbf{m}) \in \Gamma^* \times \mathbb{M}[\Lambda] \mid (q_0, \varepsilon, \mathbf{0}) \to^* (q, w, \mathbf{m}') \to^* (q_f, \varepsilon, \mathbf{m}) \text{ for some } \mathbf{m}' \in \mathbb{M}[\Lambda]\}$$

In other words, $S_{\mathcal{A},q}$ collects those pairs $(w, \mathbf{m})$ such that $\mathcal{A}$ has a run that visits the state $q$ with stack content $w$, and the whole run outputs $\mathbf{m}$. Clearly, Lemma 6.7 is a consequence of the following:

LEMMA E.2. *Given a pushdown automaton with output $\mathcal{A}$ and a state $q$, the set $S_{\mathcal{A},q}$ is effectively rational.*

In the proof of Lemma E.2, it will be convenient to argue about the dual pushdown automaton. If $\mathcal{A}$ is a pushdown automaton with output, then its *dual automaton*, denoted $\bar{\mathcal{A}}$, is obtained from $\mathcal{A}$ by changing each edge $(p, u, \mathbf{m}, q)$ into $(q, \bar{u}, \mathbf{m}, p)$, and switching the initial and final state, $q_0$, and $q_f$. Moreover, we define

$$I_{\mathcal{A},q} = \{(w, \mathbf{m}) \in \Gamma^* \times \mathbb{M}[\Lambda] \mid (q_0, \varepsilon) \to^* (q, w, \mathbf{m})\},$$

which is a one-sided version of $S_{\mathcal{A},q}$: In the right component, we only collect the multiset output until we reach $q$ and $w$. However, using the dual automaton, we can construct $S_{\mathcal{A},q}$ from the sets $I_{\mathcal{A},q}$. For subsets $S, T \subseteq \Gamma^* \times \mathbb{M}[\Lambda]$, we define

$$S \otimes T = \{(w, \mathbf{m}_1 + \mathbf{m}_2) \in \Gamma^* \times \mathbb{M}[\Lambda] \mid (w, \mathbf{m}_1) \in S, \ (w, \mathbf{m}_2) \in T\}.$$

Then clearly $S_{\mathcal{A},q} = I_{\mathcal{A},q} \otimes I_{\bar{\mathcal{A}},q}$. The next lemma follows using a simple product construction.

LEMMA E.3. *Given rational subsets $S, T \subseteq \Gamma^* \times \mathbb{M}[\Lambda]$, the set $S \otimes T$ is effectively rational.*

Because of $S_{\mathcal{A},q} = I_{\mathcal{A},q} \otimes I_{\bar{\mathcal{A}},q}$, Lemma E.2 is a direct consequence of the following.

LEMMA E.4. *Given $\mathcal{A}$ and $q$, the set $I_{\mathcal{A},q}$ is effectively rational.*

PROOF. Roughly speaking, we do the following. For each pair of states $p, p'$, we look at the set $K_{p,p'} \subseteq \mathbb{M}[\Lambda]$ of outputs that can be produced in a computation that goes from $(p, w)$ to $(p', w)$ without ever removing a letter from $w$. Note that this set does not depend on $w$. Since $K_{p,p'}$ is semi-linear, there is a finite automaton $\mathcal{B}_{p,p'}$ that can produce $K_{p,p'}$. We glue in between $p$ and $p'$ in $\mathcal{A}$. In the resulting pushdown automaton with output $\mathcal{A}'$, we can then observe that every configuration is reachable without ever performing a pop operation. Therefore, removing all pop operations from $\mathcal{A}'$ and making $q$ the only final state yields an automaton over $\Gamma^* \times \mathbb{M}[\Lambda]$ that accepts $I_{\mathcal{A},q}$.

Let us do this in detail. Let $\mathcal{A} = (Q, \Gamma, \Lambda, E, q_0, q_f)$ be a pushdown automaton with output. For each pair of states $p, p' \in Q$, we define

$$K_{p,p'} = \{\mathbf{m} \in \mathbb{M}[\Lambda] \mid (p, \varepsilon, \mathbf{0}) \to^* (p', \varepsilon, \mathbf{m})\}.$$

It follows from Parikh's theorem that each set $K_{p,p'}$ is semi-linear. In particular, we can an automaton $\mathcal{B}_{p,p'}$ over $\Gamma^* \times \mathbb{M}[\Lambda]$ that accepts $\{\varepsilon\} \times K_{p,p'}$.

Let $\mathcal{A}'$ be the pushdown automaton with output obtained from $\mathcal{A}$ by glueing in, between any pair $p, p'$ of states, the automaton $\mathcal{B}_{p,p'}$. Observe that for any reachable configuration $(p, w, \mathbf{m})$ of $\mathcal{A}$, we can reach $(p, w, \mathbf{m})$ in $\mathcal{A}'$ without using pop transitions: If in a run there is a transition that pops some $\gamma$, we can replace the part of the run that pushes that $\gamma$, then performs other instructions, and finally pops $\gamma$, with a run in some $\mathcal{B}_{p,p'}$. Conversely, any configuration reachable in $\mathcal{A}'$ in a state that already exists in $Q$, is also reachable in $\mathcal{A}$.

Now let $\mathcal{A}''$ be the pushdown automaton with output obtained from $\mathcal{A}'$ by removing all pop transitions. According to our observation, $\mathcal{A}''$ has the same set of reachable configurations in $Q$ as $\mathcal{A}$. Since $\mathcal{A}''$ has no pop transitions, it is in fact an automaton over $\Gamma^* \times \mathbb{M}[\Lambda]$. Hence, if we make $q$ the final state, we obtain an automaton over $\Gamma^* \times \mathbb{M}[\Lambda]$ for the set $I_{\mathcal{A},q}$. ∎

## E.6 Proof of Lemma 6.11

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.*

PROOF. Suppose the rational subsets are given by automata $\mathcal{A}_1, \ldots, \mathcal{A}_k$. By introducing intermediate states, we may clearly assume that every edge in these automata either reads a letter from $\Gamma$ or a singleton multiset from $\Lambda$, meaning every edge either belongs to $Q \times \Gamma \times \{\mathbf{0}\} \times Q$ or is of the form $(p, \varepsilon, \mathbf{m}, q)$ with states $p, q$ and $|\mathbf{m}| \leq 1$. Let $n$ be an upper bound on the number of states of each $\mathcal{A}_i$, $i \in [1, k]$. Let $M$ be the bound from Lemma 6.10 and let $B = M(n + 1)$.

Now let $\mathfrak{V} = (V_1, \ldots, V_k)$ be a tuple of subsets of $\mathbb{M}[\Lambda]$ that admits an $\mathfrak{S}$-consistency witness $w \in \Gamma^*$. Moreover, let $\mathbf{m} \in \mathbb{M}[\Lambda]$ with $\alpha_B(\mathbf{m}) \in V_i$. We prove the lemma by constructing a word $\bar{w} \in \Gamma^*$ with $w \leq_{\mathfrak{S}} \bar{w}$ such that $\mathbf{m} \in S_i\!\downarrow_{\bar{w}}$. This implies the lemma, because $\bar{w}$ witnesses $\mathfrak{S}$-consistency of the tuple $\mathfrak{V}' = (V_1', \ldots, V_k')$ with $V_i' = V_i \cup \{\mathbf{m}\}$ and $V_j' = V_j$ for $j \neq i$.

Since $\alpha_B(\mathbf{m}) \in V_i'$ and $w$ is a $\mathfrak{S}$-consistency witness, we know that $\alpha_B(\mathbf{m}) \in S_i\!\downarrow_w$. This means, there is a multiset $\mathbf{m}' \in \mathbb{M}[\Lambda]$ with $\mathbf{m}' \geq_1 \alpha_B(\mathbf{m})$ and $(w, \mathbf{m}') \in S_i$.

Observe that if we had $\mathbf{m}' \geq_1 \mathbf{m}$, we could just choose $w' = w$. Moreover, in those coordinates $c \in \Lambda$ where $\mathbf{m}(c) < B$, we already know that $\mathbf{m}'(c) \geq \mathbf{m}(c)$, because $\mathbf{m}' \geq \alpha_B(\mathbf{m})$. For those coordinates $c$ with $\mathbf{m}(c) \geq B$, we will obtain $\bar{w}$ by pumping an infix in $w$.

Let $c \in \Lambda$ with $\mathbf{m}(c) \geq B$. Then $\mathbf{m}'(c) \geq B$ and therefore the pair $(w, \mathbf{m}')$ is accepted on a run

$$q_0 \xrightarrow{(u_0, \mathbf{m}_0)} q_1 \xrightarrow{(u_1, \mathbf{m}_1)} \cdots \xrightarrow{(u_B, \mathbf{m}_B)} q_B \xrightarrow{(u_{B+1}, \mathbf{m}_{B+1})} q_{B+1}$$

where $w = u_0 \cdots u_{B+1}$, $\mathbf{m}' = \mathbf{m}_0 + \cdots + \mathbf{m}_{B+1}$, and $\mathbf{m}_j(c) \geq 1$ for each $j = 1, \ldots, B$. Since $B = Mn$, there is a state $p$ of $\mathcal{A}_i$ that appears at least $M$ times in the sequence $q_1, \ldots, q_B$. This means, we have a run

$$q_0 \xrightarrow{(v_0, \mathbf{n}_0)} p \xrightarrow{(v_1, \mathbf{n}_1)} p \xrightarrow{(v_2, \mathbf{n}_2)} \cdots p \xrightarrow{(v_M, \mathbf{n}_M)} q_{B+1}$$

with $w = v_0 \cdots v_M$, $\mathbf{m}' = \mathbf{n}_0 + \cdots + \mathbf{n}_M$, and $\mathbf{n}_j(c) \geq 1$ for each $j = 1, \ldots, M - 1$.

We claim that we can write $w = xyz$ such that $w = xyz \leq_{\mathfrak{S}} xy^\ell z$ for every $\ell$ and that there is a run $q_0 \xrightarrow{(x, \hat{\mathbf{n}}_1)} p \xrightarrow{(y, \hat{\mathbf{n}}_2)} p \xrightarrow{(z, \hat{\mathbf{n}}_3)} q_{B+1}$ with $\hat{\mathbf{n}}_2(c) \geq 1$. We distinguish two cases.

(1) First, suppose that there is a $j \in \{1, \ldots, M - 1\}$ with $v_j = \varepsilon$. Then we can choose this $v_j$ as the $y$ in the decomposition $w = xyz$, which is clearly as desired.

(2) Suppose that $v_j \neq \varepsilon$ for every $j \in \{1, \ldots, M - 1\}$. Then the $M$ positions in the decomposition $w = v_0 \cdots v_M$ are pairwise distinct and we can apply Lemma 6.10. It clearly yields a decomposition of $w$ as desired in our claim.

Hence, the claim holds in any case. If we now choose $\ell$ high enough, then we find a run of $\mathcal{A}_i$ on $(xy^\ell z, \bar{\mathbf{m}}_c) = (xy^\ell z, \mathbf{m}' + \ell \cdot \bar{\mathbf{n}}_2)$ where $\bar{\mathbf{m}}_c(c) \geq \mathbf{m}(c)$ and $\bar{\mathbf{m}}_c \geq_1 \mathbf{m}'$. If we repeat this step for each $c \in \Lambda$ with $\mathbf{m}(c) > B$, we arrive at a word $\bar{w}$ and a multiset $\bar{\mathbf{m}}$ with $\mathbf{m} \leq_1 \bar{\mathbf{m}}$ and $w \leq_{\mathfrak{S}} \bar{w}$ and $(\bar{w}, \bar{\mathbf{m}}) \in S_i$. This implies $\mathbf{m} \in S_i\!\downarrow_{\bar{w}}$. ■

## E.7 Proof of Proposition 6.9

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.*

PROOF. Suppose $\mathfrak{V} = (V_1, \ldots, V_k)$ is a tuple of subsets of $\mathbb{M}[\Lambda]$. Clearly, if $\mathfrak{V}$ is $\mathfrak{S}$-consistent, then so is $\alpha_B(\mathfrak{V})$. The converse follows from Lemma 6.11: Since the sets $V_1, \ldots, V_k$ are finite, we can start with $\alpha_B(\mathfrak{V})$ and successively add each multiset occurring in some $V_i$, without affecting

$\mathfrak{S}$-consistency. Then we arrive at a $\mathfrak{S}$-consistent tuple $\mathfrak{V}' = (V'_1, \ldots, V'_k)$ with $V_i \subseteq V'_i$ for $i \in [1, k]$. In particular, $\mathfrak{V}$ is $\mathfrak{S}$-consistent. ∎