Priority Downward Closures

Ashwani Anand
Max Planck Institute for Software Systems (MPI-SWS), Kaiserslautern, Germany

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

Abstract

When a system sends messages through a lossy channel, then the language encoding all sequences of messages can be abstracted by its downward closure, i.e. the set of all (not necessarily contiguous) subwords. This is useful because even if the system has infinitely many states, its downward closure is a regular language. However, if the channel has congestion control based on priorities assigned to the messages, then we need a finer abstraction: The downward closure with respect to the priority embedding. As for subword-based downward closures, one can also show that these priority downward closures are always regular.

While computing finite automata for the subword-based downward closure is well understood, nothing is known in the case of priorities. We initiate the study of this problem and provide algorithms to compute priority downward closures for regular languages, one-counter languages, and context-free languages.

2012 ACM Subject Classification Theory of computation → Models of computation

Keywords and phrases downward closure, priority order, pushdown automata, non-deterministic finite automata, abstraction, computability

Digital Object Identifier 10.4230/LIPIcs.CONCUR.2023.39

Related Version Some formal proofs and details can be found in the full version of the paper.
Full Version: `https://arxiv.org/abs/2307.07460` [2]

Funding Funded by the European Union (ERC, FINABIS, 101077902). Views and opinions expressed are however those of the author(s) only and do not necessarily reflect those of the European Union or the European Research Council Executive Agency. Neither the European Union nor the granting authority can be held responsible for them.

Acknowledgements The authors are grateful to Yousef Shakiba for discussions on the block downward closure of regular languages.

1 Introduction

When analyzing infinite-state systems, it is often possible to replace individual components by an overapproximation based on (subword) downward closures. Here, the *(subword) downward closure* of a language $L \subseteq \Sigma^*$ is the set of all words that appear as (not necessarily contiguous) subwords of members of L. This overapproximation is usually possible because the verified properties are not changed when we allow additional behaviors resulting from subwords. Furthermore, this overapproximation simplifies the system because a well-known result by Haines is that for *every language* $L \subseteq \Sigma^*$, its subword downward closure is regular.

This idea has been successfully applied to many verification tasks, such as the verification of restricted lossy channel systems [1], concurrent programs with dynamic thread spawning and bounded context-switching [3, 7], asynchronous programs (safety, termination, liveness [22], but also context-free refinement verification [8]), the analysis of thread pools [9], and safety of parameterized asynchronous shared-memory systems [25]. For these reasons, there has been a substantial amount of interest in algorithms to compute finite automata for subword downward closures of given infinite-state sytems [4–6, 12–14, 17, 18, 26–30].

One situation where downward closures are useful is that of systems that send messages through a lossy channel, meaning that every message can be lost on the way. Then clearly, the downward closure of the set of sequences of messages is exactly the set of sequences observed by the receiver. This works as long as all messages can be dropped arbitrarily.

Priorities. However, if the messages are not dropped arbitrarily but as part of congestion control, then taking the set of all subwords would be too coarse an abstraction: Suppose we want to prioritize critical messages that can only be dropped if there are no lower-priority messages in the channel. For example, RFC 2475 describes an architecture that allows specifying relative priority among the IP packets from a finite set of priorities and allows the network links to drop lower priority packets to accommodate higher priority ones when the congestion in the network reaches a critical point [11]. As another example, in networks with an Asynchronous Transfer Mode layer, cells carry a priority in order to give preferences to audio or video packages over less time-critical packages [21]. In these situations, the subword downward closure would introduce behaviors that are not actually possible in the system.

To formally capture the effect of dropping messages by priorities, Haase, Schmitz and Schnoebelen [16] introduced *Priority Channel Systems (PCS)*. These feature an ordering on words (i.e. channel contents), called the *Prioritised Superseding Order (PSO)*, which allows the messages to have an assigned priority, such that higher priority messages can supersede lower priority ones. This order indeed allows the messages to be treated discriminatively, but the superseding is asymmetric. A message can be superseded only if there is a higher priority letter coming in the channel later. This means, PSO are the "priority counterpart" of the subword order for channels with priorities. In particular, in these systems, components can be abstracted by their *priority downward closure*, the downward closure with respect to the PSO. Fortunately, just as for subwords, priority downward closures are also always regular.

This raises the question of whether it is possible to compute finite automata for the priority downward closure for given infinite-state systems. For example, consider a recursive program that sends messages into a lossy channel with congestion control. Then, the set of possible message sequences that can arrive is exactly the priority downward closure $S{\downarrow_{\mathsf{P}}}$ of the language S of sent messages. Since S is context-free in this case, we would like to compute a finite automaton for $S{\downarrow_{\mathsf{P}}}$. While this problem is well-understood for subwords, nothing is known for priority downward closures.

Contribution. We initiate the study of computing priority downward closures. We show two main results. On the one hand, we study the setting above – computing priority downward closures of context-free languages. Here, we show that one can compute a doubly-exponential-sized automaton for its priority downward closure. On the other hand, we consider a natural restriction of context-free languages: We show that for one-counter automata, there is a polynomial-time algorithm to compute the priority downward closure.

Key technical ingredients. The first step is to consider a related order on words, which we call *block order*, which also has priorities assigned to letters, but imposes them more symmetrically. Moreover, we show that under mild assumptions, computing priority downward closures reduces to computing block downward closures.

Both our constructions – for one-counter automata and context-free languages – require new ideas. For one-counter automata, we modify the subword-based downward closures construction from [4] in a non-obvious way to block downward closures. Crucially, our modification relies on the insight that, in some word, repeating existing factors will always yield a word that is larger in the block order. For context-free languages, we present a novel inductive approach: We decompose the input language into finitely many languages with fewer priority levels and apply the construction recursively.

Outline of the paper. We fix notation in Section 2 and introduce the block order and show its relationship to the priority order in Section 3. In Sections 4–6, we then present methods for computing block and priority downward closures for regular languages, one-counter languages, and context-free languages, respectively.

2 Preliminaries

We will use the convention that $[i, j]$ denotes the set $\{i, i+1, \ldots, j\}$. By Σ, we represent a finite alphabet. Σ^* (Σ^+) denotes the set of (non-empty) words over Σ. When defining the priority order, we will equip Σ with a set of priorities with total order $(\mathcal{P}, \lessdot)$, i.e. there exists a fixed priority mapping from Σ to $\mathcal{P}$. The set of priority will be the set of integers $[0, d]$, with the canonical total order. By sets $\Sigma_{=p}$ $(p \in \mathcal{P})$, we denote the set of letters in Σ with priority p. For priority $p \in \mathcal{P}$, $\Sigma_{\leq p} = \Sigma_{=0} \cup \cdots \cup \Sigma_{=p}$, i.e. the set of letters smaller than or equal to p. For a word $w = a_0 a_1 \cdots a_k$, where $a_i \in \Sigma$, by $w[i, j]$, we denote the infix $a_i a_{i+1} \cdots a_{j-1} a_j$, and by $w[i]$, we denote a_i.

Finite automata and regular languages. A *non-deterministic finite state automaton (NFA)* is a tuple $\mathcal{A} = (Q, \Sigma, \delta, q_0, F)$, where Q is a finite set of *states*, Σ is its *input alphabet*, δ is its set of *edges* i.e. a finite subset of $Q \times \Sigma \cup \{\epsilon\} \times Q$, $q_0 \in Q$ is its *initial state*, and $F \subseteq Q$ is its set of *final states*. A word is accepted by $\mathcal{A}$ if it has a run from the initial state ending in a final state. The language *recognized* by an NFA $\mathcal{A}$ is called a regular language, and is denoted by $\mathcal{L}(\mathcal{A})$. The *size of a NFA*, denoted by $|\mathcal{A}|$, is the number of states in the NFA.

(Well-)quasi-orders. A *quasi-order*, denoted as $(X, \leq)$, is a set X with a reflexive and transitive relation $\leq$ on X. If $x \leq y$ (or equivalently, $y \geq x$), we say that x is smaller than y, or y is greater than x. If $\leq$ is also anti-symmetric, then it is called a *partial order*. If every pair of elements in X is comparable by $\leq$, then it is called a *total* or *linear* order. Let $(X, \leq_1)$ and $(Y, \leq_2)$ be two quasi orders, and $h : X \to Y$ be a function. We call h a *monomorphism* if it is one-to-one and $x_1 \leq_1 x_2 \iff h(x_1) \leq_2 h(x_2)$.

A quasi order $(X, \leq)$ is called a *well-quasi order (WQO)*, if any infinite sequence of elements $x_0, x_1, x_2, \ldots$ from X contains an increasing pair $x_i \leq x_j$ with $i < j$. If X is the set of words over some alphabet, then a WQO $(X, \leq)$ is called *multiplicative* if $\forall u, u', v, v' \in X$, $u \leq u'$ and $v \leq v'$ imply that $uv \leq u'v'$.

Subwords. For $u, v \in \Sigma^*$, we say $u \preccurlyeq v$, which we refer to as *subword order*, if u is a subword (not necessarily, contiguous) of v, i.e. if

$$\begin{aligned} u &= u_1 u_2 \cdots u_k \\ \text{and, } v &= v_0 u_1 v_1 u_2 v_2 \cdots v_{k-1} u_k v_k \end{aligned}$$

where $u_i \in \Sigma$ and $v_i \in \Sigma^*$. In simpler words, $u \preccurlyeq v$ if some letters of v can be dropped to obtain u. For example, let $\Sigma = [0, 1]$. Then, $0 \preccurlyeq 00 \preccurlyeq 010 \not\preccurlyeq 110$; 0 and 00 can be obtained by dropping letters from 00 and 010, respectively. But 010 cannot be obtained from 110, as the latter does not have sufficiently many 0s. If $u \preccurlyeq v$, we say that u is *subword smaller* than v, or simply that u is a *subword* of v. And we call a mapping from the positions in u to positions in v that witnesses $u \preccurlyeq v$ as the *witness position mapping*.

Since Σ is a WQO with the equality order, by Higman's lemma, Σ^* is a WQO with the subword order. It is in fact a multiplicative WQO: if $u \preccurlyeq u'$ and $v \preccurlyeq v'$, then dropping the same letters from $u'v'$ gives us uv.

Priority order. We take an alphabet Σ with priorities totally ordered by $\lessdot$. We say $u \preccurlyeq_{\mathsf{P}} v$, which we refer to as *priority order*, if $u = \epsilon$ or,

$$\begin{aligned} u &= u_1 u_2 \cdots u_k \\ \text{and, } v &= v_1 u_1 v_2 u_2 \cdots v_k u_k, \end{aligned}$$

such that $\forall i \in [1,k]$, $u_i \in \Sigma$ and $v_i \in \Sigma^*_{\leq u_i}$. It is easy to observe that the priority order is multiplicative, and is finer than the subword order, i.e. $\forall u, v \in \Sigma^*, u \preccurlyeq_{\mathsf{P}} v \implies u \preccurlyeq v$. As shown in [16, Theorem 3.6], the priority order on words over a finite alphabet with priorities is a well-quasi ordering:

▶ **Lemma 2.1.** *$(\Sigma^*, \preccurlyeq_{\mathsf{P}})$ is a WQO.*

Downward closure. We define the *subword downward closure* and *priority downward closure* for a language $L \subseteq \Sigma^*$ as follows:

$$L{\downarrow} := \{u \in \Sigma^* \mid \exists\, v \in L \colon u \preccurlyeq v\}, \qquad L{\downarrow}_{\mathsf{P}} := \{u \in \Sigma^* \mid \exists\, v \in L \colon u \preccurlyeq_{\mathsf{P}} v\}.$$

The following is the starting point for our investigation: It shows that for every language L, there exist finite automata for its downward closures w.r.t. $\preccurlyeq$ and $\preccurlyeq_{\mathsf{P}}$.

▶ **Lemma 2.2.** *Every subword downward closed sets and every priority downward closed set is regular.*

For the subword order, this was shown by Haines [19]. The same idea applies to the priority ordering: A downward closed set is the complement of an upward closed set. Therefore, and since every upward closed set in a well-quasi ordering has finitely many minimal elements, it suffices to show that the set of all words above a single word is a regular language. This, in turn, is shown using a simple automaton construction. In the full version, we prove an analogue of this for the block ordering (Lemma 3.5).

We stress that Lemma 2.2 is not effective: It does not guarantee that finite automata for downward closures can be computed for any given language. In fact, there are language classes for which they are not computable, such as reachability sets of *lossy channel systems* and *Church-Rosser languages* [15, 23]. Therefore, our focus will be on the question of how to effectively compute automata for priority downward closures.

3 The Block Order

We first define the block order formally and then give the intuition behind the definition. Let Σ be a finite alphabet, and $\mathcal{P} = [0,d]$ be a set of priorities with a total order $\lessdot$. Then for $u, v \in \Sigma^*$, where maximum priority occurring among u and v is p, we say $u \preccurlyeq_{\mathsf{B}} v$, if

i. if $u, v \in \Sigma^*_{=p}$, and $u \preccurlyeq v$, or

ii. if

$$\begin{aligned} u &= u_0 x_0 u_1 x_1 \cdots x_{n-1} u_n \\ \text{and, } v &= v_0 y_0 v_1 y_1 \cdots y_{m-1} v_m \end{aligned}$$

where $x_0, \dots x_{n-1}, y_0, \dots, y_{m-1} \in \Sigma_{=p}$, and for all $i \in [0,n]$, we have $u_i, v_i \in \Sigma^*_{\leq p-1}$ (the u_i and v_i are called *sub-p* blocks), and there exists a strictly monotonically increasing map $\phi : [0,n] \to [0,m]$, which we call the *witness block map*, such that

a. $u_i \preccurlyeq_{\mathsf{B}} v_{\phi(i)}$, $\forall i$,
b. $\phi(0) = 0$,
c. $\phi(n) = m$, and
d. $x_i \preccurlyeq v_{\phi(i)} y_{\phi(i)} v_{\phi(i)+1} \cdots v_{\phi(i+1)}$, $\forall i \in [0, n-1]$.

Intuitively, we say that u is *block smaller* than v, if either

- both words have letters of same priority, and u is a subword of v, or,
- the largest priority occurring in both words is p. Then we split both words along the priority p letters, to obtain sequences of sub-p blocks of words, which have words of strictly less priority. Then by item iia, we embed the sub-p blocks of u to those of v, such that they are recursively block smaller. Then with items iib and iic, we ensure that the first (and last) sub-p block of u is embedded in the first (resp., last) sub-p block of v. We will see later that this constraint allows the order to be multiplicative. Finally, by item iid, we ensure that the letters of priority p in u are preserved in v, i.e. every x_i indeed occurs between the embeddings of the sub-p block u_i and u_{i+1}.

▶ **Example 3.1.** Consider the alphabet $\Sigma = \{0^a, 0^b, 1^a, 1^b, 2^a, 2^b\}$ with priority set $\mathcal{P} = [0, 2]$ and $\Sigma_{=i} = \{i^a, i^b\}$. In the following examples, the color helps to identify the largest priority occurring in the words. First, notice that $\epsilon \preccurlyeq_{\mathsf{B}} 0^a \preccurlyeq_{\mathsf{B}} 0^a 0^b$, and hence

$$1^b 0^a \preccurlyeq_{\mathsf{B}} 0^a 1^b 0^a 0^a 1^a 0^a 0^b, \qquad \text{but} \qquad 1^b 0^a \not\preccurlyeq_{\mathsf{B}} 0^a 1^b 0^a 0^a 1^a 0^b 0^b.$$

This is because $0^a \not\preccurlyeq_{\mathsf{B}} 0^b 0^b$, i.e. the last sub-1 block of the former word cannot be mapped to the last sub-1 block of the latter word. As another example, we have

$$2^a 1^b 0^a \preccurlyeq_{\mathsf{B}} 0^a 2^a 0^a 1^b 0^a 0^a 1^a 0^a 0^b, \qquad \text{but} \qquad 2^a 1^b 0^a \not\preccurlyeq_{\mathsf{B}} 0^a 2^b 0^a 1^b 0^a 0^a 1^a 0^a 0^b.$$

This is because 2^a does not exist in the latter word, violating item iid. Finally, notice that

$$1^a 1^b \not\preccurlyeq_{\mathsf{B}} 1^a 2^a 1^b, \tag{1}$$

because the sub-2 block $1^a 1^b$ would have to be mapped to a single sub-2 block in the right-hand word; but none of them can accomodate $1^a 1^b$.

Note that by items iid and iia, we have that $u \preccurlyeq_{\mathsf{B}} v \implies u \preccurlyeq v$, for all $u, v \in \Sigma^*$. Then there exists a position mapping ρ from $[0, |u|]$ to $[0, |v|]$ such that $u[i] = v[\rho(i)]$, for all i. We say that a position mapping *respects block order* if for all i, $v[\rho(i), \rho(i+1)]$ contains letters of priorities smaller than $u[i]$ and $u[i+1]$. It is easy to observe that if $u \preccurlyeq_{\mathsf{B}} v$, then there exists a position mapping from u to v respecting the block order. The following is a straightforward repeated application of Higman's Lemma [20] (see the full version).

▶ **Theorem 3.2.** *$(\Sigma^*, \preccurlyeq_{\mathsf{B}})$ is a WQO.*

In fact, the block order is multiplicative, i.e. for all $u, v, u', v' \in \Sigma^*$ such that $u \preccurlyeq_{\mathsf{B}} u'$ and $v \preccurlyeq_{\mathsf{B}} v'$, it holds that $uv \preccurlyeq_{\mathsf{B}} u'v'$.

▶ **Lemma 3.3.** *$(\Sigma^*, \preccurlyeq_{\mathsf{B}})$ is a multiplicative WQO.*

Proof. For singleton $\mathcal{P}$, the result trivially holds because it coincides with the subword order. Let $(\Sigma^*_{\le p-1}, \preccurlyeq_{\mathsf{B}})$ be multiplicative. Now we show that $(\Sigma^*_{\le p}, \preccurlyeq_{\mathsf{B}})$ is multiplicative. To this end, let $u \preccurlyeq_{\mathsf{B}} u'$, $v \preccurlyeq_{\mathsf{B}} v'$, and ϕ, ψ be the witnessing block maps respectively. We assume

$$\begin{aligned}
u &= u_0 x_0 u_1 x_1 u_2 x_2 \cdots x_{k-1} u_k \\
v &= v_0 y_0 v_1 y_1 v_2 y_2 \cdots y_{l-1} v_l \\
u' &= u'_0 x'_0 u'_1 x'_1 u'_2 x'_2 \cdots x'_{k-1} u'_{k'} \\
v' &= v'_0 y'_0 v'_1 y'_1 v'_2 y'_2 \cdots y'_{l-1} v'_{l'}
\end{aligned}$$

where $x_i, y_i, x'_i, y'_i \in \Sigma_{=p}$. Consider the function $\delta\colon [0, k+l-1] \to [0, k'+l'-1]$ with

$$i \mapsto \begin{cases} \phi(i), & \text{if } 1 \le i \le k \\ \psi(i-k+1), & \text{if } k < i \le k+l-1 \end{cases}$$

Since the k^{th} sub-p block of u and the 1^{st} sub-p block of v combines in uv to form one sub-p block, we have $k+l-1$ sub-p blocks. Similarly, $u'v'$ has $k'+l'-1$ sub-p blocks. And hence $u_k v_1 \preccurlyeq_{\mathsf{B}} u'_{k'} v'_1$, by induction hypothesis. The recursive embedding is obvious for other sub-p blocks. We also have that $\delta(0) = 0$ and $\delta(k+l-1) = k'+l'-1$. By monotonicity of ϕ and ψ, δ is also strictly monotonically increasing. Hence, δ witnesses $uv \preccurlyeq_{\mathsf{B}} u'v'$. ◀

Pumping. In the subword ordering, an often applied property is that for any words u, v, w, we have $uw \preccurlyeq uvw$, i.e. inserting any word leads to a superword. This is not true for the block ordering, as we saw in Example 3.1, (1). However, one of our key observations about the block order is the following property: If the word we insert is just a repetition of an existing factor, then this yields a larger word in the block ordering. This will be crucial for our downward closure construction for one-counter automata in Section 5.

▶ **Lemma 3.4** (Pumping Lemma). *For any $u, v, w \in \Sigma^*$, we have $uvw \preccurlyeq_{\mathsf{B}} uvvw$.*

Before we prove Lemma 3.4, let us note that by applying Lemma 3.4 multiple times, this implies that we can also repeat multiple factors. For instance, if $w = w_1 w_2 w_3 w_4 w_5$, then $w \preccurlyeq_{\mathsf{B}} w_1 w_2^2 w_3 w_4^3 w_5$. Figure 1 shows an example on how to choose the witness block map.

Proof. We proceed by induction on the number of priorities. If there is just a single priority (i.e. $\mathcal{P} = \{0\}$), then $\preccurlyeq_{\mathsf{B}}$ coincides with $\preccurlyeq$ and the statement is trivial. Let us assume the lemma is established for words with up to n priorities. We distinguish two cases.

- Suppose v contains only letters of priorities $[0, n]$. Then repeating v means repeating a factor inside a sub-$(n+1)$ block, which is a word with priorities in $[0, n]$. Hence, the statement follows by induction: Formally, this means we can use the embedding mapping that sends block i of uvw to block i of $uvvw$.
- Suppose v contains a letter of priority $n+1$. write $v = v_0 x_1 v_1 \cdots x_m v_m$, where $x_1, \ldots, x_m$ are the letters of priority $n+1$ in v and $v_0, \ldots, v_m$ are the sub-$(n+1)$ blocks of v. Then:

$$uvw = uv_0x_1v_1 \cdots x_m v_m w, \qquad uvvw = uv_0x_1v_1 \cdots x_m \underbrace{v_m v_0 x_1 v_1 \cdots x_m}_{\text{skipped}} v_m w$$

  The idea is simple: Our witness block map just skips the m sub-$(n+1)$ blocks inside of $v_m v_0 x_1 \cdots v_{m-1} x_m$. Thus, the sub-$(n+1)$ blocks in $uv_0 x_1 \cdots v_{m-1} x_m$ are mapped to the same blocks in $uv_0 x_1 \cdots v_{m-1} x_m$, and the sub-$(n+1)$ blocks in $v_m w$ are mapped to the same blocks in $v_m w$. This is clearly a valid witness block map, since the first (resp. last) sub-$(n+1)$ block is mapped to the first (resp. last), and each sub-$(n+1)$ block is mapped to an identical sub-$(n+1)$ block. ◀

Regular downward closures. As for $\preccurlyeq$ and $\preccurlyeq_{\mathsf{P}}$, we define $L{\downarrow_{\mathsf{B}}} = \{u \in \Sigma^* \mid \exists v \in L\colon u \preccurlyeq_{\mathsf{B}} v\}$ for any $L \subseteq \Sigma^*$.

▶ **Lemma 3.5.** *For every $L \subseteq \Sigma^*$, $L{\downarrow_{\mathsf{B}}}$ is a regular language.*

For the proof of Lemma 3.5, one can argue as mentioned above: The complement $\Sigma^* \setminus (L{\downarrow_{\mathsf{B}}})$ of $L{\downarrow_{\mathsf{B}}}$ is upward closed. And since $\preccurlyeq_{\mathsf{B}}$ is a WQO, $\Sigma^* \setminus (L{\downarrow_{\mathsf{B}}})$ has finitely many minimal elements. It thus remains to show that for each word $w \in \Sigma^*$, the set of words $\preccurlyeq_{\mathsf{B}}$-larger than w is regular, which is a simple exercise. Details can be found in the full version.

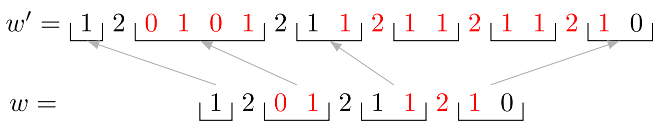

Figure 1 Here $\Sigma = [0,2]$, $\mathcal{P} = [0,2]$, and $A_i = \{i\}$, $w = 12(01)21(121)0$ and $w' = 12(01)^2 21(121)^3 0$. The repeated segments are marked in red, and the arrows denote the witness block map.

Block order vs. priority order. We will later see (Theorem 4.4) that under mild conditions, computing priority downward closures reduces to computing block downward closures. The following lemma is the main technical ingredient in this: It shows that the block order refines the priority order on words that end in the same letter, assuming the alphabet has a certain shape. A priority alphabet $(\Sigma, \mathcal{P})$ with $\mathcal{P} = [1,d]$ is called *flat* if $|\Sigma_{=i}| = 1$ for each $i \in [1,d]$.

▶ **Lemma 3.6.** *If Σ is flat and $u, v \in \Sigma^* a$ for some $a \in \Sigma$, then $u \preccurlyeq_{\mathsf{B}} v$ implies $u \preccurlyeq_{\mathsf{P}} v$.*

Proof. Since $u \preccurlyeq_{\mathsf{B}} v$, there exists a witness position mapping ρ that maps the positions of the letters in u to that of v, such that it respects the block order, and it maps the last position of u to the last of v.

Let $u = u_0 u_1 \cdots u_k$. We say that a position mapping violates the priority order at position i (for $i \in [0, k-1]$), if $v[\rho(i)+1, \rho(i+1)]$ has a letter of priority higher than that of $u[i+1]$. Note that if ρ does not violate the priority order at any position, then $u \preccurlyeq_{\mathsf{P}} v$.

Let i be the largest position at which ρ violates the priority order, i.e. $v[\rho(i)+1, \rho(i+1)]$ has a letter of priority higher than that of $u[i+1]$. We show that if ρ respects the block order till position i, there exists another witness position mapping ρ' that respects the block order till position $i-1$, and has one few position of violation (i.e. no violation at position i).

We first observe that $u[i] > u[i+1]$, which holds since ρ respects the block order till position i, implying that $v[\rho(i)+1, \rho(i+1)]$ does not have a letter of priority higher than $min\{u[i], u[i+1]\}$, and if $u[i] \leq u[i+1]$, ρ does not violate the priority order at i.

Then observe that $v[\rho(i)+1, \rho(i+1)]$ does not have a letter with priority p, where $u[i] > p > u[i+1]$, otherwise the sub-$u[i]$ block of u immediately after $u[i]$, can not be embedded to that of v immediately after $v[\rho(i)]$, since it would have to be split along p, and the first sub-p block in v will not be mapped to any in u. Then $v[\rho(i)+1, \rho(i+1)]$ has letter of priority $u[i]$ (for a violation at i). Then consider the mapping ρ' that maps i to the last $u[i]$ letter in $v[\rho(i)+1, \rho(i+1)]$ (say at $v[j]$ for some j, $\rho(i)+1 \leq j \leq \rho(i+1)$).

This mapping respects the block order till position $i-1$, trivially, as we do not change the mapping before i. We show that there is no priority order violation at position i. This holds because the only larger priority letter occurring in $v[\rho(i)+1, \rho(i+1)]$ was $u[i]$, and due to the definition of ρ', $v[\rho'(i)+1, \rho'(i+1)]$ has no letter of priority higher than $u[i+1]$. Since we do not change the mapping after position i, ρ' does not introduce a violation at any position after i. Hence we have a new position mapping that has one few position of priority order violation. ◀

▶ Remark 3.7. We want to stress that the flatness assumption in Lemma 3.6 is crucial: Consider the alphabet Σ from the Example 3.1. Then $1^a 0^a \preccurlyeq_{\mathsf{B}} 1^a 1^b 0^a$, but $1^a 0^a \not\preccurlyeq_{\mathsf{P}} 1^a 1^b 0^a$. Here only one position mapping exists, and it is not possible to remap 1^a to 1^b since they are two distinct letters of same priority. Hence, we need to assume that each priority greater than zero has at most one letter.

4 Regular Languages

In this section, we show how to construct an NFA for the block downward closure of a regular language. To this end, we show that both orders are rational transductions.

Rational transductions. A *finite state transducer* is a tuple $\mathcal{A} = (Q, X, Y, E, q_0, F)$, where Q is a finite set of states, X and Y are *input* and *output alphabets*, respectively, E is the set of *edges* i.e. finite subset of $Q \times X^* \times Y^* \times Q$, $q_0 \in Q$ is the *initial state*, and $F \subseteq Q$ is the set of *final states*. A *configuration* of $\mathcal{A}$ is a triple $(q, u, v) \in Q \times X^* \times Y^*$. We write $(q, u, v) \to_{\mathcal{A}} (q', u', v')$, if there is an edge (q, x, y, q') with $u' = ux$ and $v' = vy$. If there is an edge (q, x, y, q'), we sometimes denote this fact by $q \xrightarrow{(x,y)}_{\mathcal{A}} q'$, and say "read x at q, output y, and goto q'". The *size of a transducer*, denoted by $|\mathcal{A}|$, is the number of its states.

A *transduction* is a subset of $X^* \times Y^*$ for some finite alphabets X, Y. The *transduction defined by* $\mathcal{A}$ is $\mathcal{T}(\mathcal{A}) = \{(u, v) \in X^* \times Y^* \mid (q_0, \epsilon, \epsilon) \to^*_{\mathcal{A}} (f, u, v) \text{ for some } f \in F\}$. A transduction is called *rational* if it is defined by some finite-state transducer. Sometimes we abuse the notation and output a regular language $R \subseteq Y^*$ on an edge, instead of a letter. It should be noted that this abuse is equivalent to original definition of finite state transducers.

We say that a language class $\mathcal{C}$ is *closed under rational transductions* if for each language $L \in \mathcal{C}$, and each rational transduction $R \subseteq X^* \times Y^*$, *the language obtained by applying the transduction* R to L, $RL \overset{def}{=} \{v \in Y^* \mid (u, v) \in R \text{ for some } u \in L\}$ also belongs to $\mathcal{C}$. We call such language classes *full trio*. Regular languages, context-free languages, recursively enumerable languages are some examples of full trios [10].

Transducers for orders. It is well-known that the subword order is a rational transduction, i.e. the relation $T = \{(u, v) \in X^* \times X^* \mid v \preccurlyeq u\}$ is defined by a finite-state transducer. For example, it can be defined by a one-state transducer that can non-deterministically decide to output or drop each letter. Note that on applying the transduction to any language, it gives the subword downward closure of the language. This means, for every $L \subseteq X^*$, we have $TL = L\!\downarrow$. We will now describe analogous transducers for the priority and block order.

▶ **Theorem 4.1.** *Given a priority alphabet with priorities* $[0, k]$*, one can construct in polynomial time a transducer for* $\preccurlyeq_{\mathsf{B}}$ *and a transducer for* $\preccurlyeq_{\mathsf{P}}$*, each of size* $\mathcal{O}(k)$*.*

Proof. The transducers for the block and priority order are similar. Intuitively, both remember the maximum of the priorities dropped or to be dropped, and keep or drop the coming letters accordingly. We show the transducer for the priority order here since it is applied in Theorem 4.4. The transducer for the block order is detailed in the full version.

Let Σ be a finite alphabet, with priorities $\mathcal{P} = [0, k]$. Consider the transducer that has one state for every priority, a non-final sink state, and a distinguished final state. If the transducer is in the state for priority r and reads a letter a of priority s, then

- if $s < r$, then it outputs nothing and stays in state r,
- if $s \geq r$, then it can output nothing, and go to state s,
- if $s \geq r$, it can also output a, and go to state 0, or the accepting state non-deterministically,
- for any other scenario, goes to the sink state.

The priority 0 state is the initial state. Intuitively, the transducer remembers the largest priority letter that has been dropped, and keeps only a letter of higher priority later. To be accepting, it has to read the last letter to go to the accepting final state. ◀

The following theorem states that the class of regular languages form a full trio.

▶ **Theorem 4.2** ([24, Corollary 3.5.5])**.** *Given an NFA $\mathcal{A}$ and a transducer $\mathcal{B}$, we can construct in polynomial time an NFA of size $|\mathcal{A}| \cdot |\mathcal{B}|$ for $\mathcal{T}(\mathcal{B})(\mathcal{L}(\mathcal{A}))$.*

Theorems 4.1 and 4.2 give us a polynomial size NFA recognizing the priority and block downward closure of a regular language, which is computable in polynomial time as well.

▶ **Theorem 4.3.** *Priority and block downward closures for regular languages are effectively computable in time polynomial in the number of states in the NFA recognizing the language.*

Theorem 4.3 and Lemma 3.6 now allow us to reduce the priority downward closure computability to computability for block order.

▶ **Theorem 4.4.** *If $\mathcal{C}$ is a full trio and we can effectively compute block downward closures for $\mathcal{C}$, then we can effectively compute priority downward closures.*

Proof. The key idea is to reduce priority downward closure computation to the setting where (i) all words end in the same letter and (ii) the alphabet is flat. Since by Lemma 3.6, on those languages, the block order is finer than the priority order, computing the block order will essentially be sufficient.

Let us first establish (i). Let $L \in \mathcal{C}$. Then for each $a \in \Sigma$, the language $L_a = L \cap \Sigma^* a$ belongs to $\mathcal{C}$. Since $L = \bigcup_{a\in\Sigma} L_a \cup E$ and thus $L{\downarrow_{\mathsf{P}}} = \bigcup_{a\in\Sigma} L_a{\downarrow_{\mathsf{P}}} \cup E$, it suffices to compute priority downward closures for each L_a, where $E = \{\epsilon\}$ if $\epsilon \in L$, else $\emptyset$. This means, it suffices to compute priority downward closures for languages where all words end in the same letter.

To achieve (ii), we make the alphabet flat. We say that $(\Sigma, \mathcal{P}')$ is the *flattening* of $(\Sigma, \mathcal{P} = [0, d])$, if $\mathcal{P}'$ is obtained by choosing a total order to Σ such that if a has smaller priority than b in $(\Sigma, \mathcal{P})$, then a has smaller priority than b in $(\Sigma, \mathcal{P}')$. (In other words, we pick an arbitrary linearization of the quasi-order on Σ that expresses "has smaller priority than"). Then, we assign priorities based on this total ordering. Let $\preccurlyeq_{\mathsf{B}}^{\mathsf{flat}}$ and $\preccurlyeq_{\mathsf{P}}^{\mathsf{flat}}$ denote the block order and priority order, resp., based on the flat priority assignment. It is a simple observation that for $u, v \in \Sigma^*$, we have that $u \preccurlyeq_{\mathsf{P}}^{\mathsf{flat}} v$ implies $u \preccurlyeq_{\mathsf{P}} v$.

Now observe that for $u, v \in L_a$, Lemma 3.6 tells us that $u \preccurlyeq_{\mathsf{B}}^{\mathsf{flat}} v$ implies $u \preccurlyeq_{\mathsf{P}}^{\mathsf{flat}} v$ and therefore also $u \preccurlyeq_{\mathsf{P}} v$. This implies that $(L_a{\downarrow_{\mathsf{B}}^{\mathsf{flat}}}){\downarrow_{\mathsf{P}}} = L_a{\downarrow_{\mathsf{P}}}$. By assumption, we can compute a finite automaton $\mathcal{A}$ with $\mathcal{L}(\mathcal{A}) = L_a{\downarrow_{\mathsf{B}}^{\mathsf{flat}}}$. Since then $\mathcal{L}(\mathcal{A}){\downarrow_{\mathsf{P}}} = (L_a{\downarrow_{\mathsf{B}}^{\mathsf{flat}}}){\downarrow_{\mathsf{P}}} = L_a{\downarrow_{\mathsf{P}}}$, we can compute $L_a{\downarrow_{\mathsf{P}}}$ by applying Theorem 4.3 to $\mathcal{A}$ to compute $\mathcal{L}(\mathcal{A}){\downarrow_{\mathsf{P}}} = L_a{\downarrow_{\mathsf{P}}}$. ◀

5 One-counter Languages

In this section, we show that for the class of languages accepted by one-counter automata, which form a full-trio [10, Theorem 4.4], the block and priority downward closures can be computed in polynomial time. We prove the following theorem.

▶ **Theorem 5.1.** *Given an OCA $\mathcal{A}$, $\mathcal{L}(\mathcal{A}){\downarrow_{\mathsf{B}}}$ and $\mathcal{L}(\mathcal{A}){\downarrow_{\mathsf{P}}}$ are computable in polynomial time.*

Here, the difficulty is that existing downward closure constructions exploit that inserting any letters in a word yields a super-word. However, for the block order, this might not be true: Introducing high-priority letters might split a block unintentionally. However, we observe that the subword closure construction from [4] can be modified so that when constructing larger runs (to show that our NFA only accepts words in the downward closure), we only repeat existing factors. Lemma 3.4 then yields that the resulting word is block-larger.

According to Theorem 4.4, it suffices to show that block downward closures are computable in polynomial time (an inspection of the proof of Theorem 4.4 shows that computing the priority downward closure only incurs a polynomial overhead).

One-counter automata. One-counter automata are finite state automata with a counter that can be incremented, decremented, or tested for zero. Formally, a *one-counter automaton (OCA)* $\mathcal{A}$ is a 5-tuple $(Q, \Sigma, \delta, q_0, F)$ where Q is a finite set of states, $q_0 \in Q$ is an initial state, $F \subseteq Q$ is a set of final states, Σ is a finite alphabet and $\delta \subseteq Q \times (\Sigma \cup \{\epsilon\}) \times \{-1, 0, +1, z\} \times Q$ is a set of transitions. Transitions $(p_1, a, s, p_2) \in \delta$ are classified as *incrementing* $(s = +1)$, *decrementing* $(s = -1)$, *internal* $(s = 0)$, or *test for zero*$(s = z)$.

A *configuration* of an OCA is a pair that consists of a state and a (non-negative) counter value, i.e., $(q, n) \in Q \times \mathbb{N}$. A sequence $\pi = (p_0, c_0), t_1, (p_1, c_1), t_2, \cdots, t_m, (p_m, c_m)$ where $(p_i, c_i) \in Q \times \mathbb{Z}$, $t_i \in \delta$ and $(p_{i-1}, c_{i-1}) \xrightarrow{t_i} (p_i, c_i)$ is called:

- a *quasi-run*, denoted $\pi = (p_0, c_0) \overset{w}{\underset{\mathcal{A}}{\Longrightarrow}} (p_m, c_m)$, if none of t_i is a test for zero;
- a *run*, denoted $\pi = (p_0, c_0) \xrightarrow{w}_{\mathcal{A}} (p_m, c_m)$, if all $(p_i, c_i) \in Q \times \mathbb{N}$.

For any quasi-run π as above, the sequence of transitions $t_1, \cdots, t_m$ is called a *walk* from the state p_0 to the state p_m. A run $(p_0, c_0) \xrightarrow{w} (p_m, c_m)$ is called *accepting* in $\mathcal{A}$ if $(p_0, c_0) = (q_0, 0)$ where q_0 is the initial state of $\mathcal{A}$ and p_m is a final state of $\mathcal{A}$, i.e. $p_m \in F$. In such a case, the word w is *accepted* by $\mathcal{A}$.

Simple one-counter automata. As we will show later, computing block downward closures of OCA easily reduces to the case of simple OCA. A *simple OCA (SOCA)* is defined analogously to OCA, with the differences that (i) there are no zero tests, (ii) there is only one final state, (iii) for acceptance, the final counter value must be zero.

We first show that the block downward closures can be effectively computed for the simple one-counter automata languages.

▶ **Proposition 5.2.** *Given a simple OCA $\mathcal{A}$, we can compute $\mathcal{L}(\mathcal{A})\downarrow_{\mathsf{B}}$ in polynomial time.*

We present a rough sketch of the construction, full details can be found in the full version. The starting point of the construction is the one for subwords in [4], but the latter needs to be modified in a non-obvious way using Lemma 3.4.

Let $\mathcal{A} = (Q, \Sigma, \delta, q_0, q_f)$ be a simple OCA, with $|Q| = K$. We construct an NFA $\mathcal{B}$ that can simulate $\mathcal{A}$ in three different modes. In the first mode, it simulates $\mathcal{A}$ until the counter value reaches K, and when the value reaches $K + 1$, it switches to the second mode. The second mode simulates $\mathcal{A}$ while the counter value stays below $K^2 + K + 1$. Moreover, and this is where our construction differs from [4]: if $\mathcal{B}$ is in the second mode simulating $\mathcal{A}$ in some state q, then $\mathcal{B}$ can spontaneously execute a loop from q to q of $\mathcal{A}$ while ignoring its counter updates. When the counter value in the second mode drops to K again, $\mathcal{B}$ non-deterministically switches to the third mode to simulate $\mathcal{A}$ while the counter value stays below K. Thus, $\mathcal{B}$ only needs to track counter values in $[0, K^2 + K + 1]$, meaning they can be stored in its state. We claim that then $\mathcal{L}(\mathcal{A}) \subseteq \mathcal{L}(\mathcal{B}) \subseteq \mathcal{L}(\mathcal{A})\downarrow_{\mathsf{B}}$.

▶ **Lemma 5.3.** $\mathcal{L}(\mathcal{A}) \subseteq \mathcal{L}(\mathcal{B})$.

If a word in $\mathcal{L}(\mathcal{A})$ has a run with counters bounded by $K^2 + K + 1$, then it trivially belongs to $\mathcal{L}(\mathcal{B})$. If the counters go beyond $K^2 + K + 1$, then with the classical "unpumping" argument, one can extract two loops, one increasing the counter, one decreasing it. These loops can then be simulated by the spontaneous loops in the second mode of $\mathcal{B}$.

The more interesting inclusion is the following:

▶ **Lemma 5.4.** $\mathcal{L}(\mathcal{B}) \subseteq \mathcal{L}(\mathcal{A})\downarrow_{\mathsf{B}}$.

We have to show that each spontaneous loop in $\mathcal{B}$ can be justified by padding the run with further loop executions so as to obtain a run of $\mathcal{A}$. This is possible because to execute such a spontaneous loop, we must have gone beyond K and later go to zero again. Thus, there exists a “pumping up” loop adding, say $k \geq 0$ to the counter, and a “pumping down” loop, subtracting, say $\ell \geq 0$ from the counter. We can therefore repeat all spontaneous loops so often that their effect – when seen as transitions in $\mathcal{A}$ – is a (positive or negative) multiple M of $k \cdot \ell$. Then, we execute the k- and the ℓ-loop so often so as to get the counter values so high that (i) our repeated spontaneous loops never cross zero and (ii) the effect difference of the new loops is exactly M. Since in our construction (in contrast to [4]), the padding only *repeated words that already exist* in the run of $\mathcal{B}$, Lemma 3.4 implies that the word of $\mathcal{B}$ embeds via the block order.

General OCA. Let us now show how to construct the block downward closure of general OCAs. Suppose we are given an OCA $\mathcal{A}$. For any two states p, q, consider the simple OCA $\mathcal{A}_{p,q}$ obtained from $\mathcal{A}$ by removing all zero tests, making p initial, and q final. Then $\mathcal{L}(\mathcal{A})$ is the set of words read from $(p, 0)$ to $(q, 0)$ without using zero tests. We now compute for each p, q a finite automaton $\mathcal{B}_{p,q}$ for the block downward closure of $\mathcal{A}_{p,q}$. Clearly, we may assume that $\mathcal{B}_{p,q}$ has exactly one initial state and one final state. Finally, we obtain the finite automaton $\mathcal{B}$ from $\mathcal{A}$ as follows: We remove all transitions *except* the zero tests. Each zero test from p to q is replaced with an edge $p \xrightarrow{\varepsilon} q$. Moreover, for any states p and q coming from $\mathcal{A}$, we glue in the automaton $\mathcal{B}_{p,q}$ (by connecting p with $\mathcal{B}_{p,q}$'s initial state and connecting $\mathcal{B}_{p,q}$'s final state with q). Then, since the block order is multiplicative, we have that $L(\mathcal{B})$ accepts exactly the block downward closure of $\mathcal{A}$.

Futhermore, note that since our construction for simple OCA is polynomial, the general case is as well: The latter employs the former to $|Q|^2$ simple OCAs.

6 Context-free Languages

The key trick in our construction for OCA was that we could modify the subword construction so that the overapproximating NFA $\mathcal{B}$ has the property that in any word from $\mathcal{L}(\mathcal{B})$, we can repeat factors to obtain a word from $\mathcal{A}$. This was possible because in an OCA, essentially any pair of loops – one incrementing, one decrementing – could be repeated to pad a run.

However, in context-free languages, the situation is more complicated. With a stack, any pumping must always ensure that stack contents match: It is not possible to compensate stack effects with just two loops. In terms of grammars, the core idea for subword closures of context-free languages L is usually to overapproximate “pump-like” derivations $X \overset{*}{\Rightarrow} uXv$ by observing that – up to subwords – they can generate any $u'Xv'$ where the letters of u' can occur on the left and the letters of v' can occur on the right in derivations $X \overset{*}{\Rightarrow} \cdot X \cdot$. Showing that all such words belong to the downward closure leads to derivations $X \overset{*}{\Rightarrow} u''\bar{v}Xv''\bar{u}$, where u'', v'' are super-words of u', v' such that $X \overset{*}{\Rightarrow} u''X\bar{u}$ and $X \overset{*}{\Rightarrow} \bar{v}Xv''$ can be derived. The additional infixes could introduce high priority letters and thus split blocks unintentionally.

Therefore, we provide a novel recursive approach to compute the block downward closure by decomposing derivations at high-priority letters. This is non-trivial as this decomposition might not match the decomposition given by derivation trees. Formally, we show:

▶ **Theorem 6.1.** *Given a context-free language $L \subseteq \Sigma_{\leq n}^*$, one can construct a doubly-exponential-sized automaton for $L{\downarrow}_{\mathsf{B}}$, and thus also for $L{\downarrow}_{\mathsf{P}}$.*

We do not know if this doubly exponential upper bound is optimal. A singly-exponential lower bound follows from the subword case: It is known that subword downward closures of context-free languages can require exponentially many states [6]. However, it is not clear whether for priority or block downward closures, there is a singly-exponential construction.

We again note that Theorem 4.4 (and its proof) imply that for Theorem 6.1, it suffices to compute a finite automaton for the block downward closure of the context-free language: Computing the priority downward closure then only increases the size polynomially.

Grammars. We present the construction using *context-free grammars*, which are tuples $\mathcal{G} = (N, T, P, S)$, where N is a finite set of *non-terminal letters*, T is a finite set of *terminal letters*, P is a finite set of *productions* of the form $X \to w$ with $X \in N$ and $w \in (N \cup T)^*$, and S is the *start symbol.* For $u, v \in (N \cup T)^*$, we have $u \Rightarrow v$ if there is a production $X \to w$ in P and $x, y \in (N \cup T)^*$ with $u = xXy$ and $v = xwy$. The *language generated by* $\mathcal{G}$, is then $\mathcal{L}(\mathcal{G}) := \{w \in T^* \mid S \overset{*}{\Rightarrow} w\}$, where $\overset{*}{\Rightarrow}$ is the reflexive, transitive closure of $\Rightarrow$.

Assumption on the alphabet. In order to compute block downward closures, it suffices to do this for flat alphabets (see Section 3). The argument is essentially the same as in Theorem 4.4: By flattening the alphabet as in the proof of Theorem 4.4, we obtain a finer block order, so that first computing an automaton for the flat alphabet and then applying Theorem 4.3 to the resulting finite automaton will yield a finite automaton for the original (non-flat) alphabet. In the following, we will assume that the input grammar $\mathcal{G}$ is in Chomsky normal form, meaning every production is of the form $X \to YZ$ for non-terminals X, Y, Z, or of the form $X \to a$ for a non-terminal X and a terminal a.

Kleene grammars. Suppose we are given a context-free grammar $\mathcal{G} = (N, \Sigma, P, S)$. Roughly speaking, the idea is to construct another grammar $\mathcal{G}'$ whose language has the same block downward closure as $\mathcal{L}(\mathcal{G})$, but with the additional property that every word can be generated using a derivation tree that is *acyclic*, meaning that each path contains every non-terminal at most once. Of course, if this were literally true, $\mathcal{G}'$ would generate a finite language. Therefore, we allow a slightly expanded syntax: We allow Kleene stars in context-free productions.

This means, we allow right-hand sides to contain occurrences of B^*, where B is a non-terminal. The semantics is the obvious one: When applying such a rule, then instead of inserting B^*, we can generate any B^k with $k \geq 0$. We call grammars with such productions *Kleene grammar.* A *derivation tree* in a Kleene grammar is defined as for context-free grammars, aside from the expected modification: If some B^* occurs on a right-hand side, then we allow any (finite) number of B-labeled children in the respective place. Then indeed, a Kleene grammar can generate infinite sets using acyclic derivation trees. Given a Kleene grammar $\mathcal{H}$, let $\mathsf{acyclic}(\mathcal{H})$ be the set of words generated by $\mathcal{H}$ using acyclic derivation trees.

▶ **Lemma 6.2.** *Given a Kleene grammar $\mathcal{H}$, one can construct an exponential-sized finite automaton accepting* $\mathsf{acyclic}(\mathcal{H})$*.*

Proof sketch. The automaton simulates a (say, preorder) traversal of an acyclic derivation tree of $\mathcal{H}$. This means, its state holds the path to the currently visited node in the derivation tree. Since every path has length at most $|N|$, where N is the set of non-terminals of $\mathcal{H}$, the automaton has at most exponentially many states. ◀

Given Lemma 6.2, for Theorem 6.1, it suffices to construct a Kleene grammar $\mathcal{G}'$ of exponential size such that $\mathsf{acyclic}(\mathcal{G}')\downarrow_{\mathsf{B}} = \mathcal{L}(\mathcal{G})\downarrow_{\mathsf{B}}$.

Normal form and grammar size. We will ensure that in the constructed grammars, the productions are of the form (i) $X \to w$, where w is a word of length ≤ 3 and consisting of non-terminals Y or Kleene stars Y^* or (ii) $X \to a$ where a is a terminal. This means, the total size of the grammar is always polynomial in the number of non-terminals. Therefore, to analyze the complexity, it will suffice to measure the number of non-terminals.

Highest occurring priorities. Similar to classical downward closure constructions for context-free languages, we want to overapproximate the set of words generated by "pump derivations" of the form $X \overset{*}{\Rightarrow} uXv$. Since we are dealing with priorities, we first partition the set of such derivations according to the highest occurring priorities, on the left and on the right. Thus, for $r, s \in [0, p]$, we will consider all derivations $X \overset{*}{\Rightarrow} uXv$ where r is the highest occurring priority in u and s is the highest occurring priority in v. To ease notation, we define $\Sigma_{\max r}$ to be the set of words in $\Sigma^*_{\leq r}$ in which r is the highest occurring priority. Since $\Sigma_{\max r} = \Sigma^+_{\max r}$, we will write $\Sigma^+_{\max r}$ to remind us that this is not an alphabet. Notice that for $r \in [1, p]$, we have $\Sigma^+_{\max r} = \Sigma^*_{\leq r} r \Sigma^*_{\leq r}$ and $\Sigma^+_{\max 0} = \Sigma^*_{\leq 0}$.

Language of ends. In order to perform an inductive construction, we need a way to transform pairs $(u, v) \in \Sigma^+_{\max r} \times \Sigma^+_{\max s}$ into words over an alphabet with fewer priorities. Part of this will be achieved by the *end maps* $\overleftarrow{\tau}_r(\cdot)$ and $\overrightarrow{\tau}_s(\cdot)$ as follows. Let $\hat{\Sigma}$ be the priority alphabet obtained from Σ by adding the letters $\#$, $\overleftarrow{\#}$, and $\overrightarrow{\#}$ as letters with priority zero. Now for $r \in [1, p]$, the function $\overleftarrow{\tau}_r \colon \Sigma^+_{\max r} \to \hat{\Sigma}^*_{\leq r-1}$ is defined as:

$$\overleftarrow{\tau}_r(w) = u\overleftarrow{\#}v, \text{ where } w = urx_1r \cdots x_nrv \text{ for some } n \geq 0,\ u, v, x_1, \ldots, x_n \in \Sigma^*_{\leq r-1}.$$

Thus, $\overleftarrow{\tau}_r(w)$ is obtained from w by replacing the largest possible infix surrounded by r with $\overleftarrow{\#}$. For $r = 0$, it will be convenient to have the constant function $\overleftarrow{\tau}_0 \colon \Sigma^+_{\max 0} \to \{\overleftarrow{\#}\}$. Analogously, we define for $s \in [1, p]$ the function $\overrightarrow{\tau}_s \colon \Sigma^+_{\max s} \to \hat{\Sigma}^*_{\leq s-1}$ by

$$\overrightarrow{\tau}_s(w) = u\overrightarrow{\#}v, \text{ where } w = usx_1s \cdots x_nsv \text{ for some } n \geq 0,\ u, v, x_1, \ldots, x_n \in \Sigma^*_{\leq s-1}.$$

Moreover, we also set $\overrightarrow{\tau}_0 \colon \Sigma^+_{\max 0} \to \{\overrightarrow{\#}\}$ to be the constant function yielding $\overrightarrow{\#}$.

In particular, for $r, s \in [1, p]$, we have $\overleftarrow{\tau}_r(w), \overrightarrow{\tau}_s(w) \in \hat{\Sigma}_{\leq p-1}$ and thus we have reduced the number of priorities. Now consider for $r, s \in [0, p]$ the language

$$E_{X,r,s} = \{\overleftarrow{\tau}_r(u)\#\overrightarrow{\tau}_s(v) \mid X \overset{*}{\Rightarrow} uXv,\ u \in \Sigma^*_{\leq r} r \Sigma^*_{\leq r},\ v \in \Sigma^*_{\leq s} s \Sigma^*_{\leq s}\}.$$

For the language $E_{X,r,s}$, it is easy to construct a context-free grammar:

▶ **Lemma 6.3.** *Given $\mathcal{G}$, a non-terminal X, and $r, s \in [0, p]$, one can construct a grammar $\mathcal{E}_{X,r,s}$ for $E_{X,r,s}$ of linear size.*

Defining the sets $E_{X,r,s}$ with fresh zero-priority letters $\#$, $\overleftarrow{\#}$, $\overrightarrow{\#}$ is a key trick in our construction: Note that each word in $E_{X,r,s}$ is of the form $u\overleftarrow{\#}v\#w\overrightarrow{\#}x$ for $u, v, w, x \in \Sigma^*_{\leq p-1}$. The segments u, v, w, x come from different blocks of the entire generated word, so applying the block downward closure construction recursively to $E_{X,r,s}$ must guarantee that these segments embed as if they were blocks. However, there are only a bounded number of segments. Thus, we can reduce the number of priorities while retaining the block behavior by using fresh zero-priority letters. This is formalized in the following lemma:

▶ **Lemma 6.4.** *For $u, u', v, v' \in \Sigma^*_{\leq p}$, we have $u\#v \preccurlyeq_{\mathsf{B}} u'\#v'$ iff both (i) $u \preccurlyeq_{\mathsf{B}} u'$ and (ii) $v \preccurlyeq_{\mathsf{B}} v'$.*

Language of repeated words. Roughly speaking, the language $E_{X,r,s}$ captures the "ends" of words derived in derivations $X \overset{*}{\Rightarrow} uXv$ with $u \in \Sigma^+_{\max r}$ and $v \in \Sigma^+_{\max s}$: On the left, it keeps everything that is not between two occurrences of r and on the right, it keeps everything not between two occurrences of s. We now need languages that capture the infixes that can occur between r's and s's, respectively. Intuitively, these are the words that can occur again and again in words derived from X. There is a "left version" and a "right version". We set for $r, s \in [1, p]$:

$$\overleftarrow{R}_{X,r,s} = \{yr \mid y \in \Sigma^*_{\leq r-1},\ \exists x, z \in \Sigma^*_{\leq r},\ v \in \Sigma^+_{\max s} \colon X \overset{*}{\Rightarrow} xryrzXv\}$$
$$\overrightarrow{R}_{X,r,s} = \{ys \mid y \in \Sigma^*_{\leq s-1},\ \exists u \in \Sigma^+_{\max r},\ x, z \in \Sigma^*_{\leq r} \colon X \overset{*}{\Rightarrow} uXxsysz\}.$$

The case where one side has highest priority zero must be treated slightly differently: There are no enveloping occurrences of some $r, s \in [1, p]$. However, we can overapproximate those words by the set of all words over a particular alphabet. Specifically, for $r, s \in [0, p]$, we set

$$\overrightarrow{R}_{X,0,s} = \{a \in \Sigma_{\leq 0} \mid \exists u \in \Sigma^+_{\max 0},\ v \in \Sigma^+_{\max s} \colon X \overset{*}{\Rightarrow} uXv,\ a \text{ occurs in } u\}$$
$$\overleftarrow{R}_{X,r,0} = \{a \in \Sigma_{\leq 0} \mid \exists u \in \Sigma^+_{\max r},\ v \in \Sigma^+_{\max 0} \colon X \overset{*}{\Rightarrow} uXv,\ a \text{ occurs in } v\}$$

▶ **Lemma 6.5.** *Given $\mathcal{G}$, a non-terminal X, and $r, s \in [0, p]$, one can construct grammars $\overleftarrow{\mathcal{R}}_{X,r,s}$, $\overrightarrow{\mathcal{R}}_{X,r,s}$ for $\overleftarrow{R}_{X,r,s}$, $\overrightarrow{R}_{X,r,s}$, respectively, of linear size.*

Overapproximating derivable words. The languages $E_{X,r,s}$ and $\overleftarrow{R}_{X,r,s}$ and $\overrightarrow{R}_{X,r,s}$ now serve to define overapproximations of the set of $(u, v) \in \Sigma^+_{\max r} \times \Sigma^+_{\max s}$ with $X \overset{*}{\Rightarrow} uXv$: One can obtain each such pair by taking a word from $E_{X,r,s}$, replacing $\overleftarrow{\#}$ and $\overrightarrow{\#}$, resp., by words in $r\overleftarrow{R}^*_{X,r,s}$ ($\overleftarrow{R}^*_{X,0,s}$ if $r = 0$) and $s\overrightarrow{R}^*_{X,r,s}$ ($\overrightarrow{R}^*_{X,r,0}$ if $s = 0$), respectively. By choosing the right words from $E_{X,r,s}$, $\overleftarrow{R}_{X,r,s}$, and $\overrightarrow{R}_{X,r,s}$, we can thus obtain $u\#v$. However, this process will also yield other words that cannot be derived. However, the key idea in our construction is that every word obtainable in this way from $E_{X,r,s}$, $\overleftarrow{R}_{X,r,s}$, and $\overrightarrow{R}_{X,r,s}$ will be in the block downward closure of a pair of words derivable using $X \overset{*}{\Rightarrow} \cdot X \cdot$.

Let us make this precise. To describe the set of words obtained from $E_{X,r,s}$, $\overleftarrow{R}_{X,r,s}$, and $\overrightarrow{R}_{X,r,s}$, we need the notion of a substitution. For alphabets Γ_1, Γ_2, a *substitution* is a map $\sigma \colon \Gamma_1 \to 2^{\Gamma_2^*}$ that yields a language in Γ_2 for each letter in Γ_1. Given a word $w = w_1 \cdots w_n$ with $w_1, \dots w_n \in \Gamma_1$, we define $\sigma(w) := \sigma(w_1) \cdots \sigma(w_n)$. Then for $K \subseteq \Gamma_1^*$, we set $\sigma(K) = \bigcup_{w \in K} \sigma(w)$. Now let $\Sigma_{X,r,s} \colon \hat{\Sigma}_{\leq p} \to 2^{\hat{\Sigma}^*_{\leq p}}$ be the substitution that maps every letter in $\Sigma_{\leq p} \cup \{\#\}$ to itself (as a singleton) and maps $\overleftarrow{\#}$ to $r\overleftarrow{R}^*_{X,r,s}$ and $\overrightarrow{\#}$ to $s\overrightarrow{R}^*_{X,r,s}$. Now our observation from the previous paragraph can be phrased as:

▶ **Lemma 6.6.** *For every $u\#v \in \Sigma_{X,r,s}(E_{X,r,s})$, there are $u' \in \Sigma^+_{\max r}$ and $v' \in \Sigma^+_{\max s}$ with $u \preccurlyeq_{\mathsf{B}} u'$, $v \preccurlyeq_{\mathsf{B}} v'$, and $X \overset{*}{\Rightarrow} u'Xv'$.*

Constructing the Kleene grammar. We now construct the Kleene grammar for $\mathcal{L}(\mathcal{G})\!\downarrow_{\mathsf{B}}$ by first computing the grammars $\mathcal{E}_{X,r,s}$, $\overleftarrow{\mathcal{R}}_{X,r,s}$, and $\overrightarrow{\mathcal{R}}_{X,r,s}$ for each non-terminal X and each $r, s \in [1, p]$. Then, since $\mathcal{E}_{X,r,s}$, $\overleftarrow{\mathcal{R}}_{X,r,s}$, and $\overrightarrow{\mathcal{R}}_{X,r,s}$ generate languages with at most $p - 1$ priorities, we can call our construction recursively to obtain grammars $\mathcal{E}'_{X,r,s}$, $\overleftarrow{\mathcal{R}}'_{X,r,s}$, and $\overrightarrow{\mathcal{R}}'_{X,r,s}$, respectively. Then, we add all productions of the grammars $\mathcal{E}'_{X,r,s}$, $\overleftarrow{\mathcal{R}}'_{X,r,s}$, and

$\overrightarrow{\mathcal{R}}'_{X,r,s}$ to $\mathcal{G}'$. Moreover, we make the following modifications: Each production of the form $Y \to \overleftarrow{\#}$ (resp. $Y \to \overrightarrow{\#}$) in $\mathcal{E}_{X,r,s}$ is replaced with $Y \to Z_r \overleftarrow{S}^*_{X,r,s}$ (resp. $Y \to Z_s \overrightarrow{S}^*_{X,r,s}$), where $\overleftarrow{S}_{X,r,s}$ (resp. $\overrightarrow{S}_{X,r,s}$) is the start symbol of $\overleftarrow{\mathcal{R}}'_{X,r,s}$ (resp. $\overrightarrow{\mathcal{R}}'_{X,r,s}$), and Z_r is a fresh non-terminal used to derive r or ε: We also have $Z_r \to r$ for each $r \in [1,p]$ and $Z_0 \to \varepsilon$. Moreover, each production $Y \to \#$ in $\mathcal{E}'_X$ is removed and replaced with a production $Y \to w$ for each production $X \to w$ in $\mathcal{G}$. We call the resulting grammar $\mathcal{G}'$.

Correctness. Let us now observe that the grammar $\mathcal{G}'$ does indeed satisfy $\mathcal{L}(\mathcal{G}')\downarrow_{\mathsf{B}} = \mathcal{L}(\mathcal{G})\downarrow_{\mathsf{B}}$. The inclusion "$\supseteq$" is trivial as $\mathcal{G}'$ is obtained by adding productions. For the converse, we need some terminology. We say that a derivation tree t_1 in $\mathcal{G}'$ is obtained using an *expansion step* from t_0 if we take an X-labeled node x in t_0, where X is a non-terminal from $\mathcal{G}$, and replace this node by a derivation $X \overset{*}{\Rightarrow} uwv$ using newly added productions (i.e. using $\mathcal{E}_{X,r,s}$, $\overleftarrow{\mathcal{R}}_{X,r,s}$, and $\overrightarrow{\mathcal{R}}_{X,r,s}$ and some $Y \to w$ where $X \to w$ was the production applied to x in t_0). Then by construction of $\mathcal{G}'$, any derivation in $\mathcal{G}'$ can be obtained from a derivation in $\mathcal{G}$ by finitely many expansion steps. An induction on the number of expansion steps shows:

▶ **Lemma 6.7.** *We have* $\mathcal{L}(\mathcal{G}')\downarrow_{\mathsf{B}} = \mathcal{L}(\mathcal{G})\downarrow_{\mathsf{B}}$.

Acyclic derivations suffice. Now that we have the grammar $\mathcal{G}'$ with $\mathcal{L}(\mathcal{G}')\downarrow_{\mathsf{B}} = \mathcal{L}(\mathcal{G})\downarrow_{\mathsf{B}}$, it remains to show that every word in $\mathcal{G}'$ can be derived using an acyclic derivation:

▶ **Lemma 6.8.** $\mathsf{acyclic}(\mathcal{G}')\downarrow_{\mathsf{B}} = \mathcal{L}(\mathcal{G})\downarrow_{\mathsf{B}}$.

Essentially, this is due to the fact that any repetition of a non-terminal X on some path means that we can replace a corresponding derivation $X \overset{*}{\Rightarrow} uXv$ by using new productions from $\mathcal{E}'_{X,r,s}$, $\overleftarrow{\mathcal{R}}'_{X,r,s}$, and $\overrightarrow{\mathcal{R}}'_{X,r,s}$. Since these also have the property that every derivation can be made acyclic, the lemma follows. See the full version for details.

Complexity analysis. To estimate the size of the constructed grammar, let $f_p(n)$ be the maximal number of non-terminals of a constructed Kleene grammar for an input grammar with n non-terminals over p priorities. By Lemmas 6.3 and 6.5, there is a constant c such that each grammar $\mathcal{E}_X$, $\overleftarrow{\mathcal{R}}_X$, and $\overrightarrow{\mathcal{R}}_X$ has at most cn non-terminals. Furthermore, $\mathcal{G}'$ is obtained by applying our construction to $3n(p+1)^2$ grammars with $p-1$ priorities of size cn, and adding Z_p. Thus $f_p(n) \le n + 3n(p+1)^2 f_{p-1}(cn) + 1$. Since $f_{p-1}(n) \ge 1$, we can simplify to $f_p(n) \le 4n(p+1)^2 f_{p-1}(cn)$. It is easy to check that $f_0(n) \le 4n+1 \le 5n$, because $\mathcal{E}_{X,0,0}$ and $\overleftarrow{\mathcal{R}}_{X,0,0}$ and $\overrightarrow{\mathcal{R}}_{X,0,0}$ each only have one non-terminal. Hence $f_p(n) \le (4n(p+1)^2)^p f_0(c^p n) \le (4n(p+1)^2) \cdot 4(c^p n)$, which is exponential in the size of $\mathcal{G}$.

7 Conclusion

We have initiated the study of computing priority and block downward closures for infinite-state systems. We have shown that for OCA, both closures can be computed in polynomial time. For CFL, we have provided a doubly exponential construction.

Many questions remain. First, we leave open whether the doubly exponential bound for context-free languages can be improved to exponential. An exponential lower bound is easily inherited from the exponential lower bound for subwords [6]. Moreover, it is an intriguing question whether computability of subword downward closures for vector addition systems [17], higher-order pushdown automata [18], and higher-order recursion schemes [12] can be strengthened to block and priority downward closures.

References

1 Parosh Aziz Abdulla, Luc Boasson, and Ahmed Bouajjani. Effective lossy queue languages. In Fernando Orejas, Paul G. Spirakis, and Jan van Leeuwen, editors, *Automata, Languages and Programming, 28th International Colloquium, ICALP 2001, Crete, Greece, July 8-12, 2001, Proceedings*, volume 2076 of *Lecture Notes in Computer Science*, pages 639–651. Springer, 2001. doi:10.1007/3-540-48224-5_53.

2 Ashwani Anand and Georg Zetzsche. Priority downward closures, 2023. arXiv:2307.07460.

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

4 Mohamed Faouzi Atig, Dmitry Chistikov, Piotr Hofman, K. Narayan Kumar, Prakash Saivasan, and Georg Zetzsche. The complexity of regular abstractions of one-counter languages. In *Proceedings of the 31st Annual ACM/IEEE Symposium on Logic in Computer Science*, LICS '16, pages 207–216, New York, NY, USA, 2016. Association for Computing Machinery. doi:10.1145/2933575.2934561.

5 Mohamed Faouzi Atig, Roland Meyer, Sebastian Muskalla, and Prakash Saivasan. On the upward/downward closures of Petri nets. In Kim G. Larsen, Hans L. Bodlaender, and Jean-François Raskin, editors, *42nd International Symposium on Mathematical Foundations of Computer Science, MFCS 2017, August 21-25, 2017 – Aalborg, Denmark*, volume 83 of *LIPIcs*, pages 49:1–49:14. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2017. doi:10.4230/LIPIcs.MFCS.2017.49.

6 Georg Bachmeier, Michael Luttenberger, and Maximilian Schlund. Finite automata for the sub- and superword closure of cfls: Descriptional and computational complexity. In Adrian-Horia Dediu, Enrico Formenti, Carlos Martín-Vide, and Bianca Truthe, editors, *Language and Automata Theory and Applications – 9th International Conference, LATA 2015, Nice, France, March 2-6, 2015, Proceedings*, volume 8977 of *Lecture Notes in Computer Science*, pages 473–485. Springer, 2015. doi:10.1007/978-3-319-15579-1_37.

7 Pascal Baumann, Moses Ganardi, Rupak Majumdar, Ramanathan S. Thinniyam, and Georg Zetzsche. Context-bounded analysis of concurrent programs (invited talk). In Kousha Etessami, Uriel Feige, and Gabriele Puppis, editors, *50th International Colloquium on Automata, Languages, and Programming, ICALP 2023, July 10-14, 2023, Paderborn, Germany*, volume 261 of *LIPIcs*, pages 3:1–3:16. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2023. doi:10.4230/LIPIcs.ICALP.2023.3.

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

9 Pascal Baumann, Rupak Majumdar, Ramanathan S. Thinniyam, and Georg Zetzsche. Context-bounded verification of thread pools. *Proc. ACM Program. Lang.*, 6(POPL):1–28, 2022. doi:10.1145/3498678.

10 J. Berstel. *Transductions and Context-Free Languages*. Vieweg+Teubner Verlag, 1979.

11 S. Blake, D. Black, M. Carlson, Elwyn B. Davies, Zheng Wang, and Walter Weiss. An architecture for differentiated services. *RFC*, 2475:1–36, 1998.

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

13 Bruno Courcelle. On constructing obstruction sets of words. *Bulletin of the EATCS*, 44:178–186, January 1991.

14 Jean Goubault-Larrecq, Simon Halfon, Prateek Karandikar, K. Narayan Kumar, and Philippe Schnoebelen. The ideal approach to computing closed subsets in well-quasi-ordering. *CoRR*, abs/1904.10703, 2019. `arXiv:1904.10703`.

15 Hermann Gruber, Markus Holzer, and Martin Kutrib. The size of higman–haines sets. *Theoretical Computer Science*, 387(2):167–176, 2007. Descriptional Complexity of Formal Systems. `doi:10.1016/j.tcs.2007.07.036`.

16 Christoph Haase, Sylvain Schmitz, and Philippe Schnoebelen. The Power of Priority Channel Systems. *Logical Methods in Computer Science*, Volume 10, Issue 4, December 2014. `doi:10.2168/LMCS-10(4:4)2014`.

17 Peter Habermehl, Roland Meyer, and Harro Wimmel. The downward-closure of Petri net languages. In Samson Abramsky, Cyril Gavoille, Claude Kirchner, Friedhelm Meyer auf der Heide, and Paul G. Spirakis, editors, *Automata, Languages and Programming, 37th International Colloquium, ICALP 2010, Bordeaux, France, July 6-10, 2010, Proceedings, Part II*, volume 6199 of *Lecture Notes in Computer Science*, pages 466–477. Springer, 2010. `doi:10.1007/978-3-642-14162-1_39`.

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

19 Leonard H. Haines. On free monoids partially ordered by embedding. *Journal of Combinatorial Theory*, 6(1):94–98, 1969. `doi:10.1016/S0021-9800(69)80111-0`.

20 Graham Higman. Ordering by Divisibility in Abstract Algebras. *Proceedings of the London Mathematical Society*, s3-2(1):326–336, January 1952. `doi:10.1112/plms/s3-2.1.326`.

21 Jean-Yves Le Boudec. The asynchronous transfer mode: a tutorial. *Computer Networks and ISDN Systems*, 24(4):279–309, 1992. The ATM-Asynchronous Transfer Mode. `doi:10.1016/0169-7552(92)90114-6`.

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

23 Richard Mayr. Undecidable problems in unreliable computations. In Gaston H. Gonnet and Alfredo Viola, editors, *LATIN 2000: Theoretical Informatics*, pages 377–386, Berlin, Heidelberg, 2000. Springer Berlin Heidelberg.

24 Jeffrey Shallit. *A Second Course in Formal Languages and Automata Theory*. Cambridge University Press, 2008. `doi:10.1017/CBO9780511808876`.

25 Salvatore La Torre, Anca Muscholl, and Igor Walukiewicz. Safety of parametrized asynchronous shared-memory systems is almost always decidable. In Luca Aceto and David de Frutos-Escrig, editors, *26th International Conference on Concurrency Theory, CONCUR 2015, Madrid, Spain, September 1.4, 2015*, volume 42 of *LIPIcs*, pages 72–84. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2015. `doi:10.4230/LIPIcs.CONCUR.2015.72`.

26 Jan van Leeuwen. Effective constructions in well-partially-ordered free monoids. *Discrete Mathematics*, 21(3):237–252, 1978.

27 Georg Zetzsche. An approach to computing downward closures. In Magnús M. Halldórsson, Kazuo Iwama, Naoki Kobayashi, and Bettina Speckmann, editors, *Automata, Languages, and Programming – 42nd International Colloquium, ICALP 2015, Kyoto, Japan, July 6-10, 2015, Proceedings, Part II*, volume 9135 of *Lecture Notes in Computer Science*, pages 440–451. Springer, 2015. `doi:10.1007/978-3-662-47666-6_35`.

28 Georg Zetzsche. Computing downward closures for stacked counter automata. In Ernst W. Mayr and Nicolas Ollinger, editors, *32nd International Symposium on Theoretical Aspects of Computer Science, STACS 2015, March 4-7, 2015, Garching, Germany*, volume 30 of *LIPIcs*, pages 743–756. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2015. `doi:10.4230/LIPIcs.STACS.2015.743`.

29 Georg Zetzsche. The complexity of downward closure comparisons. In Ioannis Chatzigiannakis, Michael Mitzenmacher, Yuval Rabani, and Davide Sangiorgi, editors, *43rd International Colloquium on Automata, Languages, and Programming, ICALP 2016, July 11-15, 2016, Rome, Italy*, volume 55 of *LIPIcs*, pages 123:1–123:14. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2016. `doi:10.4230/LIPIcs.ICALP.2016.123`.

30 Georg Zetzsche. Separability by piecewise testable languages and downward closures beyond subwords. In Anuj Dawar and Erich Grädel, editors, *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2018, Oxford, UK, July 09-12, 2018*, pages 929–938. ACM, 2018. `doi:10.1145/3209108.3209201`.