The Tree Width of Automata with Auxiliary Storage*

P. Madhusudan Gennaro Parlato

University of Illinois, Urbana-Champaign, USA
{madhu, parlato}@illinois.edu

April 28, 2010

Abstract

We propose a generalization of results on the decidability of emptiness for several restricted classes of sequential and distributed automata with auxiliary storage (stacks, queues) that have recently been proved. Our generalization relies on reducing emptiness of these automata to finite-state *graph automata* (without storage) defined on monadic second-order (MSO) definable graphs of bounded tree-width, where the graph structure encodes the mechanism provided by the auxiliary storage. Our results outline a uniform mechanism to derive emptiness algorithms for automata, explaining and simplifying several existing results, as well as proving new decidability theorems.

1 Introduction

Several classes of automata with auxiliary storage have been defined over the years that have a decidable emptiness problem. Classic models like pushdown automata utilizing a *stack* have a decidable emptiness problem [10], and several new models like restricted classes of multi-stack pushdown automata, automata with queues, automata with both stacks and queues, have been proved decidable recently [14, 5, 11, 12]. These automata emptiness decidability results have often been motivated for *model-checking* systems, as emptiness corresponds to reachability of an error state, while stacks model *control recursion* in programs, and queues model *FIFO communication between processes* [14, 12, 16]. The identified decidable restrictions are, for the most part, *awkward* in their definitions— e.g. emptiness of multi-stack pushdown automata where pushes to any stack is allowed at any time, but popping is restricted to the first non-empty stack is decidable! [5]. Yet, relaxing these definitions to more natural ones seems to either destroy decidability or their power. It is hence natural to ask: why are these automata decidable? Is there a common underlying principle that explains their decidability?

We propose, in this paper, a general criterion that uniformly explains many such results— *several restricted uses of auxiliary storage are decidable because they can be simulated by graph automata working on graphs that capture the storage, and are also of bounded tree-width.*

More precisely, we can show, using generalizations of known results on the decidability of the *satisfiability* problem for monadic second-order logic (MSO) on bounded tree-width graphs [15, 6], that graph automata on *MSO-definable* graphs of bounded tree-width are decidable. Graph automata [17] are finite-state automata (without auxiliary storage) that accept or reject graphs using *tilings of the graph using states*,

*This work was partially supported by NSF CAREER award #0747041 and NSF Award #0917229.

where the restrictions on tiling determine the graphs that get accepted. The general decidability of emptiness of graph automata on MSO-definable graphs follows since the existence of acceptable tilings is MSO-definable, and in fact the emptiness algorithm works in time $|GA|^{O(k)}$, where GA is the graph automaton and k is the tree-width.

The notion of tree-width has played a role in the study of parameterized algorithms, especially through Courcelle's theorem [6] that argues that any problem that can be stated in MSO over graphs of bounded tree-width is decidable in linear time [6, 8]. This problem corresponds to the *membership* problem (given a graph G, does it satisfy φ?), while the emptiness problem for automata translates to a *satisfiability* problem (given φ, is there a graph that satisfies φ?). The notion of bounding the tree-width of graphs is much more compelling restriction when considering the satisfiability problem, as it is known the satisfiability problem for MSO (*with edge quantification*) on a class of finite graphs is decidable *if and only if* the tree-width of the class is bounded [15, 6]. (A long-standing open conjecture is that MSO without edge quantification is decidable on a class of graphs iff it has bounded clique-width.)

Our primary results in this paper show that several sequential/distributed automata with an auxiliary storage (we consider stacks and queues only in this paper), can be realized as *graph* automata working on single or multiple word-graphs decorated with special edges to capture the mechanism of the storage. Intuitively, a symbol that gets stored in a stack/queue and later gets retrieved can be simulated in a graph by having a special edge between the point where the symbol gets stored to the point where it gets retrieved. A graph automaton can retrieve the symbol at the retrieval point by using an appropriate tiling of this special edge.

The idea of converting automata with storage to graph automata without storage but working on specialized graphs is that it allows us to examine the *complexity* of storage using the *structure* of the graph that simulates it. We show that many tractable automata can be converted to graph automata working on MSO definable graphs of bounded tree-width, from which decidability of their emptiness follows.

We prove the simulation by graph automata working on MSO-definable bounded tree-width graphs for the following classes of automata:

- Multi-stack pushdown automata with bounded context-switching: This is the class of multi-stack automata where each computation of the automaton can be divided into k stages, where in each stage the automaton touches only one stack (proved decidable first in [14]). We show that they can be simulated by graph automata on graphs of tree-width $O(k)$.

- Multi-stack pushdown automata with bounded phases: These are automata that generalize the bounded-context-switching ones: the computations must be dividable into k phases, for a fixed k, where in each phase the automaton can push onto any stack, but can pop only from one stack (proved decidable recently in [11]). We show that graph automata on graphs of tree-width $O(2^k)$ can simulate them.

- Ordered multi-stack pushdown automata: The restriction here is that there a finite number of stacks that are ordered, and at any time, the automaton can push onto any stack, but pop only from the first non-empty stack. Note that the computation is not cut into phases, as in the above two restrictions. We show that automata on graphs of tree-width $O(n \cdot 2^n)$ (where n is the number of stacks) can simulate them.

- Distributed queue automata on polyforest architectures: Distributed queue automata is a model where finite-state processes at n sites work by communicating to each other using FIFO channels, modeled as queues. It was shown recently, that when the architecture is a polyforest (i.e. the underlying undirected graph is a forest), the emptiness problem is decidable (and for other architectures, it is undecidable) [12]. We

prove that graph automata working on graphs of tree-width (in fact, path-width) n, where n is the number of processes, can simulate distributed queue automata on polyforest architectures.

- Distributed queue automata with stacks on forest architectures: When we endow each process in a distributed queue automaton with a local stack, it turns out that if the automaton is *well-queuing* and the architecture is a forest, the emptiness problem is decidable [12]. The well-queuing condition demands that a process may dequeue from a queue only when its local stack is empty. Furthermore, it is known that simply dropping the well-queuing condition or dropping the condition that the architecture be a forest, makes emptiness undecidable [12]. We prove that graph automata that work on graphs that simulate both the local stacks and the queues can capture these automata, and for well-queuing automata over forest architectures, the graphs are of tree-width $O(n)$, where n is the number of processes.

The *graphs* on which the graph automata need to work to realize the above automata are also, surprisingly, uniform. For the first three classes of multi-stack automata, the graphs are simply a single word endowed with multiple sets of *nesting edges*, one for each stack. For distributed queue automata, the graphs are composed of n different word structures, one for each process, with queue edges connecting enqueing vertices to dequeuing vertices, and, if the processes have stacks, have nesting edges at each process to capture the local stack.

The tree-decompositions for these graphs, as well as the proofs that the decompositions give bounded tree-width for the restrictions, are quite involved, and are tailored to exploit the restriction placed on the automata.

The idea of interpreting stacks as nesting edges was motivated by the work relating *visibly pushdown automata* with *nested word automata* [1, 2, 3], where nesting edges capture a visible stack. Surveying the known decidable automata restrictions led us to this uniform framework for proving decidability. The automata variants we study were often first proved to be decidable by using very different means— bounded context-switching multi-stack automata were shown to be decidable using regularity of *tuples* of reachable configurations [14], ordered multi-stack automata were shown decidable using manipulations of associated grammars, followed by a Parikh theorem [5], and distributed queue automata with stacks were shown decidable by reductions to bounded-phase automata [12].

Apart from giving uniform proofs of older results, our theorems also lead to new consequences. First, automata with multi-stacks are decidable when their graphs are restricted to graphs of bounded tree-width, and in fact even bounded *clique-width* graphs [7]; this result *generalizes all the above multi-stack sequential automata*. Second, all our results extend smoothly to automata over infinite behaviors— for example, it follows easily that ordered multi-stack *Büchi* or *parity* automata on infinite words have a decidable emptiness problem (since these conditions are easily expressed in MSO). Third, several variants of the restrictions can be proved immediately decidable— for instance, consider the model where we restrict multi-stack automata to k phases, where in each phase, there is only one stack that is *pushed* into (but arbitrary pops of stacks are allowed), then it easily follows that emptiness is decidable, as the *graphs* corresponding to these automata are precisely the same as those of bounded phase automata, save for the orientation of the linear and nesting edges, and hence has the same tree-width.

In summary, we provide a uniform theory that can be used to prove the decidability of automata with storage.

Due to the variety of models we consider, we present only briefly the main results; the technical definitions of the models and the proofs of the decompositions leading to bounded tree-width for the various restrictions can be found in the Appendix.

2 Logics, graphs, graph automata, tree-width, and emptiness

We start by defining, in this section, graph automata that work on edge-labeled finite directed graphs, and show that the emptiness problem for these automata is decidable over any MSO-definable class of graphs of bounded tree-width. This result is derived from classical known results on interpretations of graphs on trees, and we sketch the derivations here.

Monadic second-order logic on graphs: Fix a finite alphabet (set) Σ. A Σ-labeled graph is a structure $(V, \{E_a\}_{a\in\Sigma})$, where V is a finite non-empty set of vertices, and each $E_a \subseteq V \times V$ is a set of a-labeled directed edges. We will assume, throughout this paper, that for any vertex v, there is at most one incoming a-labeled edge and at most one outgoing a-labeled edge.

We view graphs as logical structures, with V as the universe, and each set of edges E_a as a binary relation on vertices. Monadic second-order logic (MSO) is now the standard logic on these structures. We fix a countable set of first-order variables (we will denote these as x, y, etc.) and another countable set of set variables (denoted as X, Y, etc.). MSO is given by the following syntax:

$$\varphi ::= x = y \mid E_a(x, y) \mid x \in X \mid \varphi \vee \varphi \mid \neg\varphi \mid \exists x.\varphi \mid \exists X.\varphi,$$

where $a \in \Sigma$. The semantics is the standard one, with first-order and set variables interpreted as vertices and sets of vertices.[1]

We say a class of Σ-labeled graphs $\mathcal{C}$ is MSO-definable, if there is an MSO formula φ such that $\mathcal{C}$ is the precise class of Σ-labeled graphs that satisfy φ.

Graph automata: Fix a *class* of Σ-labeled graphs $\mathcal{C}$. A graph automaton (GA) on $\mathcal{C}$ is a tuple $(Q, \{T_a\}_{a\in\Sigma}, \mathit{type})$, where Q is a finite set of states, each $T_a \subseteq Q \times Q$ is a *tiling relation*, and $\mathit{type} : Q \to 2^\Sigma \times 2^\Sigma$ is the type-relation.

Intuitively, a graph automaton will accept a graph if there is a way to tile (label) the vertices by states so that the tiling relation is satisfied by vertices adjacent to each other, and further satisfies the type-relation. The type-relation associates each state to a pair (In, Out) of labels, and in order for a state to decorate a vertex, we require its type to match the edges incident on it— the labels of incoming (and outgoing) edges must be precisely In (and Out).

Formally, we say that a graph automaton $(Q, \{T_a\}_{a\in\Sigma}, \mathit{type})$ *accepts* a graph $(V, \{E_a\}_{a\in\Sigma})$ if there is a map $\rho : V \to Q$ that satisfies the following conditions:

- For every $(u, v) \in E_a$, with $a \in \Sigma$, $(\rho(u), \rho(v)) \in T_a$.
- For every u, $\mathit{type}(\rho(u)) = (In, Out)$, where $In = \{a \mid \exists v, (v, u) \in E_a\}$ and $Out = \{a \mid \exists v, (u, v) \in E_a\}$.

The language of a graph automaton GA over a class of graphs $\mathcal{C}$, denoted $L(GA)$, is the set of graphs in $\mathcal{C}$ that it accepts.

Note that the notion of an automaton "running" over the graph has been replaced by tiling constraints. Also, we have done away with initial or final states; we will capture these when needed using specially labeled edges in the sequel.

Our notion of graph automata is motivated by definitions of automata on graphs through tilings in the literature [17]. Graph automata can in fact be defined more powerfully (see [17]); however, for our purposes, the above definition will suffice. Most of our results will carry over to generalizations of the above definition.

[1] Note: In the literature, a variant of MSO (called MSO_2) has been considered where both vertices and edges are in a two-sorted universe that allows quantification over edges; that version is stronger than ours, but we shall not need it for our exposition.

Tree-width: We recall the definition of tree-width for graphs. A *tree-decomposition* of a graph (V, E) is a pair (T, bag), where $T = (N, \rightarrow)$ is a tree, and $bag : N \rightarrow 2^V$ is a function that satisfies:

- For every $v \in V$, there is a node $n \in N$ such that $v \in bag(n)$,
- For every edge $(u, v) \in E$, there is a node $n \in N$ such that $u, v \in bag(n)$, and
- If $u \in bag(n)$ and $u \in bag(n')$, for nodes $n, n' \in N$, then for every n'' that lies on the unique path connecting n and n', $u \in bag(n'')$.

The *width* of a tree decomposition is the size of the largest bag in it, minus one; i.e. $max_{n \in N}\{|bag(n)|\} - 1$.

The *tree-width* of a graph is the *smallest* of the widths of any of its tree decompositions.

Emptiness of graph automata on graphs of bounded tree-width: We now show that emptiness of graph automata is decidable, when evaluated over graphs that are definable in MSO and are also of bounded tree-width.

First, we recall a classical result that the *satisfiability* problem for MSO is decidable on the class of *all* graphs of tree-width k (for a fixed k) [15, 6].

Theorem 2.1 (Courcelle, Seese) *The problem of checking, given $k \in \mathbb{N}$ and $\varphi \in MSO$ over Σ-labeled graphs, whether there is a Σ-labeled graph G of tree-width at most k that satisfies φ, is decidable.*

Note that the above certainly does *not* imply that satisfiability of MSO is decidable on *any* class of graphs of bounded tree-width (take a non-recursive class of linear-orders/words for a counter-example). However, an immediate corollary is that satisfiability of MSO is also decidable on any *MSO-definable* class of graphs $\mathcal{C}$ of bounded tree-width (a class of graphs $\mathcal{C}$ is MSO definable if there is a MSO formula $\varphi_{\mathcal{C}}$ such that $\mathcal{C}$ is the precise class of graphs that satisfies $\varphi_{\mathcal{C}}$). If φ is the MSO formula, we can instantiate the above theorem for $\varphi_{\mathcal{C}} \wedge \varphi$) to obtain the following result:

Corollary 2.2 *Let $\mathcal{C}$ be a class of MSO definable Σ-labeled graphs. The problem of checking, given $k \in \mathbb{N}$ and an MSO-formula φ, whether there is a graph $G \in \mathcal{C}$ of tree-width at most k that satisfies φ, is decidable.*

We can now prove that the emptiness problem for graph automata is decidable when restricted to bounded tree-width graphs over an MSO-definable class of graphs. Intuitively, we can write an MSO formula φ that checks whether there is a proper tiling of a graph by the graph automaton that respects the tiling and typing relations. This formula will essentially use an existential quantification of a set of sets X_a (for each $a \in \Sigma$) to "guess" a tiling, and check whether the tiling and typing is proper, using universal first-order quantification on vertices. We can then instantiate the above corollary with this formula show decidability of graph automata emptiness. In fact, using a direct automaton construction on trees, we can show the complexity of graph-automata emptiness as well (see Appendix H for a gist of proof) and obtain this result:

Theorem 2.3 *Let $\mathcal{C}$ be a class of MSO definable Σ-labeled graphs. The problem of checking, given $k \in \mathbb{N}$ and a graph automaton GA, whether there is some $G \in \mathcal{C}$ of tree-width at most k that is accepted by GA, is decidable, and decidable in time $|GA|^{O(k)}$.*

The above theorem will be the key result we will use to uniformly prove decidability results in this paper. For various restrictions of sequential and distributed automata with auxiliary storage, we will translate them to graph automata over MSO-definable graphs, show that the relevant graphs are of bounded tree-width, and use the above theorem to prove decidability of emptiness.

3 Multi-stack Pushdown Automata

In this section, we will consider several restricted classes of multi-stack automata and show that their emptiness problem is decidable by showing that their behaviors correspond to *multiply-nested word graphs* of bounded tree-width, and that graph automata working on these graphs can simulate them.

A multi-stack pushdown automaton is an automaton with finite control and equipped with a finite number of stacks. A transition of this automaton consists in pushing or popping a symbol from a specified stack and changing its control or simply an internal move that affects only the control state without alteration of the stacks' contents.

Definition 3.1 (MULTI-STACK PUSHDOWN AUTOMATA**)** *For a fixed integer n, an n-stack* pushdown automaton *(n-PDA) is a tuple $M = (Q, q_0, \Gamma, \delta, Q_F)$, where Q is a finite set of states, $q_0 \in Q$ is the initial state, Γ is a finite stack alphabet, $Q_F \subseteq Q$ is the set of final states, and $\delta = \langle \delta_{push}, \delta_{pop}, \delta_{int} \rangle$ where*

- *$\delta_{push} \subseteq (Q \times Q \times \Gamma \times [n])$ is the set of push moves,*
- *$\delta_{pop} \subseteq (Q \times \Gamma \times Q \times [n])$ is the set of pop moves, and*
- *$\delta_{int} \subseteq (Q \times Q)$ is the set of internal moves.*

A multi-stack pushdown automata ($mPDA$) is an n-stack pushdown automaton, for some $n \in \mathbb{N}$.

The semantics of n-stack PDAs are as expected; however, we capture the behaviors of the machine by making *visible* the actions performed on the stacks. Behaviors are hence over the alphabet $B = \{int,\ push_1, \ldots push_n,\ pop_1, \ldots, push_n\}$. We refer the reader to Appendix A for precise definitions.

The *emptiness* problem for an mPDA M is the problem of checking if $Beh(M)$ is empty (or equivalently, whether there is a run of the mPDA).

Multiply-nested words: In the following we show that mPDAs can be naturally encoded as graph automata on the class of *multiply nested word* graphs, and the emptiness problem can be solved on the former by checking the emptiness problem of the latter. We start by defining multiply nested words graphs.

Definition 3.2 (MULTIPLY NESTED WORDS**)** *For a given integer n, an n-*nested word *(n-NW) is a tuple $N = (V, Init, Final, L, \{E_j\}_{j \in [n]})$, where*

- *V is the set of vertices;*
- *$L \subseteq V \times V$ is a non-reflexive (successor) edge relation such that L^* defines a linear ordering $<_L$ on the vertices of V;*
- *If x is the minimal element w.r.t. L, then $Init = \{(x, x)\}$; if x is the maximal element w.r.t. L, then $Final = \{(x, x)\}$;*
- *$E_j \subseteq V \times V$ is a nesting relation, for every $j \in [n]$. A nesting relation E_j is a relation that satisfies the following properties: for all $u, u', v, v' \in V$ and $j, j' \in [n]$,*
  - *if $E_j(u, v)$ then $u <_L v$ holds;*
  - *if $E_j(u, v)$ and $E'_j(u, v')$ then $v = v'$; and if $E_j(u, v)$ and $E_j(u', v)$ then $u = u'$;*
  - *if $E_j(u, v)$ and $E_j(u', v')$ and $u <_L u'$ then either $v <_L u'$ or $v' <_L v$ holds.*
  - *if $j \neq j'$, $E_j(u, v)$, and $E_{j'}(u', v')$, then u, v, u', v' are all different.*

A multiply nested word (mNW) is an n-nested word, for some $n \in \mathbb{N}$.

Figure 1 illustrates a 2-nested word.

Intuitively, mNWs are meant to capture the behaviors of runs of mPDAs, where the stacks are compiled down to edges in the graph: the relation L relates consecutive actions in the run, while the nesting edge relation E_j captures the matching push-pop relation of stack j, for every stack index $j \in [n]$. The self-looping edges $Init$ and $Final$ capture the initial and final vertex with respect to L.

The properties of multiply nested words (Definition 3.2) can be easily stated in MSO:

Proposition 3.3 *For any integer n, the class of n-NWs is MSO definable.*

We can define a 1-to-1 correspondence nw between the set of behaviors of the n-PDA M and the class of n-NWs. Given an M run ρ with $beh(\rho) = a_1 a_2 \dots a_m$, the corresponding nested word graph n-NW N is as follows. The set of vertices of N is $V = \{v_1, v_2, \dots, v_m, v_{m+1}\}$, the relation L is such that $L(v_i, v_j)$ holds iff $j = i + 1$. The edge relation E_j is defined as follows. On the word $beh(\rho)$ there are intrinsic relations that match corresponding pushes and pops of the same stack, and since we assumed that all the stacks at the end of a run are empty, we have that in $beh(\rho)$ every symbol $push_j$ is matched with a future symbol pop_j and vice-versa. Thus the edge relation E_j is defined as: $E_j(v_i, v_h)$ holds if and only if $a_i = push_j$, $a_h = pop_j$ and the pair (i, h) is a matching pair of push and pop actions in $beh(\rho)$. It is easy to see that this is a 1-to-1 correspondence.

Given any n-PDA M, we can easily translate it to a graph automaton that accepts the mNWs corresponding to the behaviors of M. Intuitively, whenever the n-PDA pushes onto the i'th stack, the graph automaton decorates the corresponding node in the nested word graph with the symbol pushed, and when this symbol gets popped later, the graph automaton, using tiling conditions on the nested edge, will recover the symbol. Hence, by using tilings on the nested edges, the graph automaton can work *without a stack*, and capture the semantics of the n-PDA precisely. We hence have:

Lemma 3.4 *For every n-PDA M, there is a (constructible) graph automaton GA on n-nested words such that $nw(Beh(M)) = L(GA)$. Hence $Beh(M) \neq \emptyset$ iff $L(GA) \neq \emptyset$.*

Note that 1-PDAs are basic pushdown automata, whose emptiness problem is decidable. The emptiness problem for n-PDA is well-known to be undecidable when $n > 1$ [10]. Thus, Lemma 3.4 can be used to show that the class of n-NWs, with $n > 1$, have unbounded tree-width.

Lemma 3.5 *The class of 1-NWs has tree-width 2. For any integer $n > 1$, the class of n-NWs has unbounded tree-width.*

Tree-decompositions of multiply nested words. In order to show restricted versions of $mPDA$s have a decidable emptiness, problem, we will first define a tree-decomposition for multiply nested words which we will use to prove both the bounded-phase automata as well as ordered automata (it turns out that bounded context-switching automata have a simpler tree decomposition).

Definition 3.6 (TREE DECOMPOSITION**)** *For any n-NW $N = (V, Init, Final, L, \{E_j\}_{j \in [n]})$, the tree-decomposition of N, $nw\text{-}td(N) = (T, bag)$ is defined as: The binary tree $T = (V, \rightarrow)$ is defined as:*

- *The set of nodes of the tree T are the vertices V of the N.*
- *If $E_j(u, v)$ holds, for any $j \in [n]$, then v is the right-child of u in T*

- *if $L(u,v)$ holds and for all $j \in [n]$ and $z \in V$, $E_j(z,v)$ does not hold, then v is the left-child of u.*

The function bag associates the minimal set of vertices to each node of T that satisfies the following:

- *$v \in bag(v)$, for all $v \in V$.*
- *if u is the parent of v in T, then $u \in bag(v)$, for every $v \in V$.*
- *if $L(u,v)$ holds then $u \in bag(z)$, for all vertices z such that z is on the unique path from u to v in T.*

Figure 2 illustrates a tree-decomposition for the 2-nested graph in Figure 1.

In the above definition of the tree-decomposition of an n-NW N, the vertices of T are the same as the vertices of N. The root of T is the minimal vertex in N according to the linear ordering induced by L. The nesting-edge-successor of any node, if any, is always its right-child. Otherwise, a vertex v is the left-child of its linear predecessor. Notice that, since for each node v there exists at most one pair (u,j) such that $E_j(u,v)$ holds, and at most one vertex u such that $L(u,v)$ holds, the tree T is uniquely determined by N.

Observe that the tree T captures all the nesting edges in: in fact if $E_j(u,v)$ holds then v must be the right-child of u, and hence $u,v \in bag(u)$. The successor relation L is not always local as the nesting-edge relation is: for example, if $L(u,v)$ and $E_j(z,v)$ hold for some j and z, then v is the right-child of z and not the left-child of u. However, the third property in the definition guaranties that all linear edges are captured by at least one bag, and also validates the requirement that nodes whose bags contain the same vertex in a tree decomposition be connected. Hence, it is clear that $nwt(N)$ defines a unique tree decomposition for every n-NWs (though its width may not be bounded).

Lemma 3.7 *For any multiply nested word graph N, $wt(N)$ is a tree-decomposition of N.*

3.1 Bounded context-switch emptiness

For any $k \in \mathbb{N}$, we say that a behavior word $w \in B^*$ is a *k-context word*, if it belongs to $(\bigcup_{j\in[n]}\{int, push_j, pop_j\}^*)^k$. In other words, w can be factorized as at most k sub-words $w_1 w_2 \dots w_h$ (with $h \leq k$) such that each w_i includes only actions of a single stack and internal actions. Let us define k-CS-$Beh(M)$ to be the set of all k-context behavior words in $Beh(M)$.

The emptiness problem for mPDAs restricted to k contexts is the problem of checking, given an $mPDA$ M, whether the language k-CS-$Beh(M)$ is empty.

As in the general case, the emptiness problem for mPDSs restricted to k contexts can be reduced to the emptiness problem for graph automata, where now the class of graphs to consider is that of mNWs restricted to k-context behaviors. For any $k,n \in \mathbb{N}$, a *k-context-switch n-nested word* is a tuple $N = (V, Init, Final, L, \{E_j\}_{j\in[n]})$ where N is a n-NW and $nw^{-1}(N)$ is a k-context behavior word.

The k-context restriction on multiply nested word graphs is easily expressible as an MSO formula ϕ; this formula will express that the graph can be factored into k segments and only nesting edges of one stack are incident on vertices of a single segment. Along with the MSO formula φ defining the class of mNWs, $\varphi \wedge \phi$ defines the class of all k-context mNWs. Moreover, a tree-decomposition where each stack is encoded as a subtree (in the usual way, as for 1-nested words), under the root, has width at most $k+1$.

Lemma 3.8 *For any $k,n \in \mathbb{N}$, the class of k-context n-NW graphs is MSO definable. Furthermore, for any k-context n-NW, there exists a tree-decomposition of width at most $k+1$.*

From the fact that the emptiness problem for mPDAs restricted to k-contexts is effectively reducible to the emptiness problem for graph automata over k-context mNWs, and using Lemma 3.8, we can instantiate Theorem 2.3 to show the following:

Theorem 3.9 *For any integer k, the emptiness problem for mPDAs restricted to k contexts is decidable. For a fixed k, the emptiness problem is in* PTIME.

The original proof of decidability of reachability of multi-stack automata under a bounded number of context-switches was proved using tuples of automata to store the *configurations of stacks* [14].The above proof is hence very different— it shows that the graph that captures the storage, i.e. multiple stacks with bounded context-switches, has bounded tree-width, and hence admits a decidable emptiness problem.

3.2 Bounded phase emptiness

A word $w \in B^*$ is a *phase* if it belongs to one of the sets $phase_j = (\{int, pop_j\} \cup \bigcup_{i\in[n]}\{push_i\})^*$, for some $j \in [n]$. A phase j describes any sequence of actions in which internal actions, pushes to all stacks, and pops from stack j permitted. A word $w \in B^*$ is a *k-phase* behavior word if it is the concatenation of at most k phases: that is, $w \in (\bigcup_{j\in[n]}(phase_j))^k$. As for the bounded context-switch case we define the set k-$Phase$-$Beh(M)$, for a mPDA M, as the set of all the k-phase words of $Beh(M)$.

The emptiness problem for mPDAs restricted to k phase behaviors asks whether k-$Phase$-$Beh(M)$ is an empty set. mPDAs restricted to bounded phases can be simulated by graph automata on a the class of bounded phase mNWs. For any $k, n \in \mathbb{N}$, a *k-phase n-nested word* N is an n-NW where $nw^{-1}(N)$ is a k-phase behavior word.

Lemma 3.10 *For any $k, n \in \mathbb{N}$, the class of k-phase n-NW graphs is MSO definable. Moreover, the tree-decomposition nw-$td(N)$, where N is any k-phase n-NW, has tree-width at most $2^k + 2^{k-1} + 1$.*

From Lemma 3.10 and Theorem 2.3 we obtain the following theorem, which also matches the precise complexity for this problem [13].

Theorem 3.11 *For any integer k, the emptiness problem for mPDAs restricted to k phases is decidable. Moreover, for a fixed k, the emptiness problem is in* PTIME.

3.3 Ordered emptiness

A run ρ of an nPDA is *ordered* if whenever a pop action happens on the stack $j \in [n]$, then all stacks of index less than j are empty: if $\rho = C_1 \xrightarrow{act_1} C_2 \ldots \xrightarrow{act_{m-1}} C_m$, then for every $i \in [m-1]$, if $act_i = pop_j$ and $C_i = \langle q, s_1, \ldots, s_n\rangle$ then $s_h = \epsilon$, for each $h < j$.

The set $ordered$-$Beh(M)$, for a mPDA M, is the set of all the ordered words of $Beh(M)$.

The emptiness problem for mPDAs restricted to ordered behaviors is the problem of checking the emptiness of $ordered$-$Beh(M)$.

For any $n \in \mathbb{N}$, an *ordered n-nested word* N is an n-NW in which $nw^{-1}(N)$ is a ordered word.

Lemma 3.12 *Let $n \in \mathbb{N}$. The class of ordered n-NW graphs is MSO definable. Furthermore, the tree-decomposition nw-$td(N)$, where N is any ordered n-NW, has width at most $(n+1) \cdot 2^{n-1} + 1$.*

From Lemma 3.12 and Theorem 2.3 to obtain the following theorem, which also matches the precise complexity for this problem [4].

Theorem 3.13 *The emptiness problem for mPDAs restricted to ordered runs is decidable. Moreover, for a fixed number of stacks, the problem is decidable in* PTIME.

4 Distributed Automata with Queues and Stacks

Distributed queue automata with stacks (DQAS) is an automaton model composed of a finite number of processes and a finite number of first-in-first-out (FIFO) channels using which they communicate, and where the local processes are endowed with a single local stack each. Each FIFO queue has a unique sender process that can enqueue onto it, and a unique receiver process that dequeues from it. Due to lack of space, we omit the formal definitions, which the reader can find in the Appendix D.

The behavior of a DQAS is modeled as a tuple $\{w_p\}_{p\in P}$, where each $w_p \in B_p^*$, where $B_p = \{int_p\} \cup (\bigcup_{q\in Q}\{send_{(p,q)}\}) \cup (\bigcup_{q\in Q}\{recv_{(p,q)}\}) \cup \{push_p, pop_p\}$. Intuitively, each w_p is a word describing the actions of the process p, where each action is annotated as either being an internal action, a push or pop onto the local stack, or a send or receive action of a message on a channel. Given such a behavior $\{w_p\}_{p\in P}$, it is easy to see that there is a unique matching of sends with receives (because queues connect a unique source to a unique sink and are FIFO).

A *stack-queue graph* (SQG) captures the behaviors of DQAS as a graph. This graph captures the distributed behavior by modeling local behaviors of the process as *disjoint* linearly ordered sets of vertices with two additional kinds of edges: edges that capture the nesting relation matching pushes and pops of the local processes (like in a nested word), and edges that match send-events of one process with receive-events in others. Formally,

Definition 4.1 (STACK-QUEUE GRAPHS**)** *A* stack-queue graph *(SQG) over the name sets P and Q, is a tuple $SQG = (\ \{(V_p, Init_p, Final_p, L_p, E_p\ )\}_{p\in P},\ \{E_q\}_{q\in Q}\ )$, where*

- *$(V_p, Init_p, Final_p, L_p, E_p\ )$ is a 1-NW, for every $p \in P$;*
- *$V_p \cap V_{p'} = \emptyset$, for all $p, p' \in P$ with $p \neq p'$;*
- *$E_q \subseteq V_p \times V_{p'}$ for two fixed $p, p' \in P$ with $p \neq p'$. Further, for all $u, x \in V_p$ and $v, y \in V_{p'}$, if $(u, v) \in E_q$ and $(x, y) \in E_q$ and $u <_{L_p} x$ holds, then $v <_{L_{p'}} y$.*
- *Any vertex $v \in \bigcup_{p\in P} V_p$ has at most one edge of $(\bigcup_{q\in Q}(E_q) \cup \bigcup_{p\in P}(E_p))$ incident on it.*

Figure 3 illustrates a stack-queue graph for three processes.
The properties defining stack-queue graphs (the definition above) can be easily expressed in MSO:

Lemma 4.2 *For any finite sets P and Q, the class of stack-queue graphs over P and Q is MSO definable.*

The class of stack-queue graphs represent all potential behaviors of any DQSA. The precise queue graphs corresponding to behaviors of a DQSA can be accepted by a graph automaton over queue graphs that decorates each of these graphs with the DQSA states and checks whether there is a run of the DQSA corresponding to the graph. Let us associate a function sqg that associates (as a $1-1$ correspondence), the stack-queue graph corresponding to any tuple of words $\{w_p\}_{p\in P}$. Then,

Lemma 4.3 *For any DQAS M over P and Q, there is an effectively constructible graph automaton on stack-queue graphs over P and Q such that $sqg(Beh(M)) = L(GA)$.*

Stack-queue graphs are complex graphs, and several restrictions are required to make them tractable. In fact, they are of unbounded tree width:

Lemma 4.4 *For any P, Q, where $P \neq \emptyset$ and $Q \neq \emptyset$, the class of stack-queue graphs over P and Q has an unbounded tree-width.*

The *architecture* of a DQSA M is the directed graph that describes the way its processes communicate trough queues: $Arch(M) = (\ P,\ \{\ (Sender(q), Receiver(q)) \mid q \in Q\ \}\ )$.

In [12], it is proved that if the underlying architecture is a directed tree (where each process hence has only one incoming queue) and if the processes are *well-queuing*, then the emptiness problem is decidable for DQASs. The well-queuing assumption demands that each process may dequeue from an incoming queue only when its local stack is empty. The stack-queue graph in Figure 3 corresponds to such a well-queuing behavior. These properties (well-queuing and tree architectures) can be expressed in MSO.

Furthermore, we can prove that these restrictions cause the graphs to be of bounded tree-width. This proof is quite involved, and is given in Appendix E. The idea is to first define the notions of *graph decompositions and their widths* that extends the notion of tree-decompositions. If $\mathcal{H}$ is a class of graphs, then a $\mathcal{H}$-decomposition of a graph G is a graph $H \in \mathcal{H}$ where each node in H has an associated bag of vertices, where every edge in G is in the union of two adjacent bags in H, and where the nodes that contain a vertex of G are connected in H. We then show that stack-queue graphs over an architecture that is a directed tree can be decomposed with a small width onto a *nested word*. This process relies on the observation that the global run can be always be executed in a particular order where messages in queues never go beyond length 1. Then, by using the small tree-width of nested words, we obtain the following result.

Lemma 4.5 *The set of all stack-queue graphs over a pair (P, Q) whose underlying architecture is a directed tree, and are well-queuing, is MSO-definable, and furthermore, have tree-width bounded by $3n - 1$ where n is the number of processes.*

Theorem 4.6 *The emptiness problem for a well-queuing DQAS M with tree-architectures is decidable. The problem is decidable in time $|M|^{O(n)}$, where n is the number of processes of M.*

In fact, the precise analysis of the tree-width that leads to the above theorem *improves* the complexity by one exponential than the one proved in [12], which gives an algorithm doubly exponential in n (see also [9]).

4.1 Distributed Queue Automata without stacks

Distributed Queue Automata without stacks (DQAs) have the same model as that of DQASs except that the local stacks at each process are not present (see Appendix F for precise definitions). Even in this restricted setting, the emptiness problem is *undecidable*. We can capture behaviors using *queue graphs* that are composed of n linear orders, one for each process, with edges connecting matching sends and receives. In general, queue graphs of distributed queue automata without stacks are also of unbounded tree width.

The properties defining queue graphs can be easily expressed in MSO:

Lemma 4.7 *For any two finite sets P and Q, the class of queue graphs over P and Q is MSO definable.*

Also, there is a 1-1 correspondence between behaviors of distributed queue automata and queue graphs:

Lemma 4.8 *For every DQA M, there is a (constructible) graph automaton GA on queue graphs such that $qg(Beh(M)) = L(GA)$. Hence $Beh(M)$ is empty iff $L(GA)$ is empty.*

In [12], it was proved that when the architecture of a DGA is a *polyforest* the emptiness problem is decidable. An architecture $Arch(M)$ of a DQA M is a *polyforest* if the underlying *undirected graph* is acyclic.

To bound the queue graphs of *polyforest* architectures, we note that we can reverse any edge of the graph, without changing its tree-width. Hence, we can direct queuing edges in a way to make the underlying

architecture a directed forest (note that since there are no stacks, the well-queuing assumption is satisfied vacuously, see Appendix G). This resulting graph hence can be interpreted on a linear word (using the same proof as for DQAS, except that now the nesting relation is not needed). Hence we obtain the following:

Lemma 4.9 *Let P and Q be two finite sets. Then, the class of polyforest queue graphs over P and Q has tree-width (even path-width) upper-bounded by $|P|$.*

Furthermore, from Lemma 4.8, Lemma 4.7 , and Theorem 2.3, we can conclude:

Theorem 4.10 *The emptiness problem for polyforest DQAs is decidable. The problem is decidable in time $|M|^{O(n)}$, where n is the number of processes of M.*

References

[1] R. Alur and P. Madhusudan. Visibly pushdown languages. In L. Babai, editor, *STOC*, pages 202–211. ACM, 2004.

[2] R. Alur and P. Madhusudan. Adding nesting structure to words. In O. H. Ibarra and Z. Dang, editors, *Developments in Language Theory*, volume 4036 of *Lecture Notes in Computer Science*, pages 1–13. Springer, 2006.

[3] R. Alur and P. Madhusudan. Adding nesting structure to words. *J. ACM*, 56(3), 2009.

[4] M. F. Atig, B. Bollig, and P. Habermehl. Emptiness of multi-pushdown automata is 2etime-complete. In M. Ito and M. Toyama, editors, *Developments in Language Theory*, volume 5257 of *Lecture Notes in Computer Science*, pages 121–133. Springer, 2008.

[5] L. Breveglieri, A. Cherubini, C. Citrini, and S. Crespi-Reghizzi. Multi-push-down languages and grammars. *Int. J. Found. Comput. Sci.*, 7(3):253–292, 1996.

[6] B. Courcelle. The expression of graph properties and graph transformations in monadic second-order logic. In G. Rozenberg, editor, *Handbook of Graph Grammars*, pages 313–400. World Scientific, 1997.

[7] B. Courcelle, J. Engelfriet, and G. Rozenberg. Handle-rewriting hypergraph grammars. *J. Comput. Syst. Sci.*, 46(2):218–270, 1993.

[8] J. Flum and M. Grohe. *Parameterized Complexity Theory (Texts in Theoretical Computer Science. An EATCS Series)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006.

[9] A. Heußner, J. Leroux, A. Muscholl, and G. Sutre. Reachability analysis of communicating pushdown systems. In C.-H. L. Ong, editor, *FOSSACS*, volume 6014 of *Lecture Notes in Computer Science*, pages 267–281. Springer, 2010.

[10] J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, 1979.

[11] S. La Torre, P. Madhusudan, and G. Parlato. A robust class of context-sensitive languages. In *LICS*, pages 161–170. IEEE Computer Society, 2007.

[12] S. La Torre, P. Madhusudan, and G. Parlato. Context-bounded analysis of concurrent queue systems. In C. R. Ramakrishnan and J. Rehof, editors, *TACAS*, volume 4963 of *Lecture Notes in Computer Science*, pages 299–314. Springer, 2008.

[13] S. La Torre, P. Madhusudan, and G. Parlato. An infinite automaton characterization of double exponential time. In M. Kaminski and S. Martini, editors, *CSL*, volume 5213 of *Lecture Notes in Computer Science*, pages 33–48. Springer, 2008.

[14] S. Qadeer and J. Rehof. Context-bounded model checking of concurrent software. In N. Halbwachs and L. D. Zuck, editors, *TACAS*, volume 3440 of *Lecture Notes in Computer Science*, pages 93–107. Springer, 2005.

[15] D. Seese. The structure of models of decidable monadic theories of graphs. *Ann. Pure Appl. Logic*, 53(2):169–195, 1991.

[16] A. Seth. Global reachability in bounded phase multi-stack pushdown systems. In *To appear in Proc. of Computer Aided Verification (CAV)*, 2010.

[17] W. Thomas. On logics, tilings, and automata. In J. L. Albert, B. Monien, and M. Rodríguez-Artalejo, editors, *ICALP*, volume 510 of *Lecture Notes in Computer Science*, pages 441–454. Springer, 1991.

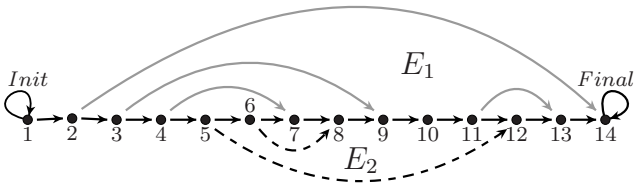

Figure 1: A 2 nested word graph.

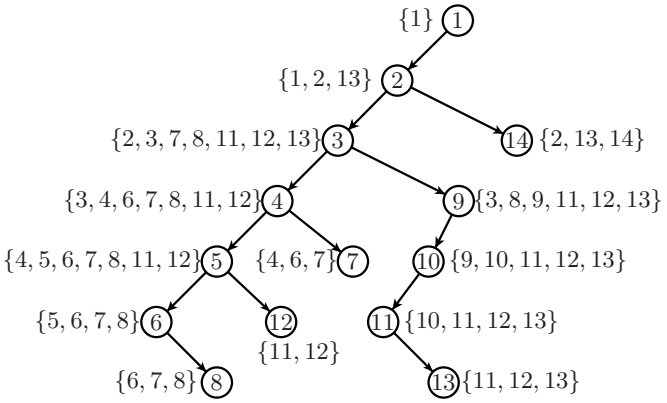

Figure 2: Tree decomposition of the graph illustrated in Figure 1.

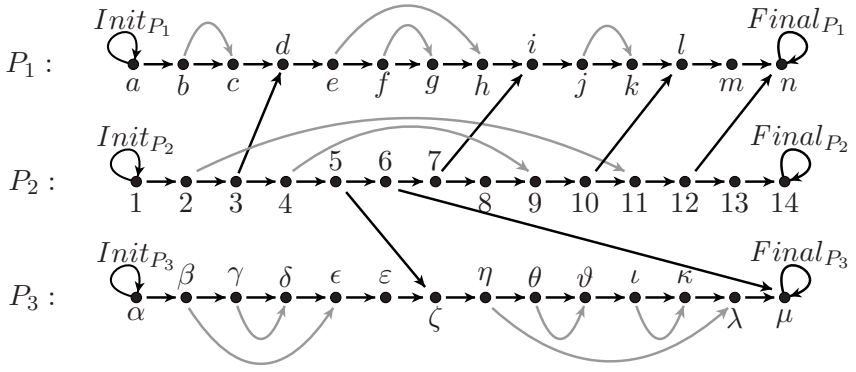

Figure 3: A stack-queue graph.

Appendix

A Semantics and behaviors of multi-stack pushdown automata

In this section we formally define the semantics of multi-stack pushdown automata, and the language describing their behaviors.

A *configuration* of an n-PDA $M = (Q, q_0, \Gamma, \delta, Q_F)$ is a tuple $\langle q, s_1, \ldots, s_n \rangle$ with $q \in Q$ and $s_j \in \Gamma^*$ is the content of stack j, for every $j \in [n]$. Let $C = \langle q, s_1, \ldots, s_n \rangle$ be a configuration of M. Then, C is an *initial* configuration if $q = q_0$ and $s_j = \epsilon$, for every $j \in [n]$. Moreover, C is a *final* configuration whenever $q \in Q_F$. Given two configurations $C = \langle q, s_1, \ldots, s_n \rangle$ and $C' = \langle q', s'_1, \ldots, s'_n \rangle$, there is a transition from C to C' on the action act from the behavior set $B = \{int,\ push_1, \ldots push_n,\ pop_1, \ldots, push_n\}$, denoted $C \xrightarrow{act} C'$, if one of the following holds:

[Push γ onto stack j] $act = push_j$, and there exists γ such that $(q, q', \gamma, j) \in \delta_{push}$, $s'_j = \gamma.s_j$, and $s'_h = s_h$ for every $h \in [n] - j$.

[Pop γ from stack j] $act = pop_j$, and there exists γ such that $(q, \gamma, q', j) \in \delta_{pop}$, $s_j = \gamma.s'_j$, and $s'_h = s_h$ for every $h \in [n] - j$.

[Internal] $act = int$, and $(q, q') \in \delta_{int}$, and $s'_h = s_h$ for each $h \in [n]$.

A *run* of M is a sequence of transitions of M $\rho = C_1 \xrightarrow{act_1} C_2 \ldots \xrightarrow{act_{m-1}} C_m$, where C_1 is initial and C_m is final. For simplicity of exposition, henceforth we assume that in C_m all the stacks are empty.

For each such run ρ of M, we associate the behavior word $beh(\rho) = act_1.act_2 \ldots act_m$, and define the set of behaviors of M as the language $Beh(M) = \{\ beh(\rho) \mid \rho \text{ is a run of } M\ \}$. Note that the behaviors capture the way the automaton handles the stacks, noting the push and pop operations and the stack on which it is performed.

B Tree-width of bounded-context multiply nested word graphs

In this section we show that any k-context multiply nested word graph has a tree-width upper-bounded by $k + 1$.

Lemma B.1 *For any $k \in \mathbb{N}$, the tree-width of any k-context MNW N is at most $k + 1$.*

The proof is simple, and we sketch the main idea. Let us create a tree-decomposition by creating a tree where the root has k subtrees, each subtree corresponding to a stack. For each stack s, we take the contexts that involve the stack s, remove the rest of the events, and build the tree (and the bags) as in the tree-decomposition of a singly nested word (of width at most 2). These trees along with the root, and the bags associated with the nodes, capture all nesting edges and all linear edges, *except the linear edges that cross contexts* (which are at most $k - 1$ in number). Now, for every pair of nodes u and v, where v is the linear successor of u, and where u and v are in different contexts, let us add u to all nodes. Clearly, the bag-sizes increase by at most $k - 1$, and the resulting tree-decomposition captures all edges and is of width at most $k + 1$.

C On the tree-width of bounded-phase and ordered multiply nested words

In this section we give an upper-bound of the tree-width of both bounded-phase and ordered multiply nested word graphs. For a given k and n, the tree-width of any k-phase n-NW is $O(2^k)$, instead the tree-width of any ordered n-NW is $O(n \cdot 2^n)$.

We show such bounds by giving a general technique to upper-bound the width of the tree decomposition $nw\text{-}td(N)$, for any n-NW which is k-phase (Section B.1) or ordered (Section B.2).

Proof strategy: Our proof strategy is the following. First, notice that in any multiply nested word, the tree decomposition we defined has all edges except the *pop-edges*, i.e. edges (u, v) where v is a pop-node for some stack (other linear edges as well as all nesting edges are local in the tree decomposition). We define, first, a notion of an *extension* of a multiply nested word, which is the same as the multiply nested word except that every edge (u, v) where v is a pop-node is replaced by a *path* of nodes which, intuitively, connects u to v by taking a *backward* path along the linear order, all the way up to the push-node v' corresponding to v and then goes on to v. The crucial property of this expansion is that all edges between u and v become local in the tree. This backward path is constructed so that it utilizes nesting edges (of the same kind as the stack v is popping from) in order to reach v'.

This extension of a multiply nested word will be used in both the proofs of bounded phase words as well as ordered multi-stack words. We show that this extension preserves the bounded phase property as well as the ordered-ness property.

The extension of a multiply nested word N then helps us build a new tree-decomposition over the same tree as we need in the theorems; i.e. using a *different* set of bags but over the same tree T deriving from $nw\text{-}td(N)$. We show that this tree-decomposition certainly has width at least as the width of $nw\text{-}td(N)$, and hence establishing that the width of this tree decomposition is bounded by the appropriate bounds for bounded phase multiply nested words and ordered multiply nested words is sufficient to prove our theorems.

We then define a notion of *generator trees* corresponding to *every node* of a multiply nested structure N. Intuitively, the generator tree of a node v consists of the copies of the node v in the extension of N, and a copy (v, h') of v is the child of a copy (v, h'), if (v, h') was created as a relabeling of (v, h) in a backward path that replaced a pop-edge. The generator tree is a technical structure that has certain structural properties (Lemma B.5 and Lemma B.6) that allows us to count the widths of the decompositions of both bounded phase words and ordered multiply nested words.

Proof outline: Throughout the section, every time we refer to N we mean the n-NW $N = (V, Init, Final, L, \{E_j\}_{j\in[n]})$. Moreover, if we refer to an ordering among N nodes, we always intend the linear ordering $<_L$. We also consider an ordering on L edges: if $e_1 = (a, b)$ and $e_2 = (c, d)$ with $e_1, e_2 \in L$, then $e_1 < e_2$ if b occurs before c. Furthermore, T is the tree obtained as $(T, bag) = nw\text{-}td(N)$. If $(u, v) \in E_j$ with $j \in [n]$, we say that u is a *push-j* node, v is a *pop-j* node, and that u and v are *matched*. Moreover, an L edge (u, v) is called a *pop-j* edge, if v is a pop-j node.

For any N, we define an n-NW $N' = (V', Init', Final', L', \{E'_j\}_{j\in[n]})$, called the *extension* of N, as follows. Intuitively, N' is obtained from N by replacing all the pop edges with a sequence of nodes. More precisely, consider a pop-j edge (u, v) and suppose that all the pop edges before (u, v) have already been replaced with paths to create a nested word N'. Then, the pop edge (u, v) is replaced with the "back-path" of N' starting from u and ending with the push node u' that matches v. The back-path is built in the following manner. Suppose we have reached a node b. Now, if b is a pop-j node — notice that v is also a pop-j node — then the next node in the back-path is a where a is the push-j node matched to b ($(a, b) \in E'_j$). (In this

way we get closer to u', which comes before a, by avoiding all nodes between a and b.) Otherwise, the next node in the path will be the L' predecessor of b. In other words, the back-path from u to u' is formed by taking linear predecessors at each state, except taking nesting edges for the stack j. Obviously all the nodes in back-paths will be renamed so that they will be unique in N'.

Now we formally define the extension of a multiply nested word N, $Ext(N)$. We do this by defining a function $expand$ that takes the *first* pop-edge in a nested word, and replaces it by a back-path. We will first start with the nested word N, with renamed vertices. Then, we will apply $expand$ to it repeatedly till all pop-edges are replaced (and we reach a fixed-point). This fixed-point will be the extension of N. First, let us define back-paths formally.

Back-paths and extensions:
Let N be a n-NW and let (u, v) be a pop-edge (i.e. v is a pop-node and u is the linear predecessor of v). Let $(v', v) \in E_j$ ($j \in [n]$). Then $BackPath_N(v)$ is the unique node sequence $v_1 \dots v_t$ such that

- $v_1 = u$ and $v_t = v'$, and
- For every $i \in [t-1]$, if v_i is a pop-j node, then v_{i+1} is the corresponding push-node, i.e. the node such that $(v_{i+1}, v_i) \in E_j$.
  Otherwise v_{i+1} is the linear predecessor of v (i.e. the node such that $(v_{i+1}, v_i) \in L$).

We now define the extension of a multiple nested word, using a systematic replacement of every pop-edge (u, v) by a linearly ordered sequence of nodes formed by a back-path from u to the push-node v' corresponding to v. Moreover, in the linearly ordered sequence that replaces the pop-edge, no node will have nesting edges incident on it. We will perform this surgery on all pop-edges, going from the left-most one to the right-most; this is important as back-paths for a pop-edge may utilize the extensions of pop-edges that occur to the left of it.

Let us fix a n-NW $N = (V, Init, Final, L, \{E_j\}_{j\in[n]})$. The extension of N will have vertices of the form (v, i) where $v \in V$ and $i \in \mathbb{N}$.

Let N_0 be the same as nested word N, except that each vertex $v \in V$ gets renamed to $(v, 1)$. In other words, $N_0 = (V \times \{1\}, Init_0, Final_0, L_0, \{E_j^0\}_{j\in[n]})$, where the various edges in N_0 are appropriately defined.

We now construct N_{i+1} from N_i using the following algorithm. Let $N_i = (X, Init', Final', L, \{E_j'\}_{j\in[n]})$, where $X \subseteq V \times \mathbb{N}$. Let $((u, 1), (v, 1))$ be the first pop-edge of its kind (i.e. with indices 1) in N_i according to the linear ordering L (if no such pop-edge exists, then we set $N_{i+1} = N_i$, and reach a fixed-point). Then $N_{i+1} = (X', Init', Final', L', \{E_j'\}_{j\in[n]})$ is defined as follows (note that the initial, final, and nesting edges do not change).

Let the back-path from u_1 be $BackPath_N(\langle u, 1\rangle) = \langle z_1, h_1\rangle \dots \langle z_t, h_t\rangle$. Note that a node $\langle x, i\rangle$ occurs at most once in the back-path. Let us now relabel this path so that the nodes $\langle x, i\rangle$ get renamed to some $\langle x, j\rangle$ so that they are not in X and do not get repeated in the back-path:

- $relabel_X(\epsilon) = \epsilon$
- $relabel_X(w\langle x, i\rangle) = relabel_X(w)\langle x, j\rangle$ where j is the least number such that $\langle x, j\rangle \notin X$ and does not occur in $relabel_X(w)$.

Let $relabel_X(BackPath_{N'}(\langle u, 1\rangle)) = \langle z_1, h_1'\rangle \dots \langle z_t, h_t'\rangle$. Then, $X' = X \cup \{\langle z_i, h_i'\rangle \mid i \in [t]\}$ and the set L' is:

$$L' = L \setminus \{(\langle u, 1\rangle, \langle v, 1\rangle)\} \cup \{(\langle z_i, h_i'\rangle, \langle z_{i+1}, h_{i+1}'\rangle) \mid i \in [t]\} \cup \{(\langle u, 1\rangle, \langle z_1, h_1'\rangle), (\langle z_t, h_t'\rangle, \langle v, 1\rangle)\}$$

Intuitively, we remove the linear edge from $\langle u, 1\rangle$ to $\langle v, 1\rangle$) and replace it with the backward path from $\langle u, 1\rangle$, appropriately renamed.

We apply the above algorithm to systematically replace pop-edges by a linearly ordered set of nodes, left to right, till we reach a fixed-point, where there are no pop-edges of the form $(\langle u, 1\rangle, \langle v, 1\rangle)$. The final multiply nested word will be the *extension* of N.

Notice that, N' is the same as N except that pop edges of N are replaced by nodes that are neither the target nor the source of any nesting edges. Therefore, if N is a k-phase MNW then also N' is, and if N is an ordered n-NW then so is N':

Lemma C.1 *Let N' be the extension of an n-NW N. Then, (1) N' is k-phase iff N is k-phase, (2) N' is ordered iff N is ordered.*

It is easy to prove that if $(\langle a, i\rangle, \langle b, j\rangle)$ is an edge in N', that is $(\langle a, i\rangle, \langle b, j\rangle) \in (L' \cup \bigcup_{h\in[n]} E'_h))$, then a and b are connected by an edge in T, which means that either a is the parent of b or vice-versa. By using N' we define a new tree decomposition of N whose underlying tree is T.

Let $T = (V, F)$. We define a map $bag' : V \to 2^V$ as follows. Map bag' associates the minimal set of vertices to each node of T according to the following rules:

1. $v \in bag'(v)$, for all $v \in V$.
2. if u is the parent of v in T, then $u \in bag'(v)$, for every $v \in V$.
3. if (u, v) is a pop edge of N, and $BackPath_{N'}(\langle u, 1\rangle) = \langle u_1, h_1\rangle \dots \langle u_t, h_t\rangle$, then $u \in bag'(u_i)$, for every $i \in [t]$.

Notice that the first and second condition defining the map bag (see Definition 3.6) and the first and second condition in the definition of bag' are the same. They only differ in the third: if u' is such that $(u', v) \in E_j$, then condition three of Definition 3.6 says that u is added to $bag(z)$ for all nodes z lying along the unique shortest path in T between u and u'. Similarly the third condition of the definition above adds u to the bag' of all the T nodes along a path in T from u to u' which may not be the shortest. However, that path has to pass trough all the nodes of the shortest path between u' and u. Thus, (T, bag') is a tree decomposition of N, and more importantly for us $bag(z) \subseteq bag'(z)$, for every node z of T. Therefore, we can upper-bound the size of $bag(u)$ by considering the size of $bag'(u)$ for every $u \in V$, as stated in the next lemma.

Lemma C.2 *Let N be an n-nested word, and $\mathcal{T} = (T, bag) = nw\text{-}td(N)$. Then, $\mathcal{T}' = (T, bag')$ is a tree decompositions of N where $width(\mathcal{T}) \leq width(\mathcal{T}')$. Furthermore, for every $v \in V$, $|bag'(v)| \leq d_v + 1$, where $d_v = |\{\, \langle v, h\rangle \in V' \,|\, h \in \mathbb{N} \,\}|$.*

Generator Trees: A convenient way to calculate d_v (in the above lemma) is to represent the set of N' nodes $\{\langle v, h\rangle \in V' \,|\, h \in \mathbb{N}\}$ as a tree, for every $v \in V$. Let $\langle v, h\rangle$, with $h > 1$, be a node of N', and let $\langle u, 1\rangle$ be the greatest push node of N' that occurs before $\langle v, h\rangle$. Intuitively, $\langle v, h\rangle$ is one of the node of the path between that have replaced the pop edge (u, v) of N. By definition of N', $\langle v, h\rangle$ is generated because there is another node $\langle v, h'\rangle$ with $h' < h$ in $BackPath(\langle u, 1\rangle)$. We call $\langle v, h'\rangle$ the *generator* of $\langle v, h\rangle$. Note that for every node $\langle v, h\rangle$ with $h > 1$ there is a unique generator of it (though the vice-versa does not hold).

Definition C.3 (GENERATOR TREES**)** *Let N' be the extension of an n-nested word N, and let V be the set of nodes of N. For every $v \in V$, we define a tree T_v as follows:*

- *$\langle v, 1\rangle$ is the root of T_v.*
- *if $\langle v, h'\rangle$ is the generator of $\langle v, h\rangle$ then $\langle v, h\rangle$ is a child of $\langle v, h'\rangle$.*

For every $v \in V$, the tree T_v is called the generator tree *of v.*

Observe that, for a given N node v, all the nodes (v, h) in N' are also nodes of T_v, thus the value d_v corresponds to the number of nodes of T_v,

We can also associate a *stack* to every node of generator tree, except the root. If a node $\langle v, 1\rangle$ is the first pop node after $\langle v, h\rangle$ (where $h > 1$), and if v is a pop node of stack j, then we say that j is the *stack* of $\langle v, h\rangle$. Intuitively, the stack associated with (v, h) is the stack whose popping led to a back-path that created (v, h).

In the following we give some properties of generator trees that will be instantiate later for the case in which N is bounded-phase and ordered. Intuitively, fix a stack j; then, any node in a multiply nested word can be touched only once on a backward path that is caused by a pop of stack j, except that when the node is a push onto stack j, in which case it may be touched twice. This is true because the backward path caused by a pop to stack j takes nesting edges of stack j as much as possible, hence skipping the nodes between the nesting edges it takes.

The first lemma states that if v is a push onto stack j, the root of the generator tree of v, namely $(v, 1)$, has at most $n + 1$ children— at most two of these children may be of stack j, and all the other children must be of distinct stacks.

Lemma C.4 *If $v \in V$ is a push-j node then the root $\langle v, 1\rangle$ of T_v has at most two children of stack j. Moreover, for every $j' \neq j$, $\langle v, 1\rangle$ has at most one child of stack j'.*

Proof By contradiction suppose that $(v, 1)$ has at least three children of stack j. Since a back-path goes always backward it contains distinct nodes. Therefore there must exist three pop-j edges in N, say $e_1 = (u_1, v_1), e_2 = (u_2, v_2), e_3 = (u_3, v_3)$, such that $\langle v, 1\rangle$ is contained in $BackPath_{N'}(\langle u_i, 1\rangle)$ for all $i \in [3]$. Suppose that e_1, e_2 and e_3, in the order, are the first three pop edges of N having the above property. It is easy to see that $\langle v_1, 1\rangle$ is the matching pop of $\langle v, 1\rangle$. Now, $BackPath_{N'}(\langle u_2, 1\rangle)$ to reaches $\langle v, 1\rangle$ must pass through $\langle v_1, 1\rangle$ (a back-path always goes backward and since the E'_j relation is nested a back-path can never jump in between $\langle v, 1\rangle$ and $\langle v_1, 1\rangle$). Thus, when $BackPath_{N'}(\langle u_2, 1\rangle)$ reaches $\langle v_1, 1\rangle$, it goes directly to $\langle v, 1\rangle$. This entails that the matching push of $\langle v_2, 1\rangle$ occurs before $\langle v, 1\rangle$. Now, $BackPath_{N'}(\langle u_3, 1\rangle)$ must pass through $\langle v_2, 1\rangle$ to reach $\langle v, 1\rangle$. But, $\langle v_2, 1\rangle$ is a pop-j node and thus the back-path jumps directly to the matching push of $\langle v_2, 1\rangle$, which comes before $\langle v, 1\rangle$. Since a back-path goes always backward, $\langle v, 1\rangle$ can never be reached by $BackPath_{N'}(\langle u_3, 1\rangle)$. This is a contradiction.

In similar way we prove that, if $j' \neq j$ then $\langle v, 1\rangle$ has at most one child of stack j'. By contradiction, let $e_1 = (u_1, v_1)$ and $e_2 = (u_2, v_2)$ be the first two pop-j' edges of N such that $BackPath_{N'}(\langle u_1, 1\rangle)$ and $BackPath_{N'}(\langle u_2, 1\rangle)$ contain $\langle v, 1\rangle$. If $BackPath_{N'}(\langle u_1, 1\rangle)$ passes through $\langle v, 1\rangle$ means that the push-j node matched by the $\langle v_1, 1\rangle$ must occur before $\langle v, 1\rangle$. Now $BackPath_{N'}(\langle u_2, 1\rangle)$ must pass through $\langle v_1, 1\rangle$ and hence jumps directly to the matched push-j node matched with $\langle v_1, 1\rangle$. Since such a node comes before $\langle v, 1\rangle$ and back-paths never go forward we have that $\langle v, 1\rangle$ cannot be reached by $BackPath_{N'}(\langle u_2, 1\rangle)$.

□

The second property we need is that for any node v, any non-root node in the generator tree of v has children whose stacks are distinct from each other. Moreover, if v is not a push, then the root also has children whose stacks are all distinct from each other.

Lemma C.5 *Let $\langle z, h\rangle \in N'$. Then, if $h > 1$ or z is not a push node of N, then for every $j \in [n]$, the node $\langle z, h\rangle$ has at most one child of stack j in T_z.*

Proof If $h > 1$ then $\langle z, h\rangle$ must be a node of a path that has replaced a pop edge, say (u, v) of N. Suppose that $e_1 = (u_1, v_1)$ and $e_2 = (u_2, v_2)$ are the first two pop-j edges (in the order) of N such that $BackPath_{N'}(\langle u_1, 1\rangle)$ and $BackPath_{N'}(\langle u_2, 1\rangle)$ contain $\langle z, h\rangle$. Thus, $\langle z, h\rangle <_{L'} \langle u_1, 1\rangle <_{L'} \langle v_1, 1\rangle <_{L'} \langle u_2, 1\rangle$. Since $BackPath_{N'}(\langle u_1, 1\rangle)$ passes through $\langle z, h\rangle$ implies that the push-j node matched by the $\langle v_1, 1\rangle$ occurs before $\langle z, h\rangle$. Now $BackPath_{N'}(\langle u_2, 1\rangle)$ has to pass through $\langle v_1, 1\rangle$, which is a pop-j node, and hence jumps directly to the push-j node matched to $\langle v_1, 1\rangle$. Such a node appears before $\langle z, h\rangle$ and since back-paths only go backward we have that $\langle v, 1\rangle$ is never reached by $BackPath_{N'}(\langle u_2, 1\rangle)$ which contradicts the hypotheses.

The other case in which $\langle z, 1\rangle$ is not a push node is similar to the case above and we do not give it here. □

C.1 Tree-width of bounded-phase multiply nested word graphs

In this section we show that the tree-width of any k-phase MNW N is $O(2^k)$.

From Lemma C.1, the extension N' of N is also a k-phase n-NW. Thus, we define $phase_{N'}$ to be the map that associates to every node $\langle v, h\rangle$ of N' its phase number.

The next lemma, which is a refinement of Lemma C.4, says that for any push-node v, the phase numbers of the children of the root of the generator tree of v are not less than that of the root, and further, all phase numbers of the children of the root are distinct from each other, save for one child. This bounds the number of children of the root to $k - j + 2$, if the root has phase j.

Lemma C.6 *For every push node $v \in V$, the phase of the children of the root $\langle v, 1\rangle$ of T_v is greater or equal to the phase of $\langle v, 1\rangle$. Moreover, except for one child of $\langle v, 1\rangle$, all the other children have different phase number.*

Proof If $\langle v, h\rangle$ is a child of $\langle v, 1\rangle$, then $\langle v, 1\rangle <_{L'} \langle v, h\rangle$, and hence $phase_{N'}(\langle v, 1\rangle) \leq phase_{N'}(\langle v, h\rangle)$. Now, if the stack number of $\langle v, h\rangle$ is different from the stack number of $\langle v, 1\rangle$ then $phase_{N'}(\langle v, 1\rangle) < phase_{N'}(\langle v, h\rangle)$. Moreover, if $\langle v, h\rangle$ and $\langle v, h'\rangle$ are two children of $\langle v, 1\rangle$ with different stack number then $phase_{N'}(\langle v, h\rangle) \neq phase_{N'}(\langle v, h'\rangle)$. Thus, from Lemma C.4 we can conclude the proof. □

By using a similar argument of the previous proof, and Lemma C.5, we can show the following lemma, which says that for any v, the children of a non-root node (v, h) in the generator tree for v have distinct phases and have phases greater than the phase of (v, h). Moreover, this is also true for the root $(v, 1)$ provided v is not a push-node.

Lemma C.7 *Let $\langle v, h\rangle \in N'$. Then, if $h > 1$ or v is not a push node of N, then for every child $\langle v, h'\rangle$ of $\langle v, h\rangle$ in T_v, $phase_{N'}(\langle v, h\rangle) < phase_{N'}(\langle v, h'\rangle)$. Moreover, for every phase number $p > phase_{N'}(\langle v, h\rangle)$, there is at most one child $\langle v, h'\rangle$ of $\langle v, h\rangle$ such that $phase_{N'}(\langle v, h'\rangle) = p$.*

By using the previous lemma we can upper-bound the number of nodes of the sub-tree of T_v rooted in any internal node of T_v, for every node v of N. Let $f : [k] \to \mathbb{N}$ defined as: $f(i) = 1 + \sum_{j=i+1}^{k} f(j)$ for every $i \in [k-1]$, and $f(k) = 1$. By a simple calculation it is easy to prove that $f(i) = 2^{k-i}$. Thus, we can upper-bound the number of nodes of any subtree of T_v rooted in an internal node $\langle v, h\rangle$ with $f(phase_{N'}(\langle v, h\rangle))$.

Now by instantiating Lemma C.6, have that

$$d_v \leq 1 + f(1) + \sum_{i=1}^{k} f(i) = 2^k + 2^{k-1},$$

and by Lemma C.2 follows that the width of the tree decomposition $nw\text{-}td(N)$ of N is at most $2^k + 2^{k-1} + 1$.

Theorem C.8 *The tree-width of any k-phase MNW is at most 2^{k+1} (where $k \geq 2$).*

C.2 Tree-width of ordered multiply nested word graph

In this section we show that the tree-width of any ordered n-nested words N is $O(n \cdot 2^{n-1})$. As in the previous section, we prove such a result by upper-bounding the number of nodes of each tree T_v, for every node v of N.

In the following we instantiate Lemma C.5 for ordered multiply nested words. We show that for any internal node (v, h) of the generator tree of a node v, the stacks of the children of v are strictly greater than that of v. The reason why the stack of a child of (v, h) cannot be lower than that of v is because of the ordered-ness of the stack accesses— if the back-path of a pop of stack j' leads through a pop of stack j, then we must have that $j \leq j'$ (the reason why it cannot $j \neq j'$ is also argued below). Hence, the depth of the tree gets bounded by the number of stacks, n, and each non-root node has at most $n - 1$ children.

Lemma C.9 *If $\langle v, h\rangle \in N'$ is a stack j node with $h > 0$, then (1) the stack j' for any child of the node $\langle v, h\rangle$ is such that $j' > j$, and (2) the stacks for the children of the node $\langle v, h\rangle$ are all distinct.*

Proof Case (2) follows from Lemma C.4. Case (1) is proved by contradiction and we distinguish two cases, one when $j' < j$ and the other one for $j' = j$. Let $\langle v, h'\rangle$ be a child of $\langle v, h\rangle$, and suppose that $\langle v, h'\rangle$ is a stack j' node. Since $h, h' > 1$, $\langle v, h\rangle$ and $\langle v, h'\rangle$ are both lying on a two different paths that replace two different pop edges of N, say $e_1 = (u_1, v_1)$ and $e_2 = (u_2, v_2)$. Thus, we have that $\langle v, h\rangle <_{L'} \langle v_1, 1\rangle <_{L'} \langle u_2, 1\rangle <_{L'} \langle v, h'\rangle$. The fact that $BackPath_{N'}(\langle u_2, 1\rangle)$ has to visit $\langle v, h\rangle$ to reach the matching push-j' node of $\langle v_2, 1\rangle$ means that it occurs before the pop-j $\langle v_1, 1\rangle$.

Now if $j' < j$, it means that there is pop-j node that comes after a push-j' node that has not matched yet. Since $j' < j$, this contradicts the ordered-ness property of N' and hence N. Instead, if $j' = j$ then $BackPath_{N'}(\langle u_2, 1\rangle)$ will never visit $\langle v, h\rangle$ because between $\langle v, h\rangle$ and $\langle u_2, 1\rangle$ there is a pop-j node whose matching pop occurs before $\langle v, h\rangle$. □

For every $i \in [n]$, let us define the map $f : [n] \to \mathbb{N}$ as $f(i) = 1 + \sum_{j=i+1}^{k} f(j)$ if $i \in [n-1]$ and $f(n) = 1$. Notice that $f(i) = 2^{n-i}$. From Lemma C.9, It is easy see that $f(i)$ upper-bounds the number of nodes of any T_v subtree rooted in one of its internal node which is a stack i node.

Thus, from Lemma C.4 we can conclude that the following upper-bounds the number of nodes of any tree T_v.

$$1 + (n+1)f(1) = 1 + (n+1) \cdot 2^{n-1}.$$

Now from Lemma C.2 we can conclude with the main theorem of the section.

Theorem C.10 *The tree-width of any ordered n-NW is $O(n \cdot 2^{n-1})$.*

D Distributed queue automata with stacks

In this section we define the syntax and the semantics of distributed queue automata with stacks.

Definition D.1 (DISTRIBUTED QUEUE AUTOMATA WITH STACKS**)** *A* distributed queue automaton with stacks *(DQSA) is a tuple* $M = (P, Q, \Pi, \Gamma, Sender, Receiver, \{A_p\}_{p\in P})$ *where* P *is a finite set of process names,* Q *is a finite set of queues,* Π *is a finite message alphabet,* Γ *is a finite stack alphabet, and* $Sender\colon Q \to P$ *and* $Receiver\colon Q \to P$ *are two maps that assign a unique* sender *process and* receiver *process for each queue, respectively. For every process* $p \in P$, $A_p = (S_p, s_0^p, F_p, \delta_p)$ *is the machine at site* p*, where* S_p *is a finite set of states,* $s_0^p \in S_p$ *is the initial state,* $F_p \subseteq S_p$ *is the set of final states, and* $\delta_p = \langle \delta_{int}^p, \delta_{send}^p, \delta_{recv}^p, \delta_{push}^p, \delta_{pop}^p \rangle$ *where*

- $\delta_{send}^p \subseteq (S_p \times Q_{send}^p \times \Pi \times S_p)$ *is the set of send moves, where* $Q_{send}^p = \{\ q \in Q \mid Sender(q) = p\ \}$*;*
- $\delta_{recv}^p \subseteq (S_p \times Q_{recv}^p \times \Pi \times S_p)$ *is the set of receive moves, where* $Q_{recv}^p = \{\ q \in Q \mid Receiver(q) = p\ \}$*;*
- $\delta_{push} \subseteq (S_p \times S_p \times \Gamma)$ *is the set of push moves;*
- $\delta_{pop} \subseteq (S_p \times \Gamma \times S_p)$ *is the set of pop moves;*
- $\delta_{int}^p \subseteq (S_p \times S_p)$ *is the set of internal moves.*

For the rest of the section we fix $M = (P, Q, \Pi, \Gamma, Sender, Receiver, \{A_p\}_{p\in P})$ to be a DQSA, where $A_p = (S_p, s_0^p, F_p, \delta_p)$ for every $p \in P$.

The semantics of DQSAs is as follows.

A *configuration* of a DQSA M is a tuple $\langle \{s_p\}_{p\in P}, \{\gamma_p\}_{p\in P}, \{\mu_q\}_{q\in Q} \rangle$ where for each $p \in P$, $s_p \in S_p$ and $\gamma_p \in \Gamma^*$ are the state and the stack content of process p respectively, and for each queue $q \in Q$, $\mu_q \in \Pi^*$ is the content of q. A configuration $C = \langle \{s_p\}_{p\in P}, \{\gamma_p\}_{p\in P}, \{\mu_q\}_{q\in Q} \rangle$ of M is an initial configuration if $s_p = s_0^p$ and $\gamma_p = \epsilon$ for each $p \in P$, and $\mu_q = \epsilon$, for each queue $q \in Q$. C is a *final* configuration if $s_p \in F_p$, for *every* process $p \in P$, and further all queues are empty, i.e. $\mu_q = \epsilon$, for each $q \in Q$.

Let the *actions of process* p be $B_p = \{int_p, push_p, pop_p\} \cup (\bigcup_{q\in Q}\{send_{(p,q)}\}) \cup (\bigcup_{q\in Q}\{recv_{(p,q)}\})$, and $B = \bigcup_{p\in P} B_p$ be the alphabet of all actions. For any two configurations $C = \langle \{s_p\}_{p\in P}, \{\gamma_p\}_{p\in P}, \{\mu_q\}_{q\in Q} \rangle$ and $C' = \langle \{s'_p\}_{p\in P}, \{\gamma'_p\}_{p\in P}, \{\mu'_q\}_{q\in Q} \rangle$, $C \xrightarrow{act} C'$, if $act \in B$ and one of the following holds:

[Send] $act = send_{(p,q)}$, and there is a move $(s_p, q, m, s'_p) \in \delta_{send}^p$ such that

- for each $\hat{p} \neq p$, $s'_{\hat{p}} = s_{\hat{p}}$,
- $\mu'_q = m.\mu_q$, and for each $\hat{q} \neq q$, $\mu'_{\hat{q}} = \mu_{\hat{q}}$.
- for each $\hat{p}$, $\gamma'_{\hat{p}} = \gamma_{\hat{p}}$.

[Receive] $act = recv_{(p,q)}$, and there is a move $(s_p, q, m, s'_p) \in \delta_{recv}^p$ such that

- for each $\hat{p} \neq p$, $s'_{\hat{p}} = s_{\hat{p}}$,
- $\mu_q = \mu'_q.m$, and for each $\hat{q} \neq q$, $\mu'_{\hat{q}} = \mu_{\hat{q}}$.
- for each $\hat{p}$, $\gamma'_{\hat{p}} = \gamma_{\hat{p}}$.

[Push] $act = push_p$, and there is a move $(s_p, s'_p, a) \in \delta_{push}^p$ such that

- for each $\hat{p} \neq p$, $s'_{\hat{p}} = s_{\hat{p}}$,
- for each $\hat{q}$, $\mu'_{\hat{q}} = \mu_{\hat{q}}$.
- $\gamma'_p = a.\gamma_p$, and for each $\hat{p} \neq p$, $\gamma'_{\hat{p}} = \gamma_{\hat{p}}$.

[Pop] $act = pop_p$, and there is a move $(s_p, a, s'_p) \in \delta^p_{pop}$ such that

- for each $\hat{p} \neq p$, $s'_{\hat{p}} = s_{\hat{p}}$,
- for each $\hat{q}$, $\mu'_{\hat{q}} = \mu_{\hat{q}}$.
- $a.\gamma'_p = \gamma_p$, and for each $\hat{p} \neq p$, $\gamma'_{\hat{p}} = \gamma_{\hat{p}}$.

[Internal] $act = int_p$, and there is a move $(s_p, s'_p) \in \delta^p_{int}$ such that

- for each $\hat{p} \neq p$, $s'_{\hat{p}} = s_{\hat{p}}$,
- for each $\hat{q}$, $\mu'_{\hat{q}} = \mu_{\hat{q}}$.
- for each $\hat{p}$, $\gamma'_{\hat{p}} = \gamma_{\hat{p}}$.

Let $w = act_1 act_2 \dots act_{m-1} \in B^*$. A *run* of M on w is a sequence $\rho = C_1 \xrightarrow{act_1} C_2 \dots \xrightarrow{act_{m-1}} C_m$, where C_1 is initial and C_m is final.

E Tree-width of well-queuing stack-queue graphs with tree architectures

Here we show that any well-queuing stack-queue graph over a pair (P, Q) whose underlying architecture is a directed forest has tree-width at $O(n)$, where $n = |P|$. The main argument is to first give a *nested word decomposition* (which is a generalization of tree decompositions) for a well-queuing stack-queue graph and then provide a tree-decomposition of the nested word and show that the decompositions compose to give a bounded-width tree decomposition of original stack-queue graph.

We start giving a more relaxed notion of decomposition of a graph onto another graph.

Definition E.1 (GRAPH DECOMPOSITION**)** *A* graph-decomposition *of a graph $G = (V, E)$ over a graph $H = (V_H, E_H)$ is a pair (H, bag) where $bag : V_H \to 2^V$ is the function that associates the minimal sets that satisfies the following:*

- *For every $v \in V$, there is a node $v' \in V_H$ such that $v \in bag(v')$.*
- *For every edge $(u, v) \in E$, there is an edge $(u', v') \in E_H$ such that $u \in bag(u')$ and $v \in bag(v')$.*
- *If $u \in bag(z')$ and $u \in bag(z'')$, for nodes $z', z'' \in V_H$, then there is a path $z_1 \dots, z_t$ in H connecting z' to z'' such that for all $i \in [t]$, $u \in bag(z_i)$.*

The width *of a graph decomposition is the size of the largest bag in it; i.e. $max_{z \in V_H}\{|bag(z)|\}$.*

Note that the above definition is slightly more general than the definition of a tree-decomposition as it demands that every edge in G be represented by *adjacent* bags in H (as opposed to the same bag in H). However, this is only a mild difference (for instance, one can always create an intermediate node on the edge to capture a bag that has both the nodes of the edge in G). The above definition will be more convenient.

Lemma E.2 *Let (H, bag_H) be a graph-decomposition of a graph G with width g. If H has a tree-decomposition of width h, then G has a tree-decomposition of width $O(g \cdot h)$.*

Proof Let $\mathcal{H} = (H, bag_H)$, and $\mathcal{T} = (T, bag_T)$ be a tree-decomposition of H of width h. Let $\mathcal{T}' = (T, bag_{T'})$, where $bag_{T'}(a) = \bigcup_{v \in bag_T(a)}(bag_H(v))$, for every node a of T. It is easy to see that $bag_{T'}(a) \leq g \cdot (h+1) - 1$. Thus, to conclude the proof we only need to show that $\mathcal{T}'$ is a tree-decomposition of G.

We first prove that for each node v_G of G there is a node v_T of T such that $v_G \in bag_{T'}(v_T)$. (H, bag_H) is a graph-decomposition of G, therefore H must have a node v_H such that $v_G \in bag_H(v_H)$. On the other hand, $\mathcal{T}$ is a tree-decomposition of H, thus T has a node v_T such that $v_H \in bag_T(v_T)$. Hence $v_G \in bag_{T'}(v_T)$.

Now we show that if (u_G, v_G) is an edge of G, then there is a node z_T of T such that $u_G, v_G \in bag_{T'}(z_T)$. H contains two adjacent nodes u_H and v_H such that $u_G \in bag_H(u_H)$, and $v_G \in bag_H(v_H)$. T must have a node z_T such that $u_H, v_H \in bag_T(z_T)$. Since $u_G \in bag_H(u_H)$ and $v_G \in bag_H(v_H)$, we have that $u_G, v_G \in bag_{T'}(z_T)$.

Suppose there are two nodes u_T and v_T of T such that $z_G \in (bag_{T'}(u_T) \cap bag_{T'}(v_T)$. By the definition of $\mathcal{T}'$, there must exist two nodes $u_H \in bag_T(u_T)$ and $v_H \in bag_T(v_T)$, such that $z_G \in (bag_H(u_H) \cap bag_H(v_H))$. Since $\mathcal{H}$ is a graph-decomposition of G, there must exist in H a path $z_1 \dots z_t$ form u_H to v_H such that $z_G \in bag_H(z_i)$ for all $i \in [t]$. Now we show that there is also a path in T from u_T to v_T such that for every node u along this path we have that $(bag_T(u) \cap \{z_1 \dots z_t\}) \neq \emptyset$, and hence $z \in bag_{T'}(u)$. This path is build as follows. For every edge (z_i, z_{i+1}) of H there must be a node u_i of T such $z_i, z_{i+1} \in bag_T(u_i)$. Since $bag_T(u_i)$ and $bag_T(u_{i+1})$ both contain u_{i+1}, then there must be a path π_i in T such that for all nodes x of π_i, $u_{i+1} \in bag_T(x)$. Thus, $\rho' u_1 \pi_1 u_2 \pi_2 \dots u_{t-1} \rho''$ is the desired path, where ρ' is the path from u_T to u_1 and ρ'' is the path from u_{t-1} to v_T.

□

Stack-queue graphs: Recall the formal definition of stack-queue graphs (Definition 4.1), and recall that a stack-queue graph over the name sets P and Q, is a tuple of the form
$SQG = (\ \{(V_p, Init_p, Final_p, L_p, E_p\ )\}_{p \in P},\ \{E_q\}_{q \in Q}\ )$

If $(u, v) \in E_p$ then u is said a *push* node and v is called a *pop* node; moreover, u is said *matched* by v. If $(u, v) \in E_q$ then u is said *send* node and v is called a *receive* node. If $(u, v) \in L_p$ then v is called the *successor* of u, and u is the *predecessor* of v. Finally, if $(v, v) = Init_p$ then v is called the *init* node of process p, as well as if $(v, v) = Final_p$ then v is said the *final* node of process p.

The *architecture* of a stack-queue graph SQG is the directed graph that describes the way its processes communicate trough queues: $Arch(SQG) = (\ P,\ \{\ (p, p') \mid E_q \subseteq E_p \times E_{p'}, E_q \neq \emptyset\ \}\ )$.

A stack-queue graph SQG is *well-queuing* if for every receive node $v \in V_p$, all the push nodes $u \in V_p$ that precede v (that is, $u <_{L_p} v$) are matched by a pop nodes occurring before v (i.e. there exists $v' \in V_p$ where $(u, v') \in E_j$ and $v' <_{L_p} v$).

In the rest of the section, whenever we refer to SQG, we mean the stack-queue graph $SQG = (\ \{(V_p, Init_p, Final_p, L_p, E_p\ )\}_{p \in P},\ \{E_q\}_{q \in Q}\ )$, which is well-queuing and with a forest architecture.

In the following we show that when SQG is well-queuing and whose architecture is a forest, then there is a nested-word decomposition of SQG of width $O(n)$, where $n = |P|$. Before we start giving a nested-word decomposition of SQG, we define a "1-stack-queue sequence" of SQG.

First, note that since the architecture is a forest, the union of the linear orderings and the ordering that orders a send event before its matching receive event, is a partial order, which we call the *causal order*. We are now interested in particular linear sequentializations of this causal order.

A well-queuing stack-queue graph naturally captures the behavior of a DQAS in which any process is allowed to dequeue only when its stack is empty. It turns out that when the architecture of a DQAS is a forest then all its behaviors can be rescheduled to a 1-stack-queue sequence that is a sequentialization of

the causal order, and further, one where there is at most one message in the entire network of queues, as well as satisfies the condition that nesting edges do not intersect— i.e. there are no four events e_1, e_2, f_1, f_2, scheduled in that order, with (e_1, f_1) a nested edge of one stack and (e_2, f_2) a nested edge of (another) stack. Intuitively, this will allow us to capture *all* the nesting edges together as one nested edge (which will help us decompose the graph onto a nested word).

We now informally describe how to construct one of these *1-stack-queue sequences*. The directed forest architecture naturally defines a partial-order: if there is a queue from p to p', then p is ordered before p', and we take the reflexive transitive closure of this relation. Let us now fix a linear order extending this partial order.

The key idea now to construct a 1-stack-queue sequence is to schedule the next event, at any point, by choosing it to be the *last* process that can do an event that is minimal amongst the pending events. Notice that in this ordering of events, when p sends a message, all its children (and in fact its descendents) must be "stuck" and waiting for a message from their incoming queues (since otherwise they can be scheduled), and hence their *local stacks* must be empty (because of the well-queuing assumption). Hence, when a process pushes onto its local stack and later pops from it, the interim push-events (involving other stacks) must get matched by pop-events in the interim itself. This ensures that the union of the nesting edge relations of all stacks on this sequentialization is a single nesting edge relation.

We have hence shown the following:

Lemma E.3 *Let $SQG = (\,\{(V_p, Init_p, Final_p, L_p, E_p)\}_{p\in P},\ \{E_q\}_{q\in Q}\,)$ be a well-queuing stack-queue graph where $Arch(SQG)$ is a directed forest. Then there is a 1-stack-queue sequentialization π of SQG, i.e. there is a linear order π of nodes in SQG such that:*

1. *Every node in $\bigcup_{p\in P} V_p$ occurs in π exactly once.*
2. *If $(\pi[i], \pi[j]) \in (\bigcup_{p\in P}(L_p \cup E_p) \cup \bigcup_{q\in Q}(E_q))$, then $i < j$. Furthermore, if $(\pi[i], \pi[j]) \in E_q$ and $(\pi[i], \pi[i']) \in L_p$ then $j < i'$.*
3. *There are no four indices $i, j, i', j' \in [|\pi|]$ with $i < i' < j < j'$ such that $(\pi[i], \pi[j]) \in E_p$ and $(\pi[i'], \pi[j']) \in E_{p'}$, for some $p, p' \in P$.*

Now by using a 1-queue sequentialization we define a 1-nested word decomposition of the well-queuing stack-queue graph over a forest architecture.

Definition E.4 (NESTED-WORD DECOMPOSITION**)**
Let $SQG = (\,\{(V_p, Init_p, Final_p, L_p, E_p)\}_{p\in P},\ \{E_q\}_{q\in Q}\,)$ be a well-queuing stack-queue graph where $Arch(SQG)$ is a directed forest, and π be a 1-stack-queue sequentialization of SQG. Then, the nested word decomposition $nwd(SQG, \pi)$ of G according to π is the pair (H, bag) where $H = (V, Init, Final, L, E)$ is a 1-NW in which V, L, E and bag are the smallest sets that satisfy the following conditions. Let $t = |\pi|$.

- $V = \{\, \pi[i] \mid i \in [t] \,\}$.
- $Init = (\pi[1], \pi[1])$, *and* $Final = (\pi[t], \pi[t])$.
- $(\pi[i], \pi[i+1]) \in L$, *for every* $i \in [t-1]$.
- *If* $(\pi[i], \pi[j]) \in E_p$, *for some* $p \in P$, *then* $(\pi[i], \pi[j]) \in E$.
- *Let* $last : (P \times [t]) \to ([t] \cup \{\perp\})$ *be the map defined as:* $last(p, i)$ *is the greatest index* $j < i$ *such that* $\pi[j]$ *is a node of* V_p. *If no nodes of* V_p *appears in* π_i, $last(p, i) = \perp$. *Then,* $bag(\pi[i]) = \{\, \pi[j] \mid last(p, i) = j$ *and* $j \neq \perp \,\}$.

Lemma E.5 *Let SQG be a well-queuing stack-queue with n processes, where $Arch(SQG)$ is a directed tree, and π be a 1-stack-queue sequentialization of SQG. Then, $nwd(SQG, \pi) = (H, bag)$ is a nested word decomposition of SQG of width at most n.*

Proof From Definition E.4 and condition 3. of Lemma E.3, it is direct to see that H is a 1-nested word and $|bag(v)| \leq n$, for every $v \in (\bigcup_{p \in P} V_p)$. We only need to prove that (H, bag) is a graph-decomposition of SQG.

From condition 1. of Lemma E.3, all nodes of SQG are also nodes of H. It is easy to see that $v \in bag(v)$.

Now we show that if (u, v) is a edge of SQG then there are two nodes u', v' of H such that $u \in bag(u')$, $v \in bag(v')$, and (u', v') is an edge of H. We distinguish the cases when (u, v) belongs to E_p or L_p, or E_q, for some $p \in P$ and $q \in Q$:

- If $(u, v) \in E_p$ then by definition of $nwd(SQG, \pi)$, $(u, v) \in E$ with $u \in bag(u)$ and $v \in bag(v)$.
- Let $(u, v) \in L_p$. If $(u', v) \in L$, then $u \in bag(u')$. In fact, by condition 2. of Lemma E.3 the first node of V_p occurring in π, and hence in H, after u must be v. Since $v \in bag(v)$, we have that $(u', v) \in L$ is the H edge that represents the SQG edge (u, v).
- Let $(u, v) \in E_q$, and $u \in V_p$. By condition 2. of Lemma E.3, u is the last node of V_p that appears in π, and hence H, before v. Thus, if u' is such that $(u', v) \in L$, then $u \in bag(u')$ and $v \in bag(v)$.

Here we prove that if v, v' are two nodes of H and $z \in (bag(v) \cap bag(v')$ then v and v' are connected by a path in H in which all the nodes along that path have z in their bags. Let $z \in V_p$, and w.l.o.g. let v occur before v' in π. By definition of H, v and v' are connected in H by the unique L path from v to v'. Moreover, if $z \in bag(v)$ and $z \in bag(v')$ then also all nodes between v and v' have z in their bags.

□

Since 1-nested words have tree-width 2, then from the previous lemma and Lemma E.2 we can conclude with the main result of the section.

Theorem E.6 *The tree-width of any well-queuing stack-queue graph with n processes whose architecture is a directed forest is $O(n)$.*

F Distributed queue automata and queue graphs

In this section we give the syntax and semantics distributed queue Automata, and define queue graphs which capture the behaviors of distributed queue automata.

Definition F.1 (DISTRIBUTED AUTOMATA WITH QUEUES) *A* distributed queue automaton *(DQA) is a tuple $M = (P, Q, \Pi, Sender, Receiver, \{A_p\}_{p \in P})$ where P is a finite set of process names, Q is a finite set of queues, Π is a finite message alphabet, and $Sender\colon Q \to P$ and $Receiver\colon Q \to P$ are two maps that assign a unique* sender *process and* receiver *process for each queue, respectively. For every process $p \in P$, $A_p = (S_p, s_0^p, F_p, \delta_p)$ is the machine at site p, where S_p is a finite set of states, $s_0^p \in S_p$ is the initial state, $F_p \subseteq S_p$ is the set of final states, and $\delta_p = \langle \delta_{int}^p, \delta_{send}^p, \delta_{recv}^p \rangle$ where*

- *$\delta_{send}^p \subseteq (S_p \times Q_{send}^p \times \Pi \times S_p)$ is the set of send moves, where $Q_{send}^p = \{\ q \in Q \mid Sender(q) = p\ \}$;*

- $\delta^p_{recv} \subseteq (S_p \times Q^p_{recv} \times \Pi \times S_p)$ *is the set of receive moves, where* $Q^p_{recv} = \{\ q \in Q \mid Receiver(q) = p\ \}$*;*
- $\delta^p_{int} \subseteq (S_p \times S_p)$ *is the set of internal moves.*

For the rest of the section we fix $M = (P, Q, \Pi, Sender, Receiver)$ to be a DQA, where $A_p = (S_p, s^p_0, F_p, \delta_p)$ for every $p \in P$.

The semantics of DQAs is as follows.

A *configuration* of a DQA M is a tuple $\langle \{s_p\}_{p\in P}, \{\mu_q\}_{q\in Q}\rangle$ where for each $p \in P$, $s_p \in S_p$ is the state of process p, and for each queue $q \in Q$, $\mu_q \in \Pi^*$ is the content of q. A configuration $C = \langle \{s_p\}_{p\in P}, \{\mu_q\}_{q\in Q}\rangle$ of M is an initial configuration if $s_p = s^p_0$ for each $p \in P$, and $\mu_q = \epsilon$, for each queue $q \in Q$. C is a ***final*** configuration if $s_p \in F_p$, for *every* process $p \in P$, and further all queues are empty, i.e. $\mu_q = \epsilon$, for each $q \in Q$.

Let the *actions of process* p be $B_p = \{int_p\} \cup (\bigcup_{q\in Q}\{send_{(p,q)}\}) \cup (\bigcup_{q\in Q}\{recv_{(p,q)}\})$, and $B = \bigcup_{p\in P} B_p$ be the alphabet of all actions. For any two configurations $C = \langle \{s_p\}_{p\in P}, \{\mu_q\}_{q\in Q}\rangle$ and $C' = \langle \{s'_p\}_{p\in P}, \{\mu'_q\}_{q\in Q}\rangle$, $C \xrightarrow{act} C'$, if $act \in B$ and one of the following holds:

[Send] $act = send_{(p,q)}$, and there is a move $(s_p, q, m, s'_p) \in \delta^p_{send}$ such that

- for each $\hat{p} \neq p$, $s'_{\hat{p}} = s_{\hat{p}}$,
- $\mu'_q = m.\mu_q$, and for each $\hat{q} \neq q$, $\mu'_{\hat{q}} = \mu_{\hat{q}}$.

[Receive] $act = recv_{(p,q)}$, and there is a move $(s_p, q, m, s'_p) \in \delta^p_{recv}$ such that

- for each $\hat{p} \neq p$, $s'_{\hat{p}} = s_{\hat{p}}$,
- $\mu_q = \mu'_q.m$, and for each $\hat{q} \neq q$, $\mu'_{\hat{q}} = \mu_{\hat{q}}$.

[Internal] $act = int_p$, and there is a move $(s_p, s'_p) \in \delta^p_{int}$ such that

- for each $\hat{p} \neq p$, $s'_{\hat{p}} = s_{\hat{p}}$,
- for each $\hat{q}$, $\mu'_{\hat{q}} = \mu_{\hat{q}}$.

Let $w = act_1 act_2 \ldots act_{m-1} \in B^*$. A *run* of M on w is a sequence $\rho = C_1 \xrightarrow{act_1} C_2 \ldots \xrightarrow{act_{m-1}} C_m$, where C_1 is initial and C_m is final.

The *behavior* of a distributed automaton is better viewed as a *tuple* of words, one for each process, rather than as a single word. This view of a behavior will let us capture its graph more naturally.

Let $\{w_p\}_{p\in P}$ be a tuple of words, one for each process, where each $w_p \in B^*_p$. We say $\{w_p\}_{p\in P}$ is a *behavior* of the DQA M if there is a word $w \in B^*$ such that there is a run of M on w, and further $w_p = w \downarrow B_p$, for each $p \in P$.

The behaviors hence capture the distributed nature of the automaton, and the actions on the queues each process performs. Let $Beh(M)$ denote the set of all behaviors of M. The *emptiness* problem for a DQA M is the problem of checking if $Beh(M)$ is empty (or equivalently, whether there is a run of M).

Queue graphs: We now define queue graphs, which capture a distributed behavior through a graph, by capturing the local behaviors of the process as disjoint linearly ordered sets of vertices, and the queues using *queue edges* that relate nodes that enqueue nodes to the corresponding dequeuing nodes. Graph automata on queue graphs can simulate distributed queue automata, without any additional storage.

Definition F.2 (QUEUE GRAPHS**)** *A* queue graph *(QG) over the name sets P and Q, is a tuple $QG = (\{(V_p, Init_p, Final_p, L_p)\}_{p\in P}, \{E_q\}_{q\in Q})$, where*

- *V_p is a finite set of vertices, for each $p \in P$. For all $p, p' \in P$ with $p \neq p'$, $V_p \cap V_{p'} = \emptyset$.*
- *$L_p \subseteq V_p \times V_p$ is a non-reflexive (successor) edge relation such that L_p^* defines a linear ordering $<_{L_p}$ on the vertices of V_p;*
- *If x is the minimal element w.r.t. L_p then $Init_p = \{(x, x)\}$; if x is the maximal element w.r.t. L_p, then $Final_p = \{(x, x)\}$;*
- *$E_q \subseteq V_p \times V_{p'}$ for some $p, p' \in P$. Further, for all $u, x \in V_p$ and $v, y \in V_{p'}$, if $E_q(u, v)$ and $E_q(x, y)$ and $u <_{L_p} x$ hold, then $v <_{L_{p'}} y$.*
- *Any vertex $v \in \bigcup_{p\in P} V_p$ has at most one edge of the kind E_q ($q \in Q$) incident on it.*

Definition F.3 (POLYFOREST QUEUE GRAPHS**)**
A queue graph $QG = (\{(V_p, Init_p, Final_p, L_p)\}_{p\in P}, \{E_q\}_{q\in Q})$ *over the name sets P and Q, is a* polyforest queue graph *if the graph $(P, \{(p, p') | \exists u \in V_p, v \in V_{p'}$ such that $(u, v) \in E_q$ for some $q \in Q)\})$ is a polyforest.*

G Tree-width of polyforest queue graphs

In this section we prove that the tree-width of polyforest queue graphs with n processes is $O(n)$. The proof is done by transforming a polyforest queue graph into a well-queuing stack-queue graph on a forest architecture in which the tree-width is preserved.

We start giving a general result that holds for any graph. Let G be any directed graph, and G' be any graph obtained from G by replacing every edge (u, v) by (u, v) itself or by (v, u). G' is called a *reverse* graph of G. Let $Reverse(G)$ be the set of all reverse graphs of G. Then, next lemma claims that G and any of its reverse graph G' have the same tree-width.

Lemma G.1 *For every directed graph G and every $G' \in Reverse(G)$, $tw(G) = tw(G')$.*

Proof It is direct to see that every tree-decomposition of G is also a tree decomposition of G' and vice-versa. Therefore, G and G' have the same tree-width. □

The idea now is that any polyforest queue graph G can be turned into a forest queue graph G' such that $G' \in Reverse(G)$— simply take the underlying undirected architecture, introduce directions to make it a directed forest, and orient edges in G according to these directions (send-receive edges may now get reversed and become receive-send edges). It is clear that this graph is of the same tree-width as G (by the above lemma). But this graph corresponds to a directed forest architecture and since it does not employ stacks, it is well-queueing by definition. Hence, by Lemma E.5, we get an interpretation on a nested word of width at most n. However, notice from the proof of this lemma, that since the original graph did not have any nesting edges, the interpretation can be defined on a linear word without any nesting edges. It hence follows:

Theorem G.2 *The tree-width of any polyforest queue graph with n processes is at most n.*

H Complexity of the emptiness problem for graph automata on MSO definable graphs of bounded tree-width

Let φ be an MSO formula over Σ-labeled graphs, GA be a graph automaton over Σ-labeled graphs, and k be a positive integer. In this section, we sketch the proof that the problem of checking whether there exists a Σ-labeled graph of tree-width at most k that conforms to φ and is accepted by GA can be decided in time $|GA|^{O(k)}$. The procedure we give reduces the problem to the emptiness problem for tree automata.

The proof is an adaptation of the proof of Courcelle's theorem given in the book by Flum and Grohe [8]. In that proof, it is shown how to encode the tree-decomposition of a graph (and hence the graph) into a tree with a *finite* set of labels, and how any MSO sentence φ over the graph can be translated to a formula φ' over the labeled tree such that a graph of tree-width k satisfies φ iff any (and all) of its tree-decomposition encodings satisfy φ'.

This encoding uses $2^{O(k^2)}$ labels, and we can, with a more careful encoding, represent it using only $2^{O(k)}$ labels. Intuitively, in the encoding in [8], each node has a $k+1$-tuple representing at most $k+1$ vertices of the graph, and a label encodes which of them correspond the same vertex in the graph. This requires $2^{O(k^2)}$ labels; however, by having a $k+1$-tuple and simply encoding which of these are *active* (representing some node in the graph) and which are inactive, and by having each active element representing a unique node in the graph, we can encode this information using $2^{O(k)}$ labels. Also, the encoding in [8] represents which vertices in the bag of the parent of a node correspond to the vertices in the current node, which again requires $2^{O(k^2)}$ labels. However, by having the *same* position in the tuple of the node and its parent represent the same vertex, we can bring the size of labels required to capture this information to $2^{O(k)}$.

Finally, in the encoding in [8], the edges between vertices in a bag are all captured using a labeling, and since there can be $O(k^2)$ edges, we require $2^{O(k^2)}$ labels. However, we can restrict ourselves to tree-decompositions where every edge node in the tree captures at most *one* edge in the graph. And hence we can capture these tree-decompositions using only $2^{O(k)}$ labels.

The above arguments show that trees with $2^{O(k)}$ labels are sufficient to capture an adequate representative class of tree-decompositions.

Now, given a MSO formula φ_C over graphs that define a class of graphs C, we can interpret φ_C using an MSO formula $\widehat{\varphi_C}$ over the labeled trees, similar to the way this interpretation is done in [8]. This can then be converted to a tree-automaton, and since this formula is fixed, we get a fixed tree automaton capturing all trees representing graphs in C.

Finally, for any graph automaton GA, we can *directly* (without going through MSO interpretations) define a tree-automaton that works over an encoding of a tree-decomposition of a graph G, and checks whether there is a run of the GA on G. This automaton must accomplish several goals:

- It will nondeterministically guess the state label for every vertex of G represented by the current node in the tree, and check at every node if the edge encoded at the node conforms to the graph automaton's tiling requirements.
- The tree automaton will also compute the In and Out edges incident on every vertex in G and ensure that the type of the state labeling the vertex conforms to the type-requirement in GA.

Such a tree-automaton can be constructed with only $|GA|^{O(k)}$ states and transitions. The product of this tree-automaton and the tree-automaton representing all encodings of graphs in C will be checked for emptiness, which is decidable in time $|GA|^{O(k)}$. The language of the product automaton is non-empty iff there is graph in C accepted by the graph automaton.