The Tree Width of Auxiliary Storage

P. Madhusudan
University of Illinois at Urbana-Champaign, USA
madhu@illinois.edu

Gennaro Parlato
LIAFA, CNRS and University of Paris Diderot, France.
gennaro@liafa.jussieu.fr

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) restricted to 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 results.

Categories and Subject Descriptors F.1.1 [*Theory of Computation*]: Models of Computation: Automata; D.2.4 [*Software Engineering*]: Software/Program Verification: Model checking; F.4.3 [*Theory of Computation*]: Formal Languages: Decision problems

General Terms Algorithms, Reliability, Theory, Verification

Keywords model checking, automata, decision procedures, bounded tree-width

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 [14], and several new models like restricted classes of multi-stack pushdown automata, automata with queues, and automata with both stacks and queues, have been proved decidable recently [8, 15, 17, 22].

The decidability of emptiness of these automata has often been motivated for *model-checking* systems. Software models can be captured using automata with auxiliary storage, as stacks can model the *control recursion* in programs while queues model *FIFO communication between processes*. In abstraction-based model-checking, data domains get abstracted from programs, resulting in automata models (e.g., the SLAM tool builds pushdown automata models using *predicate abstraction* [7], and the GETAFIX tool model-checks both single-stack and multi-stack automata models [18, 19]). The emptiness problem for these automata is the most relevant problem as it directly corresponds to checking reachability of an error state.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
POPL'11, January 26–28, 2011, Austin, Texas, USA.
Copyright © 2011 ACM 978-1-4503-0490-0/11/01. . . $10.00

However, the various identified decidable restrictions on these automata 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! [8]. Yet, relaxing these definitions to more natural ones seems to either destroy decidability or their power. It is hence natural to ask: why do these automata have decidable emptiness problems? 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 as well as their sequential or distributed nature, and are also of bounded tree-width.

More precisely, we can show, using generalizations of known results on the decidability of *satisfiability* of monadic second-order logic (MSO) on bounded tree-width graphs [9, 23], that graph automata on *MSO-definable* graphs of bounded tree-width are decidable. Graph automata [24] are finite-state automata (without auxiliary storage) that accept or reject graphs using *tilings of the graph using states*, 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.

We proceed to 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 directed paths augmented 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 by a graph automaton working on a graph where there is 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 automata with a tractable emptiness problem 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 of the following classes of automata with auxiliary storage by graph automata working on MSO-definable bounded tree-width graphs:

- 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 [22]). 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 [15]). We show that graph automata on graphs of tree-width $O(2^k)$ (not $O(k)$ as in the above case) can simulate them.

- Ordered multi-stack pushdown automata: The restriction here is that there is 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 divided 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* network graph of the architecture is a forest), the emptiness problem is decidable (and for other architectures, it is undecidable) [17]. 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 [17]. 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 [17]. 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 a set of *nesting edges relations*, one relation for each stack. For distributed queue automata, the graphs are composed of n distinct linear 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 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* [1–3], where nesting edges capture a visible stack. Our work is also motivated by the work on bounded-phase multi-stack automata [15], in which we were involved, where tree-interpretations were used to show decidability of emptiness. Surveying the other 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 different means— bounded context-switching multi-stack automata were shown to be decidable using regularity of *tuples* of reachable configurations [22], ordered multi-stack automata were shown decidable using manipulations of associated grammars, followed by a Parikh theorem [8], and distributed queue automata with stacks were shown decidable by reductions to bounded-phase automata [17].

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 [10, 11]; this result generalizes all the above multi-stack sequential automata. Second, several of our results extend 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. Third, several variants of the restrictions can be proved immediately decidable— for example, suppose 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. Section 5 gives a summary of consequences of our general result.

Due to the variety of automata models we consider we do not give all definitions and proofs in the main text. The proofs for the boundedness of tree-width for various restrictions of multi-stack pushdown automata are given in the Appendix, while the proofs regarding distributed queue automata can be found in the technical report [20].

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 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.

[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 and are related by an incidence relation; that version is stronger than ours, but we shall not need it for our exposition.

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 $(\mathit{In}, \mathit{Out})$ of sets 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 $\mathit{In} = \{a \mid \exists v, (v, u) \in E_a\}$ and $\mathit{Out} = \{a \mid \exists v, (u, v) \in E_a\}$.

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 [24]. Graph automata can in fact be defined more powerfully (see [24]); however, for our purposes, the above definition will suffice. Most of our results will carry over to generalizations of the above definition.

Tree-width: We recall the definition of tree-width for graphs (see [12]). The tree-width of a graph intuitively captures how close a graph is to a tree.

Formally, a *tree-decomposition* of a graph (V, E) is a pair (T, bag), where $T = (N, \to)$ is a tree, and $bag : N \to 2^V$ 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 all of its tree decompositions.

It is easy to see that the tree-width of a tree is 1 while the tree-width of a k-clique is $k - 1$.

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) [23]. Courcelle's classic theorem shows that checking if a *particular graph G* of tree-width k (for a fixed k) satisfies a fixed MSO formula is decidable in linear time. This result works by defining the graph in a labelled tree by MSO formulas, and by translating the MSO formula about graphs into one about trees, and using a tree-automaton for the MSO formula to check if the corresponding tree is accepted. It turns out the same proof can be used to prove the *satisfiability* theorem that we refer to above as well. Intuitively, we can interpret *all* graphs of tree-width k by using a uniform set of labeled binary trees whose labels only depend on k, translate the MSO formula on graphs to these labeled trees, and use the fact that satisfiability of MSO on trees is decidable.

THEOREM 2.1 (Seese [23]). *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 (if φ_C defines the class of graphs, and φ is the MSO formula, we can instantiate the above theorem for $\varphi_C \wedge \varphi$).

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 to 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 [20] for a gist of proof): to obtain our 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 show the decidability of emptiness of various restricted multi-stack pushdown automata (bounded context-switches, bounded phase, and ordered), by showing that they can be simulated by graph automata working over *multiply-nested word graphs* that are of bounded tree-width.

For any $n \in \mathbb{N}$, let $[n]$ denote the set $\{1, \ldots n\}$.

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

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 the *initial* configuration if $q = q_0$ and $s_j = \epsilon$, for every $j \in [n]$. Moreover, C is a *final* configuration if $q \in Q_F$ and $s_j = \epsilon$, for every $j \in [n]$. 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 $B_n = \{int,\ push_1, \ldots push_n,\ pop_1, \ldots, pop_n\}$, 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 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 they are performed.

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 a class of (edge-labelled) graphs that we call *multiply nested words*, and the emptiness problem on the former reduces to the emptiness problem on the latter. We start by defining multiply nested word 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 a finite set of vertices;*
- *$L \subseteq V \times V$ is a non-reflexive (successor) edge relation such that L^* is a linear ordering $<_L$ on the vertices of V;*
- *If x is the minimum 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 $n \in \mathbb{N}$.

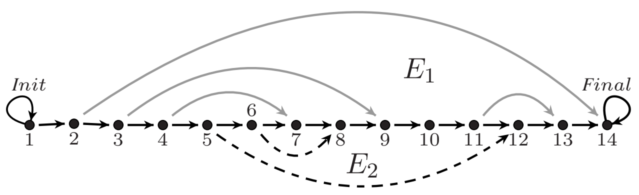


Figure 1. A 2 nested word graph.

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) 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 function nw from the set of behaviors of the n-PDA M to the class of n-NWs. Given an M run ρ with $beh(\rho) = a_1a_2 \ldots a_m$, the corresponding nested word graph n-NW N is as follows. The set of vertices of N is $V = \{v_1, v_2, \ldots, v_m, v_{m+1}\}$, 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 $n > 1$ [14]. 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 mPDAs have a decidable emptiness problem, we will first define *canonical tree-decompositions* for multiply nested words, which we will use to prove bounds on tree-width and

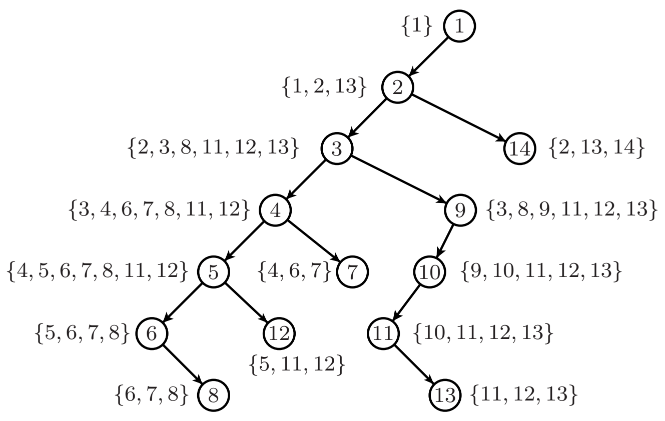


Figure 2. Tree decomposition of the graph illustrated in Figure 1.

hence prove emptiness for both bounded-phase automata as well as ordered automata (it turns out that bounded context-switching automata have a simpler tree decomposition).

DEFINITION 3.6 (CANONICAL TREE DECOMP. OF n-NWS). *For any n-NW $N = (V, Init, Final, L, \{E_j\}_{j\in[n]})$, the canonical tree-decomposition of N, $can\text{-}td(N) = (T, bag)$ is decomposition $T = (V, bag)$ 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 minimum 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 $u, v \in V$.*
- *for $u, v \in V$, 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 minimum 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.

Note 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 $u, v \in bag(v)$. 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 $can\text{-}td(N)$ 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, $can\text{-}td(N)$ is a tree-decomposition of N.*

3.1 Bounded context-switch emptiness

We show now that the multiply-nested words that correspond to bounded context-switching runs of a multi-stack automaton are of bounded tree-width, and hence admit a decidable emptiness problem.

For any $k \in \mathbb{N}$, we say that a behavior word $w \in B_n^*$ 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 $w_1 w_2 \ldots w_h$ (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 under the root (in the usual way, as in the canonical tree decomposition of 1-nested words), has width at most $k + 1$ (see Appendix A).

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 $k \in \mathbb{N}$, the emptiness problem for mPDAs restricted to k contexts is decidable, and decidable in time $O(|M|^{O(k)})$. 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* [22]. The above proof is 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

Now we show that the multiply-nested words that correspond to bounded phase runs of a multi-stack automaton are of bounded tree-width (in fact, the canonical tree-decomposition gives bounded tree-width), and hence entails decidable emptiness.

A word $w \in B_n^*$ 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 only internal actions, pushes to all stacks, and pops from stack j are permitted. A word $w \in B_n^*$ is a *k-phase* behavior word if it is the concatenation of at most k phases: that is, $w \in (\bigcup_{j\in[n]}(phase_j))^k$. We define the set k-*Phase-Beh*(M), for a mPDA M, as the set of all the k-phase words in *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 to obtain the following theorem, which also matches the 2ETIME lower bound for this problem [16].

THEOREM 3.11. *For any $k \in \mathbb{N}$, the emptiness problem for mPDAs restricted to k phases is decidable, and decidable in time $|M|^{O(2^k)}$. When the number of phases is fixed, the emptiness problem is in* PTIME.

Proofs can be found in Appendix B.

3.3 Emptiness of ordered multi-stack automata

Turning to the orderedness restriction on multi-stack automata, we show that the multiply-nested words that correspond to ordered runs are of bounded tree-width (using the canonical tree-decomposition), and hence admits a decidable emptiness problem.

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 we obtain the following theorem, which also matches the 2ETIME lower bound for this problem [6].

THEOREM 3.13. *The emptiness problem for mPDAs restricted to ordered runs is decidable, and decidable in time $|M|^{O(n.2^n)}$. When the number of stacks is fixed, the problem is decidable in* PTIME.

Proofs can be found in Appendix B.

4. Distributed Automata with Queues and Stacks

Distributed queue automata with stacks (DQSA) 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.

DEFINITION 4.1 (DISTRIBUTED QUEUE AUT. 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: $Q \rightarrow P$ and Receiver: $Q \rightarrow P$ 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. The configuration $C = \langle \{s_p\}_{p \in P}, \{\gamma_p\}_{p \in P}, \{\mu_q\}_{q \in Q} \rangle$ of M is the 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 $q \in Q$, and all stacks are also empty, i.e. $\gamma_p = \epsilon$, for each $p \in P$.

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

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

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

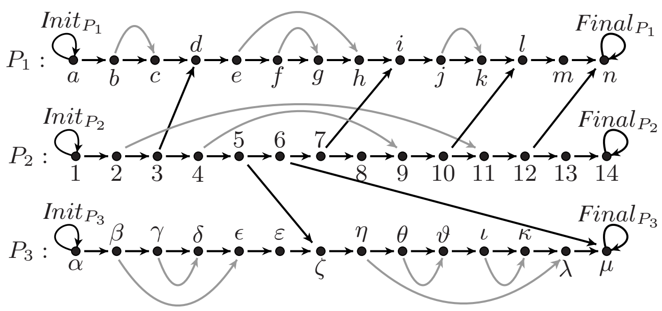
[Push] $act = push_p$, and there is a move $(s_p, s'_p, a) \in \delta_{push}^p$ such that

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

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

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

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



$Init_{P_1}$
$Final_{P_1}$
P_1 :
a b c d e f g h i j k l m n
$Init_{P_2}$
$Final_{P_2}$
P_2 :
1 2 3 4 5 6 7 8 9 10 11 12 13 14
$Init_{P_3}$
$Final_{P_3}$
P_3 :
α β γ δ ϵ ε ζ η θ ϑ ι κ λ μ

Figure 3. A stack-queue graph.

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

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.

The set of behaviors of M, $Beh(M)$ is the set of words $w \in B^*$ such that there is a run of M on w.

A *stack-queue graph* (SQG) captures the behaviors of DQSA 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 4.2 (STACK-QUEUE GRAPHS). *A* stack-queue graph *(SQG) over* $(P, Q, Sender, Receiver)$ *(where* P, Q *are finite sets,* $Sender : Q \rightarrow P$ *and* $Receiver : Q \rightarrow P$*) 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 some* $p, p' \in P$ *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 4.3. *For any tuple* $(P, Q, Sender, Receiver)$*, the class of stack-queue graphs over it 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 behavior w. Then,

LEMMA 4.4. *For any DQSA* M *over* $(P, Q, Sender, Receiver)$*, there is an effectively constructible graph automaton on stack-queue graphs over* $(P, Q, Sender, Receiver)$ *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 4.5. *For any* $(P, Q, Sender, Receiver)$*, where* $|P| > 2$ *and* $Q \neq \emptyset$*, the class of stack-queue graphs over* $(P, Q, Sender, Receiver)$ *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 [17], 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 DQSAs. 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 the technical report [20]. 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 4.6. *The set of all stack-queue graphs over a pair* $(P, Q, Sender, Receiver)$ *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.*

From Lemma 4.6, we have:

THEOREM 4.7. *The emptiness problem for a well-queuing DQSA* 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 over the one proved in [17], which gives an algorithm doubly exponential in n.

4.1 Distributed Queue Automata without stacks

Distributed Queue Automata without stacks (DQAs) are the same model as that of DQASs except that the local stacks at each process are not present. 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. Figure 4 illustrates a queue graph. In general, queue graphs of distributed queue automata without stacks are also of unbounded tree width. Formally, we define queue graph as a stack queue graph with an empty set of stack edges:

DEFINITION 4.8 (QUEUE GRAPHS). *A* queue graph *(QG) over* $(P, Q, Sender, Receiver)$*, is a tuple*

$$QG = (\,\{(V_p, Init_p, Final_p, L_p)\}_{p \in P},\ \{E_q\}_{q \in Q}\,),$$

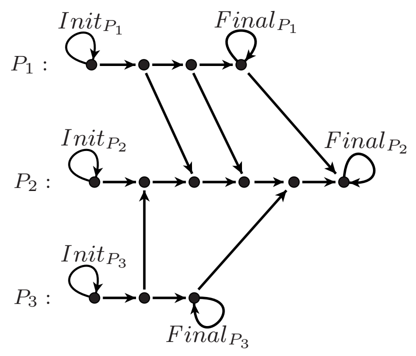


Figure 4. A queue graph.

where ($\{(V_p, Init_p, Final_p, L_p)\}_{p\in P}$, $\{E_p\}_{p\in P}$, $\{E_q\}_{q\in Q}$) *is a stack queue graph, where every* $E_p = \emptyset$*, for each* $p \in P$.

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

LEMMA 4.9. *For any* $(P, Q, Sender, Receiver)$*, the class of queue graphs over it is MSO definable.*

Also, let qg be a function that associates to every behavior of a distributed queue automaton the corresponding queue graph.

LEMMA 4.10. *For every DQA* M*, there is a (constructible) graph automaton* GA *on queue graphs such that* $qg(Beh(M)) = L(GA)$.

In [17], 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 tree-width of 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 [20]). 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 4.11. *Let* $(P, Q, Sender, Receiver)$ *be a tuple where* P *and* Q *are finite sets and* $Sender : Q \to P$ *and* $Receiver : Q \to P$*. Then the class of polyforest queue graphs over* $(P, Q, Sender, Receiver)$ *has tree-width (even path-width) bounded by* $|P|$.

Furthermore, from Lemma 4.10, Lemma 4.9 , and Theorem 2.3, we can conclude:

THEOREM 4.12. *The emptiness problem for polyforest DQAs is decidable, and decidable in time* $|M|^{O(n)}$*, where* n *is the number of processes of* M.

5. Conclusions and further results

The main contribution of this paper is to provide a uniform framework using which we can prove decidability of emptiness of a variety of automata with auxiliary storage. In this sense, our framework is the "mother" of several automata decidability results proved recently in the literature, where complex but awkward restrictions have been imposed to obtain decidability of emptiness. We also believe that our results can help in the search of new automaton models that have a tractable emptiness problem using the principles outlined by our framework.

There are several other results that follow immediately from our work, that we discuss below.

Under-approximation of abstracted programs using tree-width: The analysis of abstracted concurrent programs communicating through shared-variables is in general undecidable. These programs can be modeled as multi-stack pushdown automata. In the last few years, syntactic restrictions on the behaviors of those automata have been considered with the aim of making the analysis of such programs decidable, e.g. bounded context-switches [22], bounded-phases [15]), etc.

Since all the known syntactic restrictions correspond to graphs of bounded tree-width, we can consider the tree-width as a natural *semantic restriction* to consider for under-approximations. Given a multi-stack automaton and $k \in \mathbb{N}$, the problem of deciding whether there is a multiply nested word of tree-width k that is accepted by it is decidable, as shown in our framework, and hence can be used as an under-approximation technique to explore the state-space reached by a concurrent program. Note that this would cover all behaviors that explore k context-switches, and more, and yet has the same complexity.

Improvement in complexity for DQSAs: As mentioned earlier, Theorem 4.7 improves the complexity of the emptiness problem for a well-queuing DQSA M with tree-architectures to one exponential in the number of processes; the algorithm given in [17] is doubly exponential. This upper bound complexity matches the EXPTIME lower-bound for the emptiness problem on DQSAs [13].

Decidable emptiness problem for multi-stack pushdown automata with bounded* reverse-phase*: Our framework shows immediately the decidability of other restrictions placed on automata with auxiliary storage. For example, fix $k \in \mathbb{N}$ and consider multi-stack automata behaviors restricted to k *reverse-phases*, where in each reverse-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 for this class, since the graphs corresponding to the runs of 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.

A general Parikh theorem: We can prove a general *Parikh theorem* [21] for all classes of automata that can be compiled to graph automata of bounded tree-width. The idea is to encode the graph into a tree using the tree-decomposition, with a unique vertex of the tree for every graph node. Since a depth-first traversal of the tree can be captured by a pushdown automata, we can build a context-free grammar that generates the graph nodes in some order. Using the classic Parikh theorem for context-free grammars, we can show that the labels of the graph nodes define a semi-linear set. This is a generalization of the technique using in [15], where a similar argument was used for proving a Parikh theorem for bounded-phase multi-stack automata.

Extension to infinite behaviors: Several of our results extend to automata over infinite behaviors. For example, consider ordered multi-stack *Büchi* or *parity* automata on infinite words. We can show that there are graph automata on multiply-nested infinite graphs (with appropriate Büchi and parity conditions) that can simulate these automata, and further that these graphs have bounded tree-width. This proves that the emptiness problem for this class of automata is decidable. (See [5] for recent results in this direction.) Similar results can be obtained by extending the tractable distributed automata presented in this paper to infinite words.

There are interesting temporal logics suitable for expressing properties of single-stack pushdown systems, like the logic CARET [4]). Natural extensions of temporal logics like CARET that allow to reason with multi-stack pushdown automata are also possible, and can be proved decidable for all multi-stack automata whose runs can be modeled by graphs of bounded tree-width.

Acknowledgments

This work was partially supported by the NSF CAREER grant #0747041 and the French ANR-09-SEGI project Veridyc.

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] R. Alur, K. Etessami, and P. Madhusudan. A temporal logic of nested calls and returns. In K. Jensen and A. Podelski, editors, *TACAS*, volume 2988 of *Lecture Notes in Computer Science*, pages 467–481. Springer, 2004.

[5] M. F. Atig. Global model checking of ordered multi-pushdown systems. In K. Lodaya and M. Mahajan, editors, *Proceedings of the 30th Conference on FSTTCS*, Leibniz International Proceedings in Informatics, Chennai, India, Dec. 2010. To appear.

[6] 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.

[7] T. Ball, R. Majumdar, T. D. Millstein, and S. K. Rajamani. Automatic predicate abstraction of C programs. In *PLDI*, pages 203–213, 2001.

[8] 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.

[9] 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.

[10] B. Courcelle and S. Olariu. Upper bounds to the clique width of graphs. *Discrete Applied Mathematics*, 101(1-3):77–114, 2000.

[11] B. Courcelle, J. A. Makowsky, and U. Rotics. Linear time solvable optimization problems on graphs of bounded clique-width. *Theory Comput. Syst.*, 33(2):125–150, 2000.

[12] 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.

[13] 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.

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

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

[16] 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.

[17] 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.

[18] S. La Torre, P. Madhusudan, and G. Parlato. Analyzing recursive programs using a fixed-point calculus. In M. Hind and A. Diwan, editors, *PLDI*, pages 211–222. ACM, 2009.

[19] S. La Torre, P. Madhusudan, and G. Parlato. Model-checking parameterized concurrent programs using linear interfaces. In T. Touili, B. Cook, and P. Jackson, editors, *CAV*, volume 6174 of *Lecture Notes in Computer Science*, pages 629–644. Springer, 2010.

[20] P. Madhusudan and G. Parlato. The tree width of automata with auxiliary storage. In *IDEALS Technical Report* `http://hdl.handle.net/2142/15433`, April 2010.

[21] R. Parikh. On context-free languages. *J. ACM*, 13(4):570–581, 1966.

[22] 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.

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

[24] 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.

Appendix

A. Tree-width of bounded-context multiply nested words

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

LEMMA A.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 canonical 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 in the path from u to v. Clearly, the bag-sizes increase by at most $k - 1$, and the resulting tree-decomposition captures all edges and is of width at most $k + 1$.

B. 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 canonical tree decomposition $can\text{-}td(N)$, 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 canonical 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 *can-td*(N). We show that this tree-decomposition certainly has width at least as the width of *can-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 $N = (V, \mathit{Init}, \mathit{Final}, L, \{E_j\}_{j\in[n]})$. Moreover, whenever we refer to the 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 $b <_L c$. Furthermore, T is the tree obtained as $(T, bag) =$ *can-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 $N' = (V', \mathit{Init}', \mathit{Final}', L', \{E'_j\}_{j\in[n]})$, 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 must occur before a, and hence skipping all nodes between b and a.) 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 $\widehat{N} = (\widehat{V}, \widehat{\mathit{Init}}, \widehat{\mathit{Final}}, \widehat{L}, \{\widehat{E}_j\}_{j\in[n]})$ 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 $(v', v) \in \widehat{E}_j$ ($j \in [n]$). Then $BackPath_{\widehat{N}}(v)$ 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 $(v_{i+1}, v_i) \in \widehat{E}_j$.
  Otherwise v_{i+1} is the linear predecessor of v (i.e. the node such that $(v_{i+1}, v_i) \in \widehat{L}$).

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 $N = (V, \mathit{Init}, \mathit{Final}, L, \{E_j\}_{j\in[n]})$. 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, $N_0 = (V \times \{1\}, \mathit{Init}_0, \mathit{Final}_0, L_0, \{E^0_j\}_{j\in[n]})$, where the various edges in N_0 are appropriately defined.

We now construct N_{i+1} from N_i using the following algorithm. Let $N_i = (V_i, \mathit{Init}_i, \mathit{Final}_i, L_i, \{E^i_j\}_{j\in[n]})$, where $V_i \subseteq V \times \mathbb{N}$. 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_i (if no such pop-edge exists, then we set $N_{i+1} = N_i$, and reach a fixed-point). Then $N_{i+1} = (V_{i+1}, \mathit{Init}_i, \mathit{Final}_i, L_{i+1}, \{E^i_j\}_{j\in[n]})$ is defined as follows (note that the initial, final, and nesting edges do not change).

Let the back-path from $\langle u, 1\rangle$ be $BackPath_{N_i}(\langle u, 1\rangle) = \langle z_1, h_1\rangle \dots \langle z_t, h_t\rangle$. Note that any node occurs at most once in the back-path. Let us now relabel this path so that the nodes $\langle z_j, h_j\rangle$ get renamed to some $\langle z_j, h'_j\rangle$ so that they are not in V_i and do not get repeated in the back-path:

- *relabel*$_{V_i}(\epsilon) = \epsilon$
- *relabel*$_{V_i}(w, \langle x, m\rangle) =$ *relabel*$_{V_i}(w), \langle x, m'\rangle$ where m' is the least positive integer such that $\langle x, m'\rangle \notin V_i$ and does not occur in $relabel_{V_i}(w)$.

Let *relabel*$_X(BackPath_{N'}(\langle u, 1\rangle)) = \langle z_1, h'_1\rangle \dots \langle z_t, h'_t\rangle$. Then, $V_{i+1} = (V_i \cup \{\langle z_i, h'_i\rangle \mid i \in [t]\})$ and the set L_{i+1} is:

$$L_{i+1} = (L_i \setminus \{(\langle u, 1\rangle, \langle v, 1\rangle)\}) \cup \{(\langle z_i, h'_i\rangle, \langle z_{i+1}, h'_{i+1}\rangle) \mid i \in [t]\}$$

$$\cup \{(\langle u, 1\rangle, \langle z_1, h'_1\rangle), (\langle z_t, h'_t\rangle, \langle v, 1\rangle)\}$$

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 B.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.

We define a map $bag' : V \rightarrow 2^V$ as follows. Map bag' associates the minimum 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 $BackPath_{N'}(\langle u,1\rangle) = \langle u_1,h_1\rangle \ldots \langle u_t,h_t\rangle$, 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 one: 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 B.2. *Let N be an n-nested word, and $\mathcal{T} = (T, bag) = can\text{-}td(N)$. 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 $d_v = |\{\langle v,h\rangle \in V' \mid h \in \mathbb{N}\}|$.*

Generator Trees: A convenient way to calculate d_v (in the above lemma) is to represent the set of N' nodes $\{\langle v,h\rangle \in V' \mid h \in \mathbb{N}\}$ as a tree, for each $v \in V$. 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 B.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 $\langle v,h\rangle$ is the stack whose popping led to a back-path that created $\langle v,h\rangle$.

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 $\langle v,1\rangle$, 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 B.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 B.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.

□

B.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 B.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 B.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 B.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 B.4 we can conclude the proof. □

By using a similar argument of the previous proof, and Lemma B.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 B.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 B.6, we have that

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

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

THEOREM B.8. *The tree-width of any k-phase mNW is at most $2^k + 2^{k-1} + 1$.*

B.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 B.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 B.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 B.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 B.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 B.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 B.2 we can conclude with the main theorem of the section.

THEOREM B.10. *The tree-width of any ordered n-NW is at most $n \cdot 2^{n-1}$.*