Nested Words for Order-2 Pushdown Systems*

C. Aiswarya[1], Paul Gastin[2], and Prakash Saivasan[3]

1 **Chennai Mathematical Institute**
aiswarya@cmi.ac.in
2 **LSV, ENS-Cachan, CNRS, INRIA, Univ. Paris-Saclay, 94235 Cachan, France**
gastin@lsv.fr
3 **University of Kaiserslautern**
saivasan@rhrk.uni-kl.de

Abstract. We study linear time model checking of collapsible higher-order pushdown systems (CPDS) of order 2 (manipulating stack of stacks) against MSO and PDL (propositional dynamic logic with converse and loop) enhanced with push/pop matching relations. To capture these linear time behaviours with matchings, we propose order-2 nested words. These graphs consist of a word structure augmented with two binary matching relations, one for each order of stack, which relate a push with matching pops (or collapse) on the respective stack. Due to the matching relations, satisfiability and model checking are undecidable. Hence we propose an under-approximation, bounding the number of times an order-1 push can be popped. With this under-approximation, which still allows unbounded stack height, we get decidability for satisfiability and model checking of both MSO and PDL. The problems are ExpTime-Complete for PDL.

1998 ACM Subject Classification F.1.1 [Computation by Abstract Devices]: Models of Computation; F.3.1 [Logics and Meanings of Programs]: Specifying and Verifying and Reasoning about Programs

Keywords and phrases Higher-order pushdown systems, Nested words, Model checking, split-width

1 Introduction

The study of higher-order pushdown systems (HOPDS), has been a prominent line of research [2, 11–13, 20, 24–28]. HOPDS are equipped with a stack (order-1 stack, classical pushdown), or a stack of stacks (order-2 stack), or a stack of stacks of stacks (order-3 stack) and so on. They naturally extend the classical pushdown systems to higher orders, and at the same time they can be used to model higher-order functions [21, 27, 28] which is a feature supported by many widely-used programming languages like scala, python, etc. [29].

An extension of HOPDS known as Collapsible HOPDS (CPDS) characterizes recursive schemes [21]. In this article we focus on CPDS of order 2 (denoted 2-CPDS). Classically, HOPDS and CPDS can be thought as generating a set of words (linear behaviour), or a tree (branching behaviour) or a configuration graph [28]. Here we consider yet another way of understanding them, as generators of linear behaviours with matching relations, like in nested words [7]. We call these structures order-2 nested words (2-NWs). They are essentially words augmented with two binary relations — an order-1 nesting relation and an order-2 nesting relation which link matching pushes and pops or collapses of the stack of the respective order. See Figure 1 depicting a 2-NW $\mathcal{N}_1$ with collapse and another one $\mathcal{N}_2$ without collapse.

We provide a characterisation of the push matching a given pop or collapse, by a context-free grammar. This allows us to compute the nesting edges, given a sequence

* Partially supported by LIA InForMeL

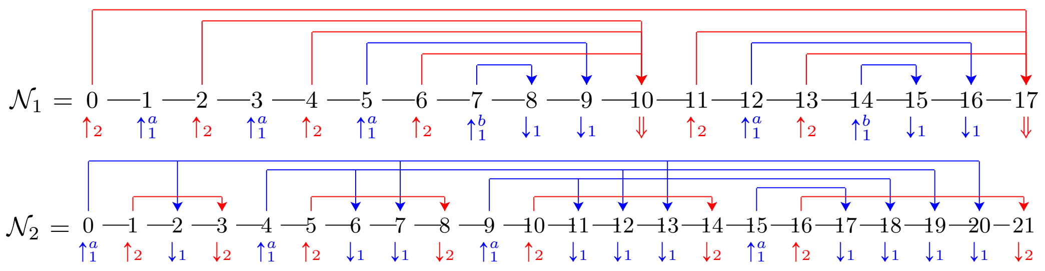

$\mathcal{N}_1$ = 0 — 1 — 2 — 3 — 4 — 5 — 6 — 7 — 8 — 9 — 10 — 11 — 12 — 13 — 14 — 15 — 16 — 17
$\uparrow_2$ $\uparrow_1^a$ $\uparrow_2$ $\uparrow_1^a$ $\uparrow_2$ $\uparrow_1^a$ $\uparrow_2$ $\uparrow_1^b$ $\downarrow_1$ $\downarrow_1$ $\Downarrow$ $\uparrow_2$ $\uparrow_1^a$ $\uparrow_2$ $\uparrow_1^b$ $\downarrow_1$ $\downarrow_1$ $\Downarrow$
$\mathcal{N}_2$ = 0 — 1 — 2 — 3 — 4 — 5 — 6 — 7 — 8 — 9 — 10 — 11 — 12 — 13 — 14 — 15 — 16 — 17 — 18 — 19 — 20 — 21
$\uparrow_1^a$ $\uparrow_2$ $\downarrow_1$ $\downarrow_2$ $\uparrow_1^a$ $\uparrow_2$ $\downarrow_1$ $\downarrow_1$ $\downarrow_2$ $\uparrow_1^a$ $\uparrow_2$ $\downarrow_1$ $\downarrow_1$ $\downarrow_1$ $\downarrow_2$ $\uparrow_1^a$ $\uparrow_2$ $\downarrow_1$ $\downarrow_1$ $\downarrow_1$ $\downarrow_1$ $\downarrow_2$

Figure 1 Two 2-NWs along with the sequence of operations generating them. $\uparrow_1$ means order-1 push, $\downarrow_1$ means order-1 pop, $\uparrow_2$ means order-2 push, $\downarrow_2$ means order-2 pop, and $\Downarrow$ means collapse.

of operations. Based on it, we have linear time algorithm, `Nestify`, for doing the same. Our tool `Nestify`, accessible at `http://www.lsv.fr/~gastin/hopda`, generates a 2-NW representation in several formats, including pdf pictures as in Example 1.

We propose propositional dynamic logic with loop and converse (LCPDL) and monadic second-order logic (MSO) over order-2 nested words to specify properties of 2-CPDS. LCPDL is a navigating logic which can walk in the order-2 nested word by moving along the nesting edges and the linear edges. It is powerful enough to subsume usual temporal logic operators. These logics are very expressive since they can use the nesting relations. We show that the satisfiability checking of these logics, and model checking of 2-CPDS against them are undecidable. The reason is that we can interpret grids in 2-NWs using these logics.

Our results are quite surprising, since they differ from the established results under classical semantics. Strikingly

- Model checking and satisfiability problems of MSO and LCPDL under 2-NW semantics turn out to be undecidable, even when the 2-CPDS is non-collapsible. Further, in our undecidability proof, the height of the order-2 stack is bounded by 2. On the other hand, MSO over non-collapsible 2-HOPDS under classical semantics is decidable [14, 24].
- The satisfiability and model checking problems described above can be reduced to that of non-collapsible 2-CPDS/2-NWs. Contrast this with the fact that, when considering HOPDS under classical semantics collapse is strictly more powerful.
- In [26], Ong showed that μ-calculus over 2-CPDS (classical semantics) is decidable. In the case of 2-NWs, LCPDL is undecidable even over non-collapsible 2-HOPDS.

Inspired by the success of under-approximation techniques in verification of otherTturing powerful settings like multi-pushdown systems, message passing systems etc., we propose an under-approximation for 2-CPDS, to confront the undecidability. This under-approximation, called bounded-pop, bounds the number of order-1 pops that a push can have. Notice that this does not bound the height of order-1 or order-2 stacks. With this restriction we gain decidability for satisfiability and model checking of MSO. For LCPDL, we show that these problems are ExpTime-Complete.

We establish decidability by showing that bounded-pop 2-NWs can be interpreted over trees. Towards a tree-interpretation, we first lift the notion of split-width [4, 16, 17] to 2-NWs, and show that bounded-pop 2-NWs have a bound on split-width. Split-width was first introduced in [17] for MSO-decidability of multiply-nested words. It was later generalised to message sequence charts with nesting (also concurrent behaviours with matching) [4, 16]. Bounded split-width 2-NWs have bounded (special) tree-width [15], and hence bounded-pop 2-NWs can be effectively interpreted over special tree-terms.

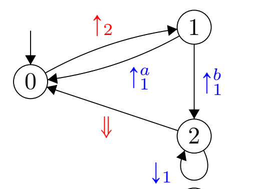
2 Order-2 pushdown systems with collapse

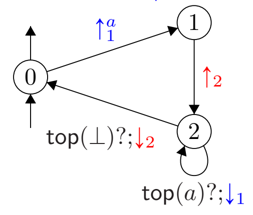
Order-2 stacks. An order-2 stack is a stack of stacks. In a collapsible order-2 stack, the stack symbols may in addition contain a pointer to some stack in the stack of stacks. Let S be a finite set of stack symbols. An order-2 stack is of the form $W = [[u_1][u_2]\dots[u_n]]$, where each $[u_i]$ is a stack over S with collapse-pointers to stacks below. Thus we may see the contents u_i of the i-th stack as a word from $S \times \{1, 2, \dots, i-1\}$ where the second component of an entry indicates the index of the stack to which the collapse-link points. The empty order-2 stack is denoted $[[]]$, where the order-2 stack contains an empty stack. We have the following operations on order-2 stacks:

- $\uparrow_2$: duplicates the topmost stack in the order-2 stack. That is, $\uparrow_2([[u_1][u_2]\dots[u_n]]) = [[u_1][u_2]\dots[u_n][u_{n+1}]]$ with $u_{n+1} = u_n$.
- $\downarrow_2$: pops the topmost stack from the order-2 stack. That is, $\downarrow_2([[u_1][u_2]\dots[u_n]]) = [[u_1][u_2]\dots[u_{n-1}]]$. Notice that $\downarrow_2([[u_1]])$ is undefined.
- $\uparrow_1^s$: pushes a symbol s to the top of the topmost stack. Further the pushed symbol contains a "collapse link" to the topmost but one stack of the order-2 stacks.[1] $\uparrow_1^s([[u_1][u_2]\dots[u_n]]) = [[u_1][u_2]\dots[u_n(s, n-1)]]$.
- $\downarrow_1$: removes the topmost element from the topmost stack. $\downarrow_1([[u_1][u_2]\dots[u_n]]) = [[u_1][u_2]\dots[u'_n]]$, if $u_n = u'_n(s, i)$. Notice that $\downarrow_1([[u_1][u_2]\dots[]])$ is undefined.
- $\Downarrow$: the collapse operation pops the stacks in the order-2 stack until the stack pointed-to by the link in the topmost symbol of the previously topmost stack becomes the topmost stack. $\Downarrow([[u_1][u_2]\dots[u_n]]) = [[u_1][u_2]\dots[u_i]]$, if $u_n = u'_n(s, i)$.
- $\mathsf{top}(s)$: checks if the topmost symbol of the topmost stack is s.
  Hence, $\mathsf{top}(s)([[u_1][u_2]\dots[u_n]]) = [[u_1][u_2]\dots[u_n]]$, if $u_n = u'_n(s, i)$. It is undefined otherwise. Also, we can check whether the topmost stack is empty by $\mathsf{top}(\perp)$.

The above defined operations form the set $\mathsf{Op}(S)$.

2-CPDS is a finite state system over a finite alphabet Σ equipped with an order-2 stack. Formally it is a tuple $\mathcal{H} = (Q, S, \Delta, q_0, F)$ where Q is the finite set of states, S is the set of stack symbols/labels, q_0 is the initial state, F is the set of accepting states, and $\Delta \subseteq Q \times \Sigma \times \mathsf{Op}(S) \times Q$ is the set of transitions. On the right, we have a 2-CPDS $\mathcal{H}_1$ with collapse and a second one $\mathcal{H}_2$ without collapse operations.




$\uparrow_1^a$
$\uparrow_2$
$\mathsf{top}(\perp)?;\downarrow_2$
$\mathsf{top}(a)?;\downarrow_1$
0
1
2

A *configuration* is a pair $C = (q, W)$ where $q \in Q$ is a state and W is an order-2 stack. The *initial* configuration $C_0 = (q_0, [[]])$. A configuration $C = (q, W)$ is *accepting* if $q \in F$. We write $C \overset{\tau}{\Rightarrow} C'$ for configurations $C = (q, W)$, $C' = (q', W')$ and transition $\tau = (q, a, \mathsf{op}(s), q')$, if $W' = \mathsf{op}(s)(W)$. A *run* ρ of $\mathcal{H}$ is an alternating sequence of configurations and transitions, starting from the initial configuration, and conforming to the relation $\Rightarrow$, i.e., $\rho = C_0 \overset{\tau_1}{\Rightarrow} C_1 \overset{\tau_2}{\Rightarrow} C_2 \dots \overset{\tau_n}{\Rightarrow} C_n$. We say that ρ is an *accepting run* if C_n is accepting.

Next, we aim at understanding the linear behaviours of 2-CPDS as order-2 nested words (2-NW). For instance, $\mathcal{N}_1$ and $\mathcal{N}_2$ of Figure 1 are generated, respectively, by $\mathcal{H}_1$ and $\mathcal{H}_2$ above. We give the formal definition below.

[1] Collapse links to order-1 stack symbols, or pushes without collapse links are not considered for simplicity. These are, however, easy to simulate thanks to $\downarrow_1$ and the top-test.

Order-2 nested words (2-NW). We propose words augmented with nesting relations to capture the behaviours of 2-CPDS, analogous to nested words for pushdown systems. We have two nesting relations $\curvearrowright^1$ and $\curvearrowright^2$, for order-1 stacks and order-2 stacks respectively. For each position j executing $\downarrow_1$, we find the position i at which the popped symbol was pushed and we link these matching positions with $i \curvearrowright^1 j$. Notice that since a stack may be duplicated multiple times, a $\uparrow_1$ event may have multiple $\downarrow_1$ partners. For instance, in $\mathcal{N}_2$ above, the $\uparrow_1$ at position 4 is matched by the $\downarrow_1$ at positions 6,12,19. Similarly, $\curvearrowright^2$ links $\uparrow_2$ events with matching $\downarrow_2$ events. Every $\uparrow_2$ event may have at most one $\curvearrowright^2$ partner, since each pushed stack is popped at most once. A $\Downarrow$ event is seen as popping several stacks in one go, hence, several $\uparrow_2$ events may be linked to single $\Downarrow$ event by $\curvearrowright^2$ relation. For instance, in $\mathcal{N}_1$ above the collapse at position 11 pops the stacks pushed at positions 2,4,6. Thus an order-2 nested word (2-NW) over an alphabet Σ is a tuple $\mathcal{N} = \langle w, \curvearrowright^1, \curvearrowright^2 \rangle$ where w is a word over Σ, and $\curvearrowright^1$ and $\curvearrowright^2$ are binary relations over positions of w.

The language of a 2-CPDS over an alphabet Σ is a set of 2-NW over Σ generated by accepting runs. It is denoted $\mathcal{L}(\mathcal{H})$. When depicting 2-NWs, we sometimes do not indicate the labelling by the finite alphabet, but often indicates the type of the stack operation. When we express the letter of the alphabet and the operation, we just write them next to each other. For example $a\uparrow_1^s$ would mean that the label is a, and that position performs $\uparrow_1^s$.

Given a generating sequence $\mathsf{op}_0\mathsf{op}_1 \cdots \mathsf{op}_n \in \mathsf{Op}^+$ of operations, either it is not valid, or there are unique $\curvearrowright^1$ and $\curvearrowright^2$ which conform to the order-2 stack policy. To characterize these relations, we define the position $\mathsf{push}_1(n)$ at which the current (after op_n) top stack symbol was pushed and the position $\mathsf{Push}_2(n)$ at which the current top order-1 stack was pushed/duplicated. We let $\mathsf{push}_1(n) = -1$ if the top (order-1) stack is empty. We let $\mathsf{Push}_2(n) = -1$ if the order-2 stack contains only one order-1 stack. For instance, with the sequence generating $\mathcal{N}_2$ we have $\mathsf{Push}_2(5) = 5 = \mathsf{Push}_2(7)$, $\mathsf{Push}_2(4) = -1 = \mathsf{Push}_2(8)$, $\mathsf{push}_1(4) = 4 = \mathsf{push}_1(5) = \mathsf{push}_1(8)$ and $\mathsf{push}_1(0) = 0 = \mathsf{push}_1(3) = \mathsf{push}_1(6)$ and $\mathsf{push}_1(2) = -1 = \mathsf{push}_1(7)$. Also, in the 2-NW $\mathcal{N}_1$ we have $\mathsf{push}_1(11) = 1$ and $\mathsf{Push}_2(11) = 0$.

Surprisingly, push_1 and Push_2 can be characterized by a context-free grammar. We denote by L_1 and L_2 the languages defined by the non-terminals S_1 and S_2 of the following grammar:

$$\begin{array}{lcl} S_1 & \to & \uparrow_1 \mid S_1\uparrow_2 \mid S_1S_1\downarrow_1 \mid S_1S_2\downarrow_2 \mid S_1S_2S_1\Downarrow \\ S_2 & \to & \uparrow_2 \mid S_2\uparrow_1 \mid S_2\downarrow_1 \mid S_2S_2\downarrow_2 \mid S_2S_2S_1\Downarrow . \end{array}$$

Proposition 1. Let $\mathsf{op}_0\mathsf{op}_1 \cdots \mathsf{op}_n \in \mathsf{Op}^+$ be a valid push/pop/collapse sequence. Then, for all $0 \leqslant i \leqslant j \leqslant n$ we have (proof in Appendix A)
P1. $\mathsf{push}_1(j) = i$ iff $\mathsf{op}_i \cdots \mathsf{op}_j \in L_1$,
P2. $\mathsf{Push}_2(j) = i$ iff $\mathsf{op}_i \cdots \mathsf{op}_j \in L_2$,
P3. $\mathsf{push}_1(j) = -1$ iff $\mathsf{op}_k \cdots \mathsf{op}_j \notin L_1$ for all $0 \leqslant k \leqslant j$,
P4. $\mathsf{Push}_2(j) = -1$ iff $\mathsf{op}_k \cdots \mathsf{op}_j \notin L_2$ for all $0 \leqslant k \leqslant j$.

This characterization will be crucial in the rest of the paper, to justify correctness of both formulas in Section 3, and also tree-automata constructions in our decision procedure. Also, it yields a linear time algorithm `Nestify` (`http://www.lsv.fr/~gastin/hopda`).

3 PDL and MSO over order-2 nested words

We introduce two logical formalisms for specifications over 2-NW. The first one is propositional dynamic logic which essentially navigates through the edges of a 2-NW, checking positional properties on the way. The second one is the yardstick monadic second-order logic, which extends MSO over words with the $\curvearrowright^1$ and $\curvearrowright^2$ binary relations.

Propositional dynamic logic was originally introduced in [18] to study the branching behaviour of programs. Here we are not interested in the branching behaviour. Instead we study the linear time behaviours (words) enriched with the nesting relations. Since these are graphs, we take advantage of the path formulas of PDL based on regular expressions to navigate in the 2-NWs. This is in the spirit of [3, 4, 10, 22] where PDL was used to specify properties of graph structures such as message sequence charts or multiply nested words.

Propositional Dynamic Logic with converse and loop (LCPDL) can express properties of nodes (positions) as boolean combinations of the existence of paths and loops. Paths are built using regular expressions over the edge relations (and their converses) of order-2 nested words. The syntax of the node formulas φ and path formulas π of LCPDL are given by

$$\begin{array}{lcl} \varphi & ::= & a \mid \varphi \vee \varphi \mid \neg\varphi \mid \langle\pi\rangle\varphi \mid \mathsf{Loop}(\pi) \\ \pi & ::= & \{\varphi\}? \mid \rightarrow \mid \leftarrow \mid \curvearrowright^1 \mid \curvearrowleft^1 \mid \curvearrowright^2 \mid \curvearrowleft^2 \mid \pi\cdot\pi \mid \pi+\pi \mid \pi^* \end{array}$$

where $a \in \Sigma$. The node formulas are evaluated on positions of an order-2 nested word, whereas path formulas are evaluated on pairs of positions. We give the semantics below (i, j, i', j' vary over positions of a 2-NW $\mathcal{N} = \langle w, \curvearrowright^1, \curvearrowright^2\rangle$):

$$\begin{array}{rcl} \mathcal{N}, i \models a & \text{if} & i\text{th letter of } w \text{ is } a \\ \mathcal{N}, i \models \varphi_1 \vee \varphi_2 & \text{if} & \mathcal{N}, i \models \varphi_1 \text{ or } \mathcal{N}, i \models \varphi_2 \\ \mathcal{N}, i \models \neg\varphi & \text{if} & \text{it is not the case that } \mathcal{N}, i \models \varphi \\ \mathcal{N}, i \models \langle\pi\rangle\varphi & \text{if} & \mathcal{N}, i, j \models \pi \text{ and } \mathcal{N}, j \models \varphi \text{ for some j} \\ \mathcal{N}, i \models \mathsf{Loop}(\pi) & \text{if} & \mathcal{N}, i, i \models \pi \\ \mathcal{N}, i, j \models \{\varphi\}? & \text{if} & i = j \text{ and } \mathcal{N}, i \models \varphi \\ \mathcal{N}, i, j \models \rightarrow & \text{if} & j \text{ is the successor position of } i \text{ in the word } w \\ \mathcal{N}, i, j \models \leftarrow & \text{if} & \mathcal{N}, j, i \models \rightarrow \\ \mathcal{N}, i, j \models \curvearrowright^1 & \text{if} & i \curvearrowright^1 j \text{ in the 2-NW } \mathcal{N} \\ \mathcal{N}, i, j \models \curvearrowleft^1 & \text{if} & \mathcal{N}, j, i \models \curvearrowright^1 \\ \mathcal{N}, i, j \models \curvearrowright^2 & \text{if} & i \curvearrowright^2 j \text{ in the 2-NW } \mathcal{N} \\ \mathcal{N}, i, j \models \curvearrowleft^2 & \text{if} & \mathcal{N}, j, i \models \curvearrowright^2 \\ \mathcal{N}, i, j \models \pi_1\cdot\pi_2 & \text{if} & \text{there is a position } k \text{ such that } \mathcal{N}, i, k \models \pi_1 \text{ and } \mathcal{N}, k, j \models \pi_2 \\ \mathcal{N}, i, j \models \pi_1 + \pi_2 & \text{if} & \mathcal{N}, i, j \models \pi_1 \text{ or } \mathcal{N}, i, j \models \pi_2 \\ \mathcal{N}, i, j \models \pi^* & \text{if} & \text{there exist positions } i_1, \ldots, i_n \text{ for some } n \geqslant 1 \\ & & \text{such that } i = i_1,\ j = i_n \text{ and } \mathcal{N}, i_m, i_{m+1} \models \pi \text{ for all } 1 \leqslant m < n \end{array}$$

An LCPDL *sentence* is a boolean combination of atomic sentences of the form $\mathsf{E}\varphi$. An atomic LCPDL sentence is evaluated on an order-2 nested word $\mathcal{N}$. We have $\mathcal{N} \models \mathsf{E}\varphi$ if there exists a position i of $\mathcal{N}$ such that $\mathcal{N}, i \models \varphi$.

We use abbreviations to include true, false, conjunction, implication, 'φ holds after all π paths' ($[\pi]\varphi$) etc. We simply write $\langle\pi\rangle$ instead of $\langle\pi\rangle\mathsf{true}$ to check the existence of a π path from the current position. In particular, we can check the type of a node with $\mathsf{ispush}_1 = \langle\curvearrowright^1\rangle$, $\mathsf{ispop}_1 = \langle\curvearrowleft^1\rangle$, and similarly for ispush_2 and ispop_2. Notice that a collapse node satisfies ispop_2. Also, $\mathsf{A}\varphi = \neg\mathsf{E}\neg\varphi$ states that φ holds on all nodes of the 2-NW.

Example 2. We give now path formulas corresponding to the functions push_1 and Push_2 defined at the end Section 2. We use the characterization of Proposition 1. Consider first the macro $\mathsf{isfirstpush}_2 = \mathsf{ispush}_2 \wedge \neg\mathsf{Loop}(\leftarrow^+ \cdot \curvearrowright^2 \cdot \curvearrowleft^2)$ which identifies uniquely the target of a collapse. Then, the deterministic path formulas for push_1 and Push_2 are given by

$$\pi_{\mathsf{push}_1} = (\{\mathsf{ispush}_2\}? \cdot \leftarrow + \curvearrowleft^1 \cdot \leftarrow + \curvearrowleft^2 \cdot \{\mathsf{isfirstpush}_2\}? \cdot \leftarrow)^* \cdot \{\mathsf{ispush}_1\}?$$
$$\pi_{\mathsf{Push}_2} = (\{\mathsf{ispush}_1\}? \cdot \leftarrow + \{\mathsf{ispop}_1\}? \cdot \leftarrow + \curvearrowleft^2 \cdot \{\mathsf{isfirstpush}_2\}? \cdot \leftarrow)^* \cdot \{\mathsf{ispush}_2\}?$$

The matching push of an order-1 pop should coincide with the one dictated by π_{push_1} starting from the previous node, i.e., before the top symbol was popped. The situation is similar for an order-2 pop. Hence, the following sentence states that $\curvearrowright^1$ and $\curvearrowright^2$ are well-nested.

$$\phi_{\mathsf{wn}} = \mathsf{A}\left((\mathsf{ispop}_1 \implies \mathsf{Loop}(\leftarrow \cdot \pi_{\mathsf{push}_1} \cdot \curvearrowright^1)) \wedge (\mathsf{ispush}_2 \implies \mathsf{Loop}(\curvearrowright^2 \cdot (\leftarrow \cdot \pi_{\mathsf{Push}_2})^+))\right)$$

The *satisfiability problem* SAT(LCPDL) asks: Given an LCPDL sentence ϕ, does there exist a 2-NW $\mathcal{N}$ such that $\mathcal{N} \models \phi$? The *model checking problem* MC(LCPDL) asks, given an LCPDL sentence ϕ and a 2-CPDS $\mathcal{H}$, whether $\mathcal{N} \models \phi$ for all 2-NW $\mathcal{N}$ in $\mathcal{L}(\mathcal{H})$.

Theorem 3. *The problems* SAT(LCPDL) *and* MC(LCPDL) *are both undecidable, even for 2-NW (or 2-CPDS) without collapse and order-2 stacks of bounded height.*

Proof. Notice that the 2-NWs generated by the non-collapsible 2-CPDS $\mathcal{H}_2$ (cf. page 3) embed larger and larger half-grids. For instance, the 2-NW $\mathcal{N}_2$ of Figure 1 embeds a half-grid of size four. The lines are embedded within $\curvearrowright^2$: 2, then 6,7, then 11,12,13, and finally 17,18,19,20. Moving right in the grid amounts to moving right in the 2-NW, without crossing a $\downarrow_2$. Moving down in the grid (e.g., from 6) amounts to going to the next order-1 pop $\downarrow_1$ (which is 12) attached to the same order-1 push $\uparrow_1$ (which is 4).

To prove the undecidability, we encode, in LCPDL, the computation of a Turing Machine on the half-grid embedded in the 2-NW $\mathcal{N}_2$. First, we write a formula grid stating that the 2-NW is of the correct form. We use $\mathsf{empty}_1 = \langle \curvearrowleft^1 \rangle \neg \langle \leftarrow \rangle$ to state that the $\uparrow_1$ matching the current $\downarrow_1$ is the first event of the 2-NW, hence the top order-1 stack is empty after the current $\downarrow_1$. Then, $\mathsf{grid}_1 = \mathsf{E}(\mathsf{ispush}_1 \wedge \neg\langle \leftarrow \rangle)$ states that the first event is a $\uparrow_1$. Next,

$$\mathsf{grid}_2 = \mathsf{A}(\mathsf{ispush}_1 \implies \langle \rightarrow \cdot \{\mathsf{ispush}_2\}? \cdot (\rightarrow \cdot \{\mathsf{ispop}_1\}?)^+ \cdot \{\mathsf{empty}_1\}? \cdot \rightarrow \cdot \{\mathsf{ispop}_2\}? \rangle)$$

states that the successor of every $\uparrow_1$ is a $\uparrow_2$ followed by a sequence of $\downarrow_1$ (the line of the grid) emptying the top order-1 stack, followed by a $\downarrow_2$ which restores the order-1 stack. Finally, $\mathsf{grid}_3 = \mathsf{A}(\mathsf{ispop}_2 \implies \neg\langle \rightarrow \rangle \vee \langle \rightarrow \cdot \curvearrowright^1 \rangle)$ states that a $\downarrow_2$ is either the last event of the 2-NW, or is followed by a $\uparrow_1$ starting a new line in the grid. One can check that a 2-NW $\mathcal{N}$ satisfies $\mathsf{grid} = \mathsf{grid}_1 \wedge \mathsf{grid}_2 \wedge \mathsf{grid}_3$ iff it is of the form of the 2-NW depicted above.

We can almost interpret in LCPDL the half-grid in a 2-NW $\mathcal{N}$ satisfying grid. Nodes of the grid correspond to $\downarrow_1$ events. Moving right in the line of the grid corresponds to the path expression $\rightarrow \cdot \{\mathsf{ispop}_1\}?$ and similarly for going left. Moving down in the half-grid (e.g., from 6 to 12 in $\mathcal{N}_2$), corresponds to going to the next-pop-from-same-push in the 2-NW. We do not know whether the next-pop relation, denoted $\hookrightarrow$, can be written as a path expression in LCPDL. But we have a macro for checking a node formula φ at the next-pop:

$$\langle \hookrightarrow \rangle \varphi ::= \mathsf{Loop}((\{\mathsf{ispop}_1\}? \cdot \rightarrow)^+ \cdot \{\mathsf{ispop}_2\}? \cdot \rightarrow \cdot \rightarrow \cdot (\rightarrow \cdot \{\mathsf{ispop}_1\}?)^+ \cdot \{\varphi\}? \cdot \curvearrowleft^1 \cdot \curvearrowright^1)$$

With this, we can write an LCPDL formula to encode the computation of a Turing machine starting from the empty configuration. Consecutive lines of the grid correspond to consecutive configurations. For instance, to check that a transition $(p, a, q, b, \rightarrow)$ of the Turing machine is applied at some node, we write the formula $(p \wedge \langle \rightarrow \rangle a) \implies \langle \hookrightarrow \rangle (b \wedge \langle \rightarrow \rangle q)$.

We deduce that SAT(LCPDL) is undecidable. Since the 2-CPDS $\mathcal{H}_2$ on page 3 generates all 2-NW satisfying the grid formula, we deduce that MC(LCPDL) is also undecidable. ◻

Monadic Second-order Logic over 2-NW, denoted MSO, extends the classical MSO over words with two binary predicates $\curvearrowright^1$ and $\curvearrowright^2$. A formula ϕ can be written using the syntax:

$$\phi ::= a(x) \mid x < y \mid x \curvearrowright^1 y \mid x \curvearrowright^2 y \mid \phi \vee \phi \mid \neg\phi \mid \exists x\, \phi \mid \exists X \phi \mid x \in X$$

where $a \in \Sigma$, x, y are first-order variables and X is a second-order variable. The semantics is as expected. As in the case of LCPDL, we use common abbreviations.

Example 4. The binary relation $\hookrightarrow$ which links consecutive pops matching the same push can be easily expressed in the first-order fragment as

$$\phi_{\hookrightarrow}(x, y) = \exists z \left(z \curvearrowright^1 x \wedge z \curvearrowright^1 y \wedge \neg\exists z' \left(z \curvearrowright^1 z' \wedge x < z' < y\right)\right) .$$

Example 5. The set of all 2-NW can be characterised in MSO. It essentially says that $<$ is a total order, and that the matching relations are valid. For the latter, we first state that matching relations are compatible with the linear order, and that they are disjoint in the following sense: the target of a $\curvearrowright^1$ (resp. the source of a $\curvearrowright^2$) is not part of another matching, and the source of a $\curvearrowright^1$ (resp. the target of a $\curvearrowright^2$) is not part of $\curvearrowright^2$ (resp. $\curvearrowright^1$). Finally, to state that $\curvearrowright^1$ and $\curvearrowright^2$ are well-nested, we take the idea from Example 2.

Example 6. Given a 2-CPDS $\mathcal{H}$, its language $\mathcal{L}(\mathcal{H})$ can be characterised by an MSO formula. The formula essentially guesses the transitions taken at every position using second-order variables and verifies that this guess corresponds to a valid accepting run. The only difficulty is to check that top-tests are satisfied. If the transition guessed for position x contains $\mathsf{top}(s)$ then we find position y corresponding to $\mathsf{push}_1(x)$ (expressible by an MSO formula equivalent of the formula π_{push_1}) and check that the transition guessed at position y contains $\uparrow_1^s$.

The *satisfiability problem* SAT(MSO) asks, given an MSO sentence ϕ, whether $\mathcal{N} \models \phi$ for some 2-NW $\mathcal{N}$. The *model checking problem* MC(MSO) asks, given an MSO sentence ϕ and a 2-CPDS $\mathcal{H}$, whether $\mathcal{N} \models \phi$ for all 2-NW $\mathcal{N} \in \mathcal{L}(\mathcal{H})$. Since LCPDL can be expressed in MSO, we deduce from Theorem 3 that

Theorem 7. *The problems* SAT(MSO) *and* MC(MSO) *are undecidable.*

Remark. Configuration graphs of 2-CPDS render MSO undecidable [21]. For instance, the 2-CPDS $\mathcal{H}_1$ (cf. page 3) embeds an infinite half-grid in its configuration graph (see, App B). Notice that 2-NW generated by $\mathcal{H}_1$ (for instance, $\mathcal{N}_1$ of Figure 1) does not represent the configuration graph, but rather some path in it, with extra matching information.

4 Eliminating collapse

We can reduce the satisfiability and model checking problems to variants where there are no collapse operations. The idea is to simulate a collapse with a sequence of order-2 pops ($\downarrow_2$). These pops will be labelled by a special symbol # so that we do not confuse it with a normal order-2 pop. An internal node is added before such a sequence which indicates the label of the collapse node (see Figure 2). Surprisingly, in LCPDL/MSO we can express that the number of $\#\downarrow_2$ in the sequence is correct. We give the details below.

2-NW to 2-NW without collapse. We expand every collapse node labelled a by a sequence of the form: $\underline{a}(\#\downarrow_2)^+$, with the intention that if we merge all the nodes in this sequence into a single node labelled by $a\Downarrow$, we obtain the original 2-NW back. Here $\#\downarrow_2$ is a single position, which is labelled #, and is the target of a $\curvearrowright^2$ edge. Notice that the

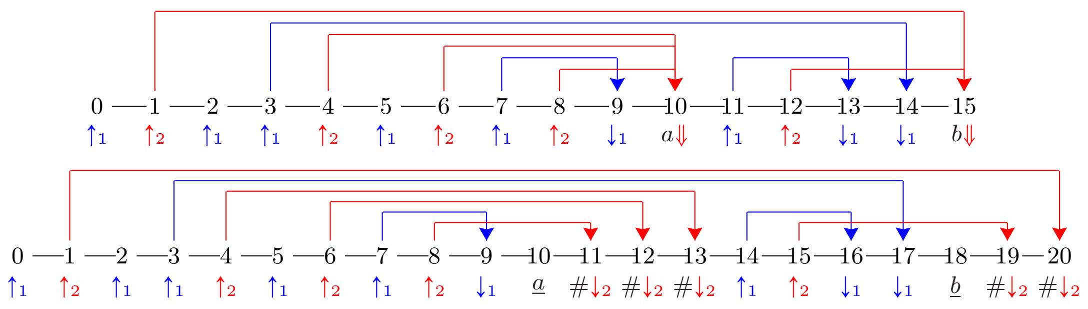

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
↑1 ↑2 ↑1 ↑1 ↑2 ↑1 ↑2 ↑1 ↑2 ↓1 a⇓ ↑1 ↑2 ↓1 ↓1 b⇓
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
↑1 ↑2 ↑1 ↑1 ↑2 ↑1 ↑2 ↑1 ↑2 ↓1 a #↓2 #↓2 #↓2 ↑1 ↑2 ↓1 ↓1 b #↓2 #↓2

Figure 2 A 2-NW (top) and its encoding in 2-NW without collapse (below). We show the labels only on the nodes of interest.

number of $\#{\downarrow_2}$'s needed for such an encoding is not bounded. We will ensure, with the help of LCPDL/MSO, that the encoding has the precise number of $\#{\downarrow_2}$'s needed.

2-CPDS to 2-CPDS without collapse. Given an 2-CPDS $\mathcal{H}$, we construct a new 2-CPDS $\mathcal{H}'$ where, for each collapse transition $t = (q, a, \Downarrow, q')$ there is an extra state q_t. Further, instead of the transition t we have the following three transitions: $(q, \underline{a}, \mathsf{Nop}, q_t)$, $(q_t, \#, \downarrow_2, q_t)$, and $(q_t, \#, \downarrow_2, q')$. Notice that for every 2-NW in the language of $\mathcal{H}$, its encoding without collapse will be in the language of $\mathcal{H}'$. However, the language of $\mathcal{H}'$ may contain spurious runs where the $(q_t, \#, \downarrow_2, q_t)$ is iterated an incorrect number of times. These spurious runs will be discarded by adding a precondition to the specification.

LCPDL to LCPDL without collapse. We identify a collapse node by the first node in a block of the form $\underline{a}(\#{\downarrow_2})^+$. We call the positions labelled by symbols other than # *representative positions*. Intuitively, we will be evaluating node formulas only at representative positions, and path formulas connect a pair of representative positions. Checking whether the current node is labelled a, would now amount to checking whether the current node is labelled by a or $\underline{a}$. Further, in path formulas, moving to right ($\rightarrow$) would mean going to the next representative position in the right. Similarly for $\leftarrow$. The path formula $\curvearrowright^2$ would correspond to taking the $\curvearrowright^2$ edge and moving left until a representative position is reached. Notice that $\curvearrowleft^2$ at a collapse node can non-deterministically choose any $\uparrow_2$ matched to it. Hence we express this as $(\rightarrow\cdot\{\#\}?)^*\cdot\curvearrowleft^2$. Notice that this formula also handles the case when the current node is not of the form $\underline{a}$. The other modalities remain unchanged. This translation can be done in linear time and the size of the resulting formula is linear in the size of the original formula. Translation of a LCPDL node formula φ and path formula π to LCPDL without collapse is given below. The translation is denoted $\overline{\varphi}$ and $\overline{\pi}$ respectively.

$$\begin{array}{rcl@{\qquad}rcl@{\qquad}rcl}
\overline{a} & \equiv & a \vee \underline{a} & \overline{\{\varphi\}?} & \equiv & \{\overline{\varphi}\}? & \overline{\curvearrowleft^1} & \equiv & \curvearrowleft^1 \\
\overline{\varphi_1 \vee \varphi_2} & \equiv & \overline{\varphi_1} \vee \overline{\varphi_2} & \overline{\pi_1 \cdot \pi_2} & \equiv & \overline{\pi_1} \cdot \overline{\pi_2} & \overline{\rightarrow} & \equiv & \rightarrow \cdot (\{\#\}? \cdot \rightarrow)^* \cdot \{\neg\#\}? \\
\overline{\neg\varphi} & \equiv & \neg\overline{\varphi} & \overline{\pi_1 + \pi_2} & \equiv & \overline{\pi_1} + \overline{\pi_2} & \overline{\leftarrow} & \equiv & \leftarrow \cdot (\{\#\}? \cdot \leftarrow)^* \cdot \{\neg\#\}? \\
\overline{\langle\pi\rangle\varphi} & \equiv & \langle\overline{\pi}\rangle\overline{\varphi} & \overline{\pi^*} & \equiv & \overline{\pi}^* & \overline{\curvearrowleft^2} & \equiv & (\rightarrow \cdot \{\#\}?)^* \cdot \curvearrowleft^2 \\
\overline{\mathsf{Loop}(\pi)} & \equiv & \mathsf{Loop}(\overline{\pi}) & \overline{\curvearrowright^1} & \equiv & \curvearrowright^1 & \overline{\curvearrowright^2} & \equiv & \curvearrowright^2 \cdot (\{\#\}? \cdot \leftarrow)^* \cdot \{\neg\#\}?
\end{array}$$

MSO to MSO without collapse. Translation of MSO is similar in spirit to that of LCPDL. Every atomic formula (binary relations and unary predicates) is translated as done in the case of LCPDL. The rest is then a standard relativisation to the representative positions. Overloading notations, we denote the translation of an MSO formula φ by $\overline{\varphi}$.

Identifying valid encodings with LCPDL/MSO. First we need to express that the number of $\#{\downarrow_2}$ is correct. Towards this, we will state that, the matching push of the last pop in a $\underline{a}(\#{\downarrow_2})^+$ block indeed corresponds to the push of the stack in which the topmost stack symbol was pushed. We explain this below. From a representative position labelled $\underline{a}$ we can move to the ${\uparrow_1}$ position x where the topmost stack symbol of the topmost stack was pushed, by taking a π_{push_1} path. The ${\uparrow_2}$ position y which pushed the stack which contains the element pushed at x can be reached by a π_{Push_2} path from x. We then say that the matching pop of y is indeed the last $\#{\downarrow_2}$ labelled node in the sequence, and that it indeed belongs to the very sequence of $\underline{a}$ we started with. We can state this in LCPDL with:

$$\phi_{\mathsf{valid}} = \mathsf{A} \bigwedge_{a \in \Sigma} \left(\underline{a} \implies \mathsf{Loop}(\pi_{\mathsf{push}_1} \cdot \pi_{\mathsf{Push}_2} \cdot \curvearrowright^2 \cdot \{\neg\langle\rightarrow\rangle\#\}? \cdot (\{\#\}? \cdot \leftarrow)^+)\right) .$$

Satisfiability and model checking problems. The satisfiability problem of LCPDL/MSO formula ϕ over 2-NW with collapse reduces to the satisfiability problem of $\phi_{\mathsf{valid}} \wedge \overline{\phi}$ over 2-NW without collapse. The model checking problem of 2-CPDS $\mathcal{H}$ against a LCPDL/MSO formula ϕ reduces to the model checking problem of $\mathcal{H}'$ against $\phi_{\mathsf{valid}} \implies \overline{\phi}$.

Remark. In [1] it is shown that for every 2-CPDS there exists an order-2 pushdown system without collapse generating the same word language (without nesting relations). Our result of this section shows that, model checking and satisfiability checking, even in the presence of nesting relations, can be reduced to the setting without collapse.

5 Bounded-pop 2-NW

In this section we will define an under-approximation of 2-NWs which regains decidability of the verification problems discussed in Section 3.

Bounded-pop 2-NWs are 2-NWs in which a pushed symbol may be popped at most a bounded number of times. This does not limit the number of times an order-1 stack may be copied, nor the height of any order-1 or order-2 stack. Our restriction amounts to bounding the number of $\curvearrowright^1$ partners that a push may have. Let β denote this bound for the rest of the paper. The class of 2-NW in which every order-1 push has at most β many matching pops is called *β-pop-bounded order-2 nested words*. It is denoted 2-NW(β).

Bounded-pop model checking The under-approximate satisfiability problem and model checking problems are defined as expected. The problem SAT(LCPDL, β) (resp. SAT(MSO, β)) asks, given an LCPDL (resp. MSO) sentence ϕ and a natural number β, whether $\mathcal{N} \models \phi$ for some 2-NW $\mathcal{N} \in$ 2-NW(β). The problem MC(LCPDL, β) (resp. MC(MSO, β)) asks, given an LCPDL (resp. MSO) sentence ϕ, a 2-CPDS $\mathcal{H}$ and a natural number β, whether that $\mathcal{N} \models \phi$ for all 2-NW $\mathcal{N} \in \mathcal{L}(\mathcal{H}) \cap$ 2-NW(β).

Theorem 8. *The problems* SAT(LCPDL, β) *and* MC(LCPDL, β) *are* ExpTime*-Complete. The problems* SAT(MSO, β) *and* MC(MSO, β) *are decidable.*

2-CPDS can simulate nested-word automata (NWA) [7] by not using any order-1 stack operations. ${\uparrow_2}$ and ${\downarrow_2}$ will play the role of push and pop of NWA. Satisfiability of PDL over nested words, and model checking of PDL against NWA are known to be ExpTime-Complete [8,9]. The ExpTime-hardness of SAT(LCPDL, β) and MC(LCPDL, β) follows. The decision procedures establishing Theorem 8 are given in the next section. Thanks to Section 4 we will restrict our attention to 2-NW without collapse.

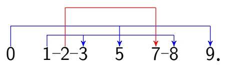
6 Split-width and decision procedures

In all of this section, by 2-NWs we mean order-2 nested words *without collapse*. We lift the notion of split-decomposition and split-width to 2-NWs and show that words in 2-NW(β) have split-width bounded by $2\beta + 2$. Then we show that nested words in 2-NW(β) can be interpreted in binary trees, which is the core of our decision procedures.

A Split-2-NW is a 2-NW in which the underlying word has been split in several factors. Formally, a split-2-NW is a tuple $\overline{\mathcal{N}} = \langle u_1, \dots, u_m, \curvearrowright^1, \curvearrowright^2\rangle$ such that $\mathcal{N} = \langle u_1 \cdots u_m, \curvearrowright^1, \curvearrowright^2\rangle$ is a 2-NW. The number m of factors in a split-2-NW is called it's width. A 2-NW is a split-2-NW of width one.

A split-2-NW can be seen as a labelled graph whose vertices are the positions of the underlying word (concatenation of the factors) and we have order-1 edges $\curvearrowright^1$, order-2 edges $\curvearrowright^2$ and successor edges $\rightarrow$ between consecutive positions *within* a factor. We say that a split-2-NW is connected if the underlying graph is *connected*. If a split-2-NW is not connected, then its connected components form a partition of its factors.

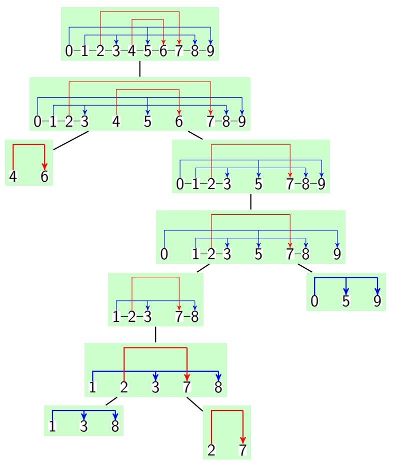
Example 9. Consider the split-2-NW on the right. It has two connected components, and its width is five.



The split game is a two-player turn based game $\mathcal{G} = \langle V = V_\forall \uplus V_\exists, E\rangle$ where Eve's positions $V_\exists$ consists of (connected or not) split-2-NWs, and Adam's positions $V_\forall$ consists of non connected split-2-NWs. The edges E of $\mathcal{G}$ reflect the moves of the players. Eve's moves consist in removing some successor edges in the graph, i.e., splitting some factors, so that the resulting graph is not connected. Adam's moves amounts to choosing a connected component. A split-2-NW is atomic if it is connected and all its factors are singletons. An atomic split-2-NW contains either a single internal event, or, a single push with all its corresponding pops. A play on a split-2-NW $\overline{\mathcal{N}}$ is path in $\mathcal{G}$ starting from $\overline{\mathcal{N}}$ to an atomic split-2-NW. The cost of the play is the maximum width of any split-2-NW encountered in the path. Eve's objective is to minimize the cost and Adam's objective is to maximize the cost.

A strategy for Eve from a split-2-NW $\overline{\mathcal{N}}$ can be described with a *split-tree* T which is a binary tree labelled with split-2-NW satisfying:

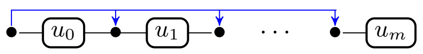
1. The root is labelled by $\overline{\mathcal{N}}$ and leaves are labelled by atomic split-2-NW.
2. Eve's move: Each unary node is labelled with some split-2-NW $\overline{\mathcal{N}}$ and its child is labelled with $\overline{\mathcal{N}}'$ obtained by splitting some factors of $\overline{\mathcal{N}}$.
3. Adam's move: Each binary node is labelled with some non connected split-2-NW $\overline{\mathcal{N}} = \overline{\mathcal{N}}_1 \uplus \overline{\mathcal{N}}_2$ where $\overline{\mathcal{N}}_1$, $\overline{\mathcal{N}}_2$ are the labels of its children. Note that $\mathsf{width}(\overline{\mathcal{N}}) = \mathsf{width}(\overline{\mathcal{N}}_1) + \mathsf{width}(\overline{\mathcal{N}}_2)$.



The *width* of a split-tree T, denoted $\mathsf{width}(T)$, is the maximum width of the split-2-NWs labelling T. The *split-width* of a split-2-NW $\overline{\mathcal{N}}$ is the minimal width of all split-trees for $\overline{\mathcal{N}}$. A split-tree is depicted above. The width of the split-2-NW labelling binary nodes are five. Hence the split-width of the split-2-NW labelling the root is at most five.

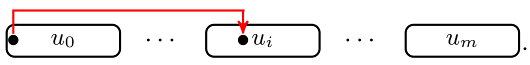
Theorem 10. *Nested words in 2-NW(β) have split-width bounded by $k = 2\beta + 2$.*

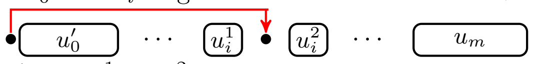
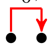
Proof. First, we say that a split-word $\overline{\mathcal{N}} = \langle u_0, u_1, \dots, u_m, \curvearrowright^1, \curvearrowright^2\rangle$ with $m \leqslant \beta$ is *good* if

$\mathsf{Close}(\overline{\mathcal{N}}) =$  is a valid 2-NW (nesting edges between the factors are not depicted). Notice that any 2-NW $\mathcal{N} = \mathsf{Close}(\mathcal{N})$ is vacuously good.

Our strategy is to decompose good split-words into atomic split-words or smaller good split-words so that we can proceed inductively. Good split-words have width at most $\beta + 1$. On decomposing a good split-word to obtain smaller good split-words, we may temporarily generate (not necessarily good) split-words of higher width, but we never exceed $k = 2\beta + 2$.

Consider any good split-word $\overline{\mathcal{N}}$. We have two cases: either it begins with a $\uparrow_1$, or it begins with a $\uparrow_2$.

If $\overline{\mathcal{N}}$ begins with a $\uparrow_2$: $\overline{\mathcal{N}} =$ . We split factors u_0 and u_i to get $\overline{\mathcal{N}'}$ of width at most $m + 4$.

$\overline{\mathcal{N}'} =$  Note that, there cannot be any $\rightsquigarrow^1$ edges from $u'_0, \ldots, u_i^1$ to $u_i^2, \ldots, u_m$ since the duplicated stack is lost at the $\downarrow_2$. Further, there cannot be any $\rightsquigarrow^2$ edges from $u'_0, \ldots, u_i^1$ to $u_i^2, \ldots, u_m$ since $\rightsquigarrow^2$ is well-nested. Hence, $\overline{\mathcal{N}'}$ can be divided into atomic  and split-words $\overline{\mathcal{N}_1} = \boxed{\;u'_0\;} \;\cdots\; \boxed{u_i^1}$ and $\overline{\mathcal{N}_2} = \boxed{u_i^2} \;\cdots\; \boxed{\;u_m\;}$. Since $\overline{\mathcal{N}}$ is good, we can prove that $\overline{\mathcal{N}_1}$ and $\overline{\mathcal{N}_2}$ are also good.

Before moving to the more involved case where $\overline{\mathcal{N}}$ begins with a $\uparrow_1$, we first see a couple of properties of 2-NW which become handy. Examples are given in Appendix C.

The *context-pop* of an order-1 pop at position x, denoted $\mathsf{ctxt\text{-}pop}(x)$, is the position of the order-2 pop of the innermost $\rightsquigarrow^2$ edge that encloses x. If $z \rightsquigarrow^2 y$ and $z < x < y$ and there is no $z' \rightsquigarrow^2 y'$ with $z < z' < x < y' < y$, then $\mathsf{ctxt\text{-}pop}(x) = y$. If x does not have a context-pop, then it is *top-level.*

Consider any 2-NW $\langle a_1 a_2 w, \rightsquigarrow^1, \rightsquigarrow^2\rangle$ beginning with two order-1 pushes with corresponding pops at positions $y_1^1, \ldots, y_m^1$ and $y_1^2, \ldots, y_n^2$ respectively. Then for each y_i^1, there is a y_j^2 with $y_j^2 < y_i^1$ such that either $\mathsf{ctxt\text{-}pop}(y_i^1) \leqslant \mathsf{ctxt\text{-}pop}(y_j^2)$ or y_j^2 is top-level. We call this property *existence of covering pop* for later reference.

Consider any 2-NW $\langle aw, \rightsquigarrow^1, \rightsquigarrow^2\rangle$ beginning with an order-1 push with corresponding pops at positions $y_1, \ldots, y_m$. The *context-suffix* of y_i, denoted by $\mathsf{ctxt\text{-}suffix}(y_i)$ is the factor of w strictly between y_i and $\mathsf{ctxt\text{-}pop}(y_i)$, both y_i and $\mathsf{ctxt\text{-}pop}(y_i)$ not included. If y_i is top-level, then $\mathsf{ctxt\text{-}suffix}(y_i)$ is the biggest suffix of w not including y_i. We can prove that, for each i, $\mathsf{ctxt\text{-}suffix}(y_i)$ is not connected to the remaining of w via $\rightsquigarrow^1$ or $\rightsquigarrow^2$ edges. The notion of $\mathsf{ctxt\text{-}suffix}(y_i)$ may be lifted to split-words $\overline{\mathcal{N}} = \langle au_0, u_1, u_2, \ldots, u_n, \rightsquigarrow^1, \rightsquigarrow^2\rangle$ also. In this case, $\mathsf{ctxt\text{-}suffix}(y_i)$ may contain several factors. Still $\mathsf{ctxt\text{-}suffix}(y_i)$ is not connected to the remaining factors. Moreover, if $\overline{\mathcal{N}}$ is good, so is $\mathsf{ctxt\text{-}suffix}(y_i)$.

Now we are ready to describe the decomposition for the second case where the good split-word $\overline{\mathcal{N}}$ begins with a $\uparrow_1$. Let $\overline{\mathcal{N}} = \langle au_0, u_1, u_2, \ldots u_n, \rightsquigarrow^1, \rightsquigarrow^2\rangle$ beginning with an order-1 push with corresponding pops at positions $y_1, \ldots, y_m$. Note that $n, m \leqslant \beta$. We proceed as follows. We split at most two factors of $\overline{\mathcal{N}} = \overline{\mathcal{N}_m}$ to get $\overline{\mathcal{N}'_m}$ in which $\mathsf{ctxt\text{-}suffix}(y_m)$ is a (collection of) factor(s). Recall that $\mathsf{ctxt\text{-}suffix}(y_m)$ is not connected to other factors. Hence we divide the split-word $\overline{\mathcal{N}'_m}$ to get $\mathsf{ctxt\text{-}suffix}(y_m)$ as a split-word and the remaining as another split-word $\overline{\mathcal{N}_{m-1}}$. Note that $\mathsf{ctxt\text{-}suffix}(y_m)$ is good, so we can inductively decompose it. We proceed with $\overline{\mathcal{N}_{m-1}}$ (which needs not be good). We split at most two factors of $\overline{\mathcal{N}_{m-1}}$ to get $\overline{\mathcal{N}'_{m-1}}$ in which $\mathsf{ctxt\text{-}suffix}(y_{m-1})$ is a (collection of) factor(s). We divide $\overline{\mathcal{N}'_{m-1}}$ to get $\mathsf{ctxt\text{-}suffix}(y_{m-1})$ which is good, and $\overline{\mathcal{N}_{m-2}}$. We proceed similarly on $\overline{\mathcal{N}_{m-2}}$ until we get $\overline{\mathcal{N}_0}$.

Note that the width of $\overline{\mathcal{N}_i}$ (resp. $\overline{\mathcal{N}'_i}$) is at most $n + 1 + m - i$ (resp. $n + 1 + m - i + 2$). Hence the width of this stretch of decomposition is bounded by $n + m + 2$. Since $n, m \leqslant \beta$, the bound $k = 2\beta + 2$ of split-width is not exceeded.

Now we argue that the width of $\overline{\mathcal{N}_1}$ is at most $m+1$. This is where we need the invariant of being good. Consider $\mathcal{N} = \mathsf{Close}(\overline{\mathcal{N}})$, which is a 2-NW since $\overline{\mathcal{N}}$ is good. Now, $\mathcal{N}$ starts with two order-1 pushes. Using the existence of covering pop property, we deduce that every split/hole in $\overline{\mathcal{N}}$ must belong to some $\mathsf{ctxt\text{-}suffix}(y_i)$. Hence, all splits from $\overline{\mathcal{N}}$ are removed in $\overline{\mathcal{N}_0}$, and only m splits corresponding to the removed $\mathsf{ctxt\text{-}suffix}(y_i)$ persist.

We proceed with $\overline{\mathcal{N}_0}$. We make at most $m+1$ new splits to get $\overline{\mathcal{N}'}$ in which the first push and its pops at positions $y_1, \ldots, y_m$ are singleton factors. The width of $\overline{\mathcal{N}'}$ is at most $2m+2 \leqslant 2\beta+2$. Then we divide $\overline{\mathcal{N}'}$ to get atomic split-word consisting of the first push and its pops, and another split-word $\overline{\mathcal{N}''}$. The width of $\overline{\mathcal{N}''}$ is at most $m+1 \leqslant \beta+1$. Further $\overline{\mathcal{N}''}$ is good. By induction $\overline{\mathcal{N}''}$ can also be decomposed. ◻

In Appendix D, we show that 2-NWs of split-width at most k have special tree-width (STW) at most $2k$. We deduce that 2-NWs of bounded split-width can be interpreted in special tree terms (STTs), which are binary trees denoting graphs of bounded STW. Special tree-width and special tree terms were introduced by Courcelle in [15].

A crucial step towards our decision procedures is then to construct a tree automaton $\mathcal{A}^{k-\mathsf{sw}}_{\mathsf{2\text{-}NW}}$ which accepts special tree terms denoting graphs that are 2-NW of split-width at most k. The main difficulty is to make sure that the edge relations $\curvearrowright^1$ and $\curvearrowright^2$ of the graph are well nested. To achieve this with a tree automaton of size $2^{\mathsf{Poly}(k)}$, we use the characterization given by the LCPDL formula ϕ_{wn} of Example 2. Similarly, we can construct a tree automaton $\mathcal{A}^{\beta}_{\mathsf{2\text{-}NW}}$ of size $2^{\mathsf{Poly}(\beta)}$ accepting STTs denoting nested words in 2-NW(β).

Next, we show that for each 2-CPDS $\mathcal{H}$ we can construct a tree automaton $\mathcal{A}^{\beta}_{\mathcal{H}}$ of size $2^{\mathsf{Poly}(\beta,|\mathcal{H}|)}$ accepting STTs denoting nested words in 2-NW(β) which are accepted by $\mathcal{H}$. We deduce that non-emptiness checking of 2-CPDS with respect to 2-NW(β) is in ExpTime.

Finally, for each LCPDL formula ϕ we can construct a tree automaton $\mathcal{A}^{\beta}_{\phi}$ of size $2^{\mathsf{Poly}(\beta,|\phi|)}$ accepting STTs denoting nested words in 2-NW(β) which satisfy ϕ. We deduce that SAT(LCPDL, β) and MC(LCPDL, β) can be solved in ExpTime.

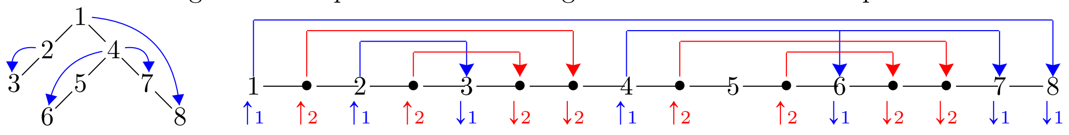
Similarly, for each MSO formula ϕ we can construct a tree automaton $\mathcal{A}^{\beta}_{\phi}$ accepting STTs denoting nested words in 2-NW(β) which satisfy ϕ. We deduce that SAT(MSO, β) and MC(MSO, β) are decidable.

7 Related work

In [11], Broadbent studies nested structures of order-2 HOPDS. A suffix rewrite system that rewrites nested words is used to capture the graph of ϵ-closure of an order-2 HOPDS. The objective of the paper as well as the use of nested words is different from ours.

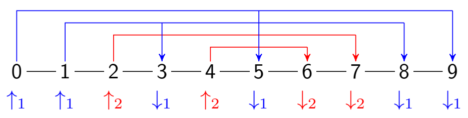
Nested trees. 2-NWs have close relation with nested trees. A nested tree [5,6] is a tree with an additional binary relation such that every branch forms a well-nested word [7]. It provides a "visible" representation of the branching behaviour of a pushdown system.

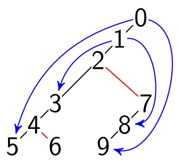
Every finite nested tree can be embedded inside a 2-NW without collapse. In our encoding, order-1 matching relation captures the nesting relation in the nested tree, and order-2 matching relation captures the branching structure. See the example below:



(Encoding of) every left sub-tree is enclosed within a $\bullet \curvearrowright^2 \bullet$ pair, whose source and target nodes are not part of the original nested-tree. There is a bijective correspondence between the other nodes in the 2-NW and nodes in the nested-tree, and the edges in the nested-tree can be easily MSO-interpreted in the 2-NW. This implies that MSO over 2-NW is undecidable, since it is undecidable over nested-trees [6].

Conversely, every 2-NW can be MSO interpreted in a nested-tree. We may assume that 2-NW is collapse-free (cf. Section 4). The $\downarrow_2$ nodes are linked in the tree as the right child of the matching push, and the other nodes are linked as the left child of its predecessor.





On nested trees, μ-calculus was shown to be decidable in [5]. First-order logic over nested trees, when the signature does not contain the order $<$ (but only the successor $\rightarrow$) is shown decidable in [23], but MSO over nested trees is undecidable [5]. Our under-approximation of bounded-pop which regains decidability corresponds to bounding the number of matching pops that a tree-node can have, which in turn bounds the degree of the nodes in the tree.

8 Conclusion

In this paper we study the linear behaviour of a 2-CPDS by giving extra structure to words. The specification formalisms can make use of this structure to describe properties of the system. This added structure comes with the cost of undecidable verification problems. We identify an under approximation that regains decidability for verification problems. Our decision procedure makes use of the split-width technique.

This work is a first step towards further questions that must be investigated. One direction would be to identify other under-approximations which are orthogonal / more lenient than bounded-pop for decidability. Whether similar results can be obtained for order-n CPDS is also another interesting future work. The language theory of CPDS where the language consists of nested-word like structures is another topic of interest.

References

1 K. Aehlig, J. G. de Miranda, and C. H. L. Ong. Safety is not a restriction at level 2 for string languages. In *FOSSACS 2005*, pages 490–504. Springer Berlin Heidelberg, 2005.

2 A. V. Aho. Indexed grammars – an extension of context free grammars. In *8th Annual Symposium on Switching and Automata Theory*, pages 21–31. IEEE, 1967.

3 C. Aiswarya and P. Gastin. Reasoning about distributed systems: WYSIWYG. In *34th International Conference on Foundation of Software Technology and Theoretical Computer Science, FSTTCS 2014, Proceedings*, volume 29 of *Leibniz International Proceedings in Informatics*, pages 11–30. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2014.

4 C. Aiswarya, P. Gastin, and K. Narayan Kumar. Verifying communicating multi-pushdown systems via split-width. In *Proceedings of ATVA'14*, volume 8837 of *Lecture Notes in Computer Science*, pages 1–17. Springer, 2014.

5 R. Alur, S. Chaudhuri, and P. Madhusudan. Languages of nested trees. In *Proceedings of CAV'06*, volume 4144 of *Lecture Notes in Computer Science*, pages 329–342. Springer, 2006.

6 R. Alur, S. Chaudhuri, and P. Madhusudan. Software model checking using languages of nested trees. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 33(5):15, 2011.

7 R. Alur and P. Madhusudan. Adding nesting structure to words. *Journal of the ACM*, 56(3), 2009.

8 B. Bollig, A. Cyriac, P. Gastin, and M. Zeitoun. Temporal logics for concurrent recursive programs: Satisfiability and model checking. In *Proceedings of MFCS'11*, volume 6907 of *Lecture Notes in Computer Science*, pages 132–144. Springer, 2011.

9 B. Bollig, A. Cyriac, P. Gastin, and M. Zeitoun. Temporal logics for concurrent recursive programs: Satisfiability and model checking. *Journal of Applied Logic*, 12(4):395–416, 2014.

10 B. Bollig, D. Kuske, and I. Meinecke. Propositional dynamic logic for message-passing systems. *Logical Methods in Computer Science*, 6(3:16), 2010.

11 C. H. Broadbent. Prefix rewriting for nested-words and collapsible pushdown automata. In *Proceedings, Part II, of ICALP'12*, volume 7392, pages 153–164. Springer, 2012.

12 T. Cachat. Higher order pushdown automata, the Caucal hierarchy of graphs and parity games. In *Proceedings of ICALP'03*, volume 2719 of *Lecture Notes in Computer Science*, pages 556–569. Springer, 2003.

13 A. Carayol and S. Wöhrle. The Caucal hierarchy of infinite graphs in terms of logic and higher-order pushdown automata. In *Proceedings of FSTTCS'03*, volume 2914 of *Lecture Notes in Computer Science*, pages 112–123. Springer, 2003.

14 D. Caucal. On infinite terms having a decidable monadic theory. In *Proceedings of MFCS'02*, LNCS, pages 165–176, Berlin, Heidelberg, 2002. Springer.

15 B. Courcelle. Special tree-width and the verification of monadic second-order graph pr operties. In *Porceedings of FSTTCS'10*, volume 8 of *LIPIcs*, pages 13–29, 2010.

16 A. Cyriac. *Verification of Communicating Recursive Programs via Split-width.* Phd thesis, Laboratoire Spécification et Vérification, ENS Cachan, France, 2014. `http://www.lsv.ens-cachan.fr/Publis/PAPERS/PDF/cyriac-phd14.pdf`.

17 A. Cyriac, P. Gastin, and K. Narayan Kumar. MSO decidability of multi-pushdown systems via split-width. In *Proceedings of CONCUR'12*, volume 7454 of *Lecture Notes in Computer Science*, pages 547–561. Springer, 2012.

18 M.J. Fischer and R.E. Ladner. Propositional dynamic logic of regular programs. *Journal of Computer and System Sciences*, 18(2):194–211, 1979.

19 S. Göller, M. Lohrey, and C. Lutz. PDL with intersection and converse: satisfiability and infinite-state model checking. *The Journal of Symbolic Logic*, 74(1):279–314, 2009.

20 S. A. Greibach. Full AFLs and nested iterated substitution. *Information and Control*, 16(1):7–35, 1970.

21 M. Hague, Andrzej S. Murawski, C.-H. Luke Ong, and O. Serre. Collapsible pushdown automata and recursion schemes. In *Proceedings of LICS'08*, pages 452–461. IEEE Computer Society, 2008.

22 J.G. Henriksen and P.S. Thiagarajan. Dynamic linear time temporal logic. *Ann. Pure Appl. Logic*, 96(1-3):187–207, 1999.

23 A. Kartzow. First-order logic on higher-order nested pushdown trees. *ACM Transactions on Computational Logic (TOCL)*, 14(2):8, 2013.

24 T. Knapik, D. Niwinski, and P. Urzyczyn. Higher-order pushdown trees are easy. In *Proceedings of FOSSACS'02*, volume 2303 of *Lecture Notes in Computer Science*, pages 205–222. Springer, 2002.

25 A. N. Maslov. Multilevel stack automata. *Problemy Peredachi Informatsii*, 12(1):55–62, 1976.

26 C.-H. Luke Ong. On model-checking trees generated by higher-order recursion schemes. In *Proceedings of LICS'06)*, pages 81–90. IEEE Computer Society, 2006.

27 C.-H. Luke Ong. Recursion schemes, collapsible pushdown automata and higher-order model checking. In *Proceedings of LATA'13*, volume 7810 of *Lecture Notes in Computer Science*, pages 13–41. Springer, 2013.

28 C.-H. Luke Ong. Higher-order model checking: An overview. In *Proceedings of LICS'15*, pages 1–15. IEEE Computer Society, 2015.

29 Wikipedia. Higher-order function — wikipedia, the free encyclopedia, 2015. [Online; accessed 12-July-2015], from `https://en.wikipedia.org/w/index.php?title=Higher-order_function&oldid=669089706`.

A Characterisation of the matching relations

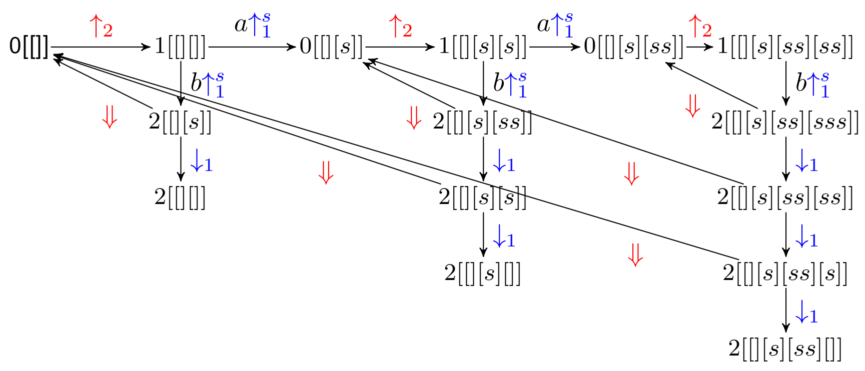
Proposition 1. Let $\mathsf{op}_0\mathsf{op}_1\cdots\mathsf{op}_n \in \mathsf{Op}^+$ be a valid push/pop/collapse sequence. Then, for all $0 \leqslant i \leqslant j \leqslant n$ we have (proof in Appendix A)
P1. $\mathsf{push}_1(j) = i$ iff $\mathsf{op}_i\cdots\mathsf{op}_j \in L_1$,
P2. $\mathsf{Push}_2(j) = i$ iff $\mathsf{op}_i\cdots\mathsf{op}_j \in L_2$,
P3. $\mathsf{push}_1(j) = -1$ iff $\mathsf{op}_k\cdots\mathsf{op}_j \notin L_1$ for all $0 \leqslant k \leqslant j$,
P4. $\mathsf{Push}_2(j) = -1$ iff $\mathsf{op}_k\cdots\mathsf{op}_j \notin L_2$ for all $0 \leqslant k \leqslant j$.

Proof. Notice that (P3) and (P4) follow directely from (P1) and (P2). We prove (P1) and (P2) simultaneously by induction on j and by a case splitting depending on op_j.

- Assume $\mathsf{op}_j = \uparrow_1$.
  (P1) We have, $\mathsf{push}_1(j) = j$. Moreover, $\mathsf{op}_i\cdots\mathsf{op}_j \in L_1$ iff $i = j$.
  (P2) We have $\mathsf{op}_i\cdots\mathsf{op}_j \in L_2$ iff $j > i$ and $\mathsf{op}_i\cdots\mathsf{op}_{j-1} \in L_2$ iff $j > i$ and (by induction) $\mathsf{Push}_2(j-1) = i$ iff $\mathsf{Push}_2(j) = i$.
- Assume $\mathsf{op}_j = \uparrow_2$.
  (P1) We have $\mathsf{op}_i\cdots\mathsf{op}_j \in L_1$ iff $j > i$ and $\mathsf{op}_i\cdots\mathsf{op}_{j-1} \in L_1$ iff $j > i$ and (by induction) $\mathsf{push}_1(j-1) = i$ iff $\mathsf{push}_1(j) = i$.
  (P2) We have, $\mathsf{Push}_2(j) = j$. Moreover, $\mathsf{op}_i\cdots\mathsf{op}_j \in L_2$ iff $i = j$.
- Assume $\mathsf{op}_j = \downarrow_1$.
  (P1) We have $\mathsf{op}_i\cdots\mathsf{op}_j \in L_1$ iff $\mathsf{op}_i\cdots\mathsf{op}_{k-1} \in L_1$ and $\mathsf{op}_k\cdots\mathsf{op}_{j-1} \in L_1$ for some $i < k < j$ iff (by induction) $\mathsf{push}_1(k-1) = i$ and $\mathsf{push}_1(j-1) = k$ for some $i < k < j$ iff $i = \mathsf{push}_1(\mathsf{push}_1(j-1)-1)$ iff $\mathsf{push}_1(j) = i$.
  Let us explain the last equivalence. The symbol which is popped by $\mathsf{op}_j = \downarrow_1$ was the top symbol at $j-1$, which was pushed at $\mathsf{push}_1(j-1) = k$. We deduce that the top symbol after $\mathsf{op}_j = \downarrow_1$ is the top symbol before $\mathsf{op}_k = \uparrow_1$. Therefore, $\mathsf{push}_1(j) = \mathsf{push}_1(k-1)$.
  (P2) We have $\mathsf{op}_i\cdots\mathsf{op}_j \in L_2$ iff $j > i$ and $\mathsf{op}_i\cdots\mathsf{op}_{j-1} \in L_2$ iff $j > i$ and (by induction) $\mathsf{Push}_2(j-1) = i$ iff $\mathsf{Push}_2(j) = i$.
- Assume $\mathsf{op}_j = \downarrow_2$.
  (P1) We have $\mathsf{op}_i\cdots\mathsf{op}_j \in L_1$ iff $\mathsf{op}_i\cdots\mathsf{op}_{k-1} \in L_1$ and $\mathsf{op}_k\cdots\mathsf{op}_{j-1} \in L_2$ for some $i < k < j$ iff (by induction) $\mathsf{push}_1(k-1) = i$ and $\mathsf{Push}_2(j-1) = k$ for some $i < k < j$ iff $i = \mathsf{push}_1(\mathsf{Push}_2(j-1)-1)$ iff $\mathsf{push}_1(j) = i$.
  Let us explain the last equivalence. The order-1 stack which is popped by $\mathsf{op}_j = \downarrow_2$ was the top order-1 stack at $j-1$, which was pushed at $\mathsf{Push}_2(j-1) = k$. We deduce that the top order-1 stack after $\mathsf{op}_j = \downarrow_2$ is the top order-1 stack before $\mathsf{op}_k = \uparrow_2$. Hence, the top symbol after $\mathsf{op}_j = \downarrow_1$ is the top symbol before $\mathsf{op}_k = \uparrow_1$. Therefore, $\mathsf{push}_1(j) = \mathsf{push}_1(k-1)$.
  (P2) The proof is obtained mutatis mutandis from the proof of (P1) above.
- Assume $\mathsf{op}_j = \Downarrow$.
  (P1) We have $\mathsf{op}_i\cdots\mathsf{op}_j \in L_1$ iff $\mathsf{op}_i\cdots\mathsf{op}_{k-1} \in L_1$ and $\mathsf{op}_k\cdots\mathsf{op}_{\ell-1} \in L_2$ and $\mathsf{op}_\ell\cdots\mathsf{op}_{j-1} \in L_1$ for some $i < k < \ell < j$ iff (by induction) $\mathsf{push}_1(k-1) = i$ and $\mathsf{Push}_2(\ell-1) = k$ and $\mathsf{push}_1(j-1) = \ell$ for some $i < k < \ell < j$ iff $i = \mathsf{push}_1(\mathsf{Push}_2(\mathsf{push}_1(j-1)-1)-1)$ iff $i = \mathsf{push}_1(j)$.
  Let us explain the last equivalence. The collapse operation depends on the top symbol before $\mathsf{op}_j = \Downarrow$. This symbol was pushed at $\mathsf{push}_1(j-1) = \ell$. The collapse link which was created by $\mathsf{op}_\ell = \uparrow_1$ points to the order-1 stack just below the top order-1 stack before (or after) op_ℓ. This top order-1 stack was pushed at $k = \mathsf{Push}_2(\ell-1) = \mathsf{Push}_2(\ell)$. Now, the order-2 stack after $\mathsf{op}_j = \Downarrow$ is exactely the order-2 stack before $\mathsf{op}_k = \uparrow_2$. Therefore, $\mathsf{push}_1(j) = \mathsf{push}_1(k-1)$.

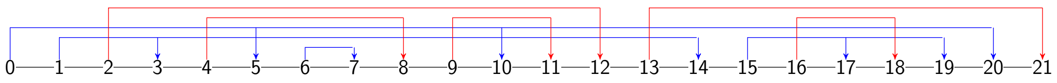
(P2) The proof is obtained mutatis mutandis from the proof of (P1) above. ◻

B Infinite half grid in the configuration graph of $\mathcal{H}_1$



C Example showing context-pop and context-suffix

Consider the 2-NW below.



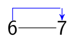
We have,

ctxt-pop$(3) = 12$ ctxt-pop$(5) =$ ctxt-pop$(7) = 8$

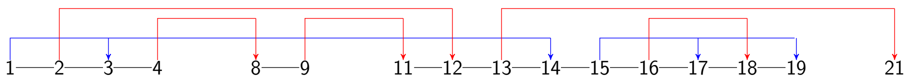
ctxt-pop$(10) = 11$ ctxt-pop$(17) = 18$

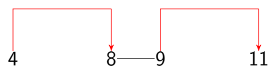
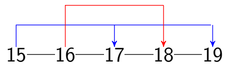
ctxt-pop$(14) =$ ctxt-pop$(19) =$ ctxt-pop$(20) = 21$

Node 0 and node 1 are top-level $\uparrow_1$.

Further, ctxt-suffix$(5) =$  and ctxt-suffix$(10) =$ ctxt-suffix$(20) = \epsilon$.
Notice that the context-suffixes of 3 and 14 are connected to position 0. This is because the corresponding $\uparrow_1$ at position 1 is not the left-most.

Now, if we remove the $\uparrow_1$ at position 0, its corresponding pops 5, 10, 20 and their context-suffixes we obtain



Then ctxt-suffix$(3) =$  and ctxt-suffix$(14) =$  are not connected to the rest of the 2-NW.

D Tree interpretation and decision procedures

In this section, we show that 2-NWs of split-width at most k have special tree-width (STW) at most $2k$. We deduce that 2-NWs of bounded split-width can be interpreted in special tree terms (STTs), which are binary trees denoting graphs of bounded STW. Special tree-width and special tree terms were introduced by Courcelle in [15].

A crucial step towards our decision procedures is then to construct a tree automaton $\mathcal{A}^{k-\mathsf{sw}}_{\mathsf{2\text{-}NW}}$ which accepts special tree terms denoting graphs that are 2-NW of split-width at most k. The main difficulty is to make sure that the edge relations $\curvearrowright^1$ and $\curvearrowright^2$ of the graph are well nested. To achieve this with a tree automaton of size $2^{\mathsf{Poly}(k)}$, we use the characterization given by the LCPDL formula ϕ_{wn} of Example 2. Similarly, we can construct a tree automaton $\mathcal{A}^{\beta}_{\mathsf{2\text{-}NW}}$ of size $2^{\mathsf{Poly}(\beta)}$ accepting STTs denoting nested words in 2-NW(β).

Special tree terms form an algebra to define graphs. A (Σ, Γ)-labelled graph is a tuple $G = \langle V, (E_\gamma)_{\gamma\in\Gamma}, \lambda\rangle$ where $\lambda\colon V \to \Sigma$ is the vertex labelling and $E_\gamma \subseteq V^2$ is the set of edges for each label $\gamma \in \Gamma$. For 2-NW, we have three types of edges, so $\Gamma = \{\to, \curvearrowright^1, \curvearrowright^2\}$. The syntax of k-STTs over (Σ, Γ) is given by

$$\tau = (i, a) \mid \mathsf{Add}^{\gamma}_{i,j}\, \tau \mid \mathsf{Forget}_i\, \tau \mid \mathsf{Rename}_{i,j}\, \tau \mid \tau \oplus \tau$$

where $a \in \Sigma$, $\gamma \in \Gamma$ and $i, j \in [k] = \{1, \ldots, k\}$ are colors.

Each k-STT represents a colored graph $[\![\tau]\!] = (G_\tau, \chi_\tau)$ where G_τ is a (Σ, Γ)-labelled graph and $\chi_\tau\colon [k] \to V_\tau$ is a partial injective function assigning a vertex of G_τ to some colors. $[\![(i, a)]\!]$ consists of a single a-labelled vertex with color i. $\mathsf{Add}^{\gamma}_{i,j}$ adds a γ-labelled edge to the vertices colored i and j (if such vertices exist). Forget_i removes color i and $\mathsf{Rename}_{i,j}$ exchanges the colors i and j. Finally, $\oplus$ constructs the disjoing union of the two graphs provided they use different colors. This operation is undefined otherwise. The special tree-width of a graph G is the least k such that $G = G_\tau$ for some $(k + 1)$-STT τ.

For instance, atomic split-2-NWs are denoted by STTs of the following form

- $(1, a)$ for an internal event labelled a,
- $\mathsf{Add}^{\curvearrowright^2}_{1,2}((1, a) \oplus (2, b))$ for an order-2 matching pair, and
- $\mathsf{Add}^{\curvearrowright^1}_{1,2} \cdots \mathsf{Add}^{\curvearrowright^1}_{1,p}((1, a_1) \oplus \cdots \oplus (p, a_p))$ for order-1 push with $p - 1$ pops.

We call these STTs *atomic*.

To each split-tree T of width k with root labelled $\overline{\mathcal{N}}$, we associate a $2k$-STT τ such that $[\![\tau]\!] = (\overline{\mathcal{N}}, \chi)$ and all endpoints of factors of $\overline{\mathcal{N}}$ have different colors. Since we have at most k factors, we may use at most $2k$ colors. A leaf of T is labelled with an atomic split-2-NW and we associate the corresponding atomic STT as defined above. At a binary node, assuming that τ_ℓ and τ_r are the STTs of the children, we first define τ'_r by renaming colors in τ_r so that colors in τ_ℓ and τ'_r are disjoint, then we let $\tau = \tau_\ell \oplus \tau'_r$. At a unary node x with child x', some factors of the spilt-2-NW $\overline{\mathcal{N}}_x$ are split resulting in the split-2-NW $\overline{\mathcal{N}}_{x'}$. Assume that factor u of $\overline{\mathcal{N}}_x$ is split in two factors u' and u'' of $\overline{\mathcal{N}}_{x'}$. The right and left endpoints of u' and u'' respectively are colored, say by i and j, in the STT τ' associated with x'. Then, we add a successor edge ($\mathsf{Add}^{\to}_{i,j}$) and we forget i if $|u'| > 1$ and j if $|u''| > 1$. We proceed similarly if a factor of $\overline{\mathcal{N}}_x$ is split in more than two factors of $\overline{\mathcal{N}}_{x'}$, and we iterate for each factor of $\overline{\mathcal{N}}_x$ which is split in $\overline{\mathcal{N}}_{x'}$.

Proposition 11. There is a tree automaton $\mathcal{A}^{\beta}_{\mathsf{2\text{-}NW}}$ of size $2^{\mathsf{Poly}(\beta)}$ accepting k-STTs ($k = 4\beta + 4$) and such that $\text{2-NW}(\beta) = \{G_\tau \mid \tau \in \mathcal{L}(\mathcal{A}^{\beta}_{\mathsf{2\text{-}NW}})\}$.

The automaton $\mathcal{A}^{\beta}_{\text{2-NW}}$ will accept precisely those k-STTs arising from split-trees as described above. The construction of $\mathcal{A}^{\beta}_{\text{2-NW}}$ is given below. The main difficulty is to check that the graph denoted by a special tree term denotes a valid 2-NW: $< = \rightarrow^{+}$ should be a total order and the relations $\curvearrowright^{1}$ and $\curvearrowright^{2}$ should be well-nested. Since we start from nested words in 2-NW(β), we obtain split-trees of width at most $2\beta+2$ by Theorem 10. Notice that $\mathcal{A}^{\beta}_{\text{2-NW}}$ needs not accept *all* k-STTs denoting graphs that are nested words in 2-NW(β).

Proof. First, we construct a tree automaton $\mathcal{A}^{\beta}_{\text{word}}$ whose states constists of

- $n \leqslant 2\beta+2$: Number of factors in the split-2-NW.
- $c_\ell, c_r : [n] \to [k]$: Functions describing colors of the left and right endpoints of the factor.

To check more easily absence of $\rightarrow$-cycles, factors are numbered according to their guessed ordering in the final 2-NW. The transitions of $\mathcal{A}^{\beta}_{\text{word}}$ ensure the following conditions:

- Atomic STTs: the automaton checks that they denote atomic split nested words in 2-NW(β). Then, the number n of factors is at most $\beta+1$ and c_ℓ, c_r are the identity maps $\mathsf{id}\colon [n] \to [k]$.
- Consider a subterm $\tau = \tau_1 \oplus \tau_2$, we have $n = n_1 + n_2$. We guess how factors of τ_1 and τ_2 will be shuffled on each process and we inherit c_ℓ and c_r accordingly.
- Consider a subterm $\tau = \mathsf{Add}^{\rightarrow}_{i,j}(\tau')$. Let (n', c'_ℓ, c'_r) be the state at τ'. We check that there are factors $x, y \in [n']$ such that $c'_r(x) = i$, $c'_\ell(y) = j$ and $y = x+1$ (this checks that the guessed ordering of the factors is correct). The states at τ is easy to compute.
  - $n = n' - 1$
  - $c_\ell(z) = c'_\ell(z)$ for $z \leqslant x$ and $c_\ell(z) = c'_\ell(z+1)$ for $z > x$
  - $c_r(z) = c'_r(z)$ for $z < x$ and $c_r(z) = c'_r(z+1)$ for $z \geqslant x$
- $\mathsf{Forget}_i\,\tau$: Check that $i \notin \mathsf{Im}(c_\ell) \cup \mathsf{Im}(c_r)$ is not in the image of the mappings c_ℓ and c_r. We always keep the colors of the endpoints of the factors.
- $\mathsf{Rename}_{i,j}$: Update c_ℓ and c_r accordingly.
- Root: Check that n=1 (a single factor).

When an STT τ is accepted by $\mathcal{A}^{\beta}_{\text{word}}$, then the relation $\rightarrow$ of the graph G_τ defines a total order on the vertices. Hence, we have an underlying word with some nesting relations $\curvearrowright^{1}$ and $\curvearrowright^{2}$. But we did not check that these relations are well-nested. To check this property, we use the LCPDL formula ϕ_{wn} of Example 2.

Consider an STT τ accepted by $\mathcal{A}^{\beta}_{word}$, let $[\![\tau]\!] = (G_\tau, \chi_\tau)$ where $G_\tau = (V, \rightarrow, \curvearrowright^{1}, \curvearrowright^{2}, \lambda)$. we know that $(V, \rightarrow, \lambda)$ defines a word in Σ^{+}. The graph G_τ can be interpreted in τ: we can build walking automata of size $\mathsf{Poly}(k)$ for $\rightarrow$, $\curvearrowright^{1}$, $\curvearrowright^{2}$ and their converse. Hence we can build an alternating two-way tree automaton of size $\mathsf{Poly}(k)$ checking ϕ_{wn}. We obtain an equivalent normal tree automaton $\mathcal{A}^{\beta}_{\text{nw}}$ of size $2^{\mathsf{Poly}(k)}$

The final tree automaton is $\mathcal{A}^{\beta}_{\text{2-NW}} = \mathcal{A}^{\beta}_{\text{word}} \cap \mathcal{A}^{\beta}_{\text{nw}}$. ◻

Proposition 12. For each 2-CPDS $\mathcal{H}$ we can construct a tree automaton $\mathcal{A}^{\beta}_{\mathcal{H}}$ of size $2^{\mathsf{Poly}(\beta,|\mathcal{H}|)}$ such that $\mathcal{L}(\mathcal{A}^{\beta}_{\text{2-NW}} \cap \mathcal{A}^{\beta}_{\mathcal{H}}) = \{\tau \in \mathcal{L}(\mathcal{A}^{\beta}_{\text{2-NW}}) \mid G_\tau \in \mathcal{L}(\mathcal{H})\}$.

Proof. The tree automaton $\mathcal{A}^{\beta}_{\mathcal{H}}$ essentially guesses the transitions of the 2-CPDS and checks that they form an accepting run. To this end, we first construct a tree automaton reading STTs whose leaves are additionally labelled with transitions of $\mathcal{H}$. Then, we project away these additional labels to obtain the automaton $\mathcal{A}^{\beta}_{\mathcal{H}}$.

Consider an atomic STT describing a 2-NW of the form $\begin{tikzcd} a \arrow[r, bend left] \arrow[rr, bend left] \arrow[rrr, bend left] & b & c & d \end{tikzcd}$. The tree automaton checks that the transitions labelling the leaves are of the form $(q_1^a, a, \uparrow_1^s, q_2^a)$, $(q_1^b, b, \downarrow_1, q_2^b)$, $(q_1^c, c, \downarrow_1, q_2^c)$ and $(q_1^d, d, \downarrow_1, q_2^d)$.

Another interesting case is for internal events carrying top tests. We construct an alternating two-way walking tree automaton (A2A) of size $\mathsf{Poly}(\beta, |\mathcal{H}|)$, which visits all leaves of the input STT. For each leaf x which is labelled with a top test transition $(q_1, a, \mathsf{top}(s), q_2)$, the A2A walks to the leaf y following the LCPDL formula π_{push_1} of Example 2. The top symbol of the order-2 stack at node x was pushed at node y. Hence, the A2A checks that the transition labelling y is of the form $(q_1', b, \uparrow_1^s, q_2')$. This A2A is transformed into a classical tree automaton of size $2^{\mathsf{Poly}(\beta, |\mathcal{H}|)}$.

It remains to check that, when following the linear order $\to$, the transitions labelling the leaves form an accepting run. To this end, the bottom-up tree automaton remembers; for every factor of the split-2-NW associated with a node of the STT, the starting control state and the ending control state. It is then easy to verify when adding a $\to$ edge between two factors u and v that the target state of factor u is the source state of factor v. Finally, at the root of the STT, the automaton accepts if there is only one factor and its source/target state is initial/final.

The size of the tree automaton $\mathcal{A}_{\mathcal{H}}^{\beta}$ is $2^{\mathsf{Poly}(\beta, |\mathcal{H}|)}$. ◻

We deduce from Proposition 12 that non-emptiness checking of 2-CPDS with respect to 2-NW(β) is in EXPTIME.

Proposition 13. For each LCPDL formula ϕ we can construct a tree automaton $\mathcal{A}_{\phi}^{\beta}$ of size $2^{\mathsf{Poly}(\beta, |\phi|)}$ such that $\mathcal{L}(\mathcal{A}_{\text{2-NW}}^{\beta} \cap \mathcal{A}_{\phi}^{\beta}) = \{\tau \in \mathcal{L}(\mathcal{A}_{\text{2-NW}}^{\beta}) \mid G_\tau \in \mathcal{L}(\phi)\}$.

Proof. The idea is to translate the LCPDL formula ϕ to an alternating two-way tree automaton (A2A) of size $\mathsf{Poly}(\beta, |\phi|)$. Due to the specific form of the STTs accepted by $\mathcal{A}_{\text{2-NW}}^{\beta}$, it is easy to encode the nesting relations $\curvearrowright^1$ and $\curvearrowright^2$ with a walking automaton. We can also easily build a walking automaton for the successor relation $\to$ by tracking the colors until we reach a node labelled $\mathsf{Add}_{i,j}^{\to}$. One main difficulty is to cope with loops of LCPDL. Here we use the result of [19] for PDL with converse and intersection. In general, this can cause an exponential blow-up in the size of the A2A. But loop is a special case with bounded intersection-width and hence still allows a polynomial sized A2A. Finally, the A2A for ϕ is translated to the normal tree automaton $\mathcal{A}_{\phi}^{\beta}$, causing an exponential blow-up. ◻

We deduce that the bounded-pop satisfiability problem of LCPDL can be solved in exponential time by checking emptiness of $\mathcal{A}_{\text{2-NW}}^{\beta} \cap \mathcal{A}_{\phi}^{\beta}$. Also, the bounded-pop model checking problem of 2-CPDS against LCPDL can be solved in exponential time by checking emptiness of $\mathcal{A}_{\text{2-NW}}^{\beta} \cap \mathcal{A}_{\mathcal{H}}^{\beta} \cap \mathcal{A}_{\neg\phi}^{\beta}$.

Similarly, for each MSO formula ϕ, we can construct a tree automaton $\mathcal{A}_{\phi}^{\beta}$ such that $\mathcal{L}(\mathcal{A}_{\text{2-NW}}^{\beta} \cap \mathcal{A}_{\phi}^{\beta}) = \{\tau \in \mathcal{L}(\mathcal{A}_{\text{2-NW}}^{\beta}) \mid G_\tau \in \mathcal{L}(\phi)\}$. We deduce the decidability of the bounded-pop satisfiability problem of MSO and of the bounded-pop model checking problem of 2-CPDS against MSO. This concludes the proof of Theorem 8.