Saturation of Concurrent Collapsible Pushdown Systems

M. Hague

Royal Holloway University of London, and LIGM, Marne-la-Vallée
matthew.hague@rhul.ac.uk

Abstract

Multi-stack pushdown systems are a well-studied model of concurrent computation using threads with first-order procedure calls. While, in general, reachability is undecidable, there are numerous restrictions on stack behaviour that lead to decidability. To model higher-order procedures calls, a generalisation of pushdown stacks called collapsible pushdown stacks are required. Reachability problems for multi-stack collapsible pushdown systems have been little studied. Here, we study ordered, phase-bounded and scope-bounded multi-stack collapsible pushdown systems using saturation techniques, showing decidability of control state reachability and giving a regular representation of all configurations that can reach a given control state.

1998 ACM Subject Classification F.1.1 Models of Computation

Keywords and phrases Concurrency, Automata, Higher-Order, Verification, Model-Checking

Digital Object Identifier 10.4230/LIPIcs.xxx.yyy.p

1 Introduction

Pushdown systems augment a finite-state machine with a stack and accurately model first-order recursion. Such systems then are ideal for the analysis of sequential first-order programs and several successful tools, such as Moped [27] and SLAM [3], exist for their analysis. However, the domination of multi- and many-core machines means that programmers must be prepared to work in concurrent environments, with several interacting execution threads.

Unfortunately, the analysis of concurrent pushdown systems is well-known to be undecidable. However, most concurrent programs don't interact pathologically and many restrictions on interaction have been discovered that give decidability (e.g. [5, 6, 28, 15, 16]).

One particularly successful approach is *context-bounding*. This underapproximates a concurrent system by bounding the number of context switches that may occur [26]. It is based on the observation that most real-world bugs require only a small number of thread interactions [25]. Additionally, a number of more relaxed restrictions on stack behaviour have been introduced. In particular phase-bounded [31], scope-bounded [32], and ordered [7] (corrected in [2]) systems. There are also generic frameworks — that bound the tree- [22] or split-width [10] of the interactions between communication and storage — that give decidability for all communication architectures that can be defined within them.

Languages such as C++, Haskell, Javascript, Python, or Scala increasingly embrace higher-order procedure calls, which present a challenge to verification. A popular approach to modelling higher-order languages for verification is that of (higher-order recursion) schemes [11, 23, 17]. Collapsible pushdown systems (CPDS) are an extension of pushdown systems [14] with a "stack-of-stacks" structure. The "collapse" operation allows a CPDS to retrieve information about the context in which a stack character was created. These features give CPDS equivalent modelling power to schemes [14].

These two formalisms have good model-checking properties. E.g, it is decidable whether a μ-calculus formula holds on the execution graph of a scheme [23] (or CPDS [14]). Al-

Conference title on which this volume is based on.
Editors: Billy Editor, Bill Editors; pp. 1–12
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

though, the complexity of such analyses is high, it has been shown by Kobayashi [16] (and Broadbent *et al.* for CPDS [8]) that they can be performed in practice on real code examples.

However concurrency for these models has been little studied. Work by Seth considers phase-bounding for CPDS without collapse [29] by reduction to a finite state parity game. Recent work by Kobayashi and Igarashi studies context-bounded recursion schemes [19].

Here, we study global reachability problems for ordered, phase-bounded, and scope-bounded CPDS. We use *saturation* methods, which have been successfully implemented by e.g. Moped [27] for pushdown systems and C-SHORe [8] for CPDS. Saturation was first applied to model-checking by Bouajjani *et al.* [4] and Finkel *et al.* [12]. We presented a saturation technique for CPDS in ICALP 2012 [9]. Here, we present the following advances.

1. Global reachability for ordered CPDSs (§5). This is based on Atig's algorithm [1] for ordered PDSs and requires a non-trivial generalisation of his notion of *extended* PDSs (§3). For this we introduce the notion of *transition automata* that encapsulate the behaviour of the saturation algorithm. In the full article we show how to use the same machinery to solve the global reachability problem for phase-bounded CPDSs.
2. Global reachability for scope-bounded CPDSs (§6). This is a backwards analysis based upon La Torre and Napoli's forwards analysis for scope-bounded PDSs, requiring new insights to complete the proofs.

Because the naive encoding of a single second-order stack has an undecidable MSO theory (we show this folklore result in the full paper) it remains a challenging open problem to generalise the generic frameworks above ([22, 10]) to CPDSs, since these frameworks rely on MSO decidability over graph representations of the storage and communication structure.

A full version of this paper with all definitions and proofs is available [13].

2 Preliminaries

Before defining CPDSs, we define $2\uparrow_0(x) = x$ and $2\uparrow_{i+1}(x) = 2^{2\uparrow_i(x)}$.

2.1 Collapsible Pushdown Systems (CPDS)

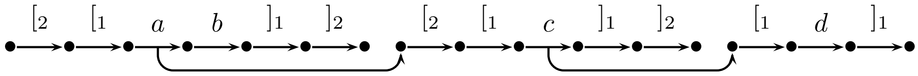
For a readable introduction to CPDS we defer to a survey by Ong [24]. Here, we can only briefly describe higher-order collapsible stacks and their operations. We use a notion of collapsible stacks called *annotated stacks* (which we refer to as collapsible stacks). These were introduced in ICALP 2012, and are essentially equivalent to the classical model [9].

Higher-Order Collapsible Stacks An order-1 stack is a stack of symbols from a stack alphabet Σ, an order-n stack is a stack of order-$(n-1)$ stacks. A collapsible stack of order n is an order-n stack in which the stack symbols are annotated with collapsible stacks which may be of any order $\leq n$. Note, often in examples we will omit annotations for clarity. We fix the maximal order to n, and use k to range between n and 1. We simultaneously define for all $1 \leq k \leq n$, the set Stacks_k^n of order-k stacks whose symbols are annotated by stacks of order at most n. Note, we use subscripts to indicate the order of a stack. Furthermore, the definition below uses a least fixed-point. This ensures that all stacks are finite. An order-k stack is a collapsible stack in Stacks_k^n.

▶ **Definition 2.1** (Collapsible Stacks)**.** The family of sets $(\mathrm{Stacks}_k^n)_{1\leq k\leq n}$ is the smallest family (for point-wise inclusion) such that:

1. for all $2 \leq k \leq n$, Stacks_k^n is the set of all (possibly empty) sequences $[w_1 \dots w_\ell]_k$ with $w_1, \dots, w_\ell \in \mathrm{Stacks}_{k-1}^n$.
2. Stacks_1^n is all sequences $[{a_1}^{w_1} \dots {a_\ell}^{w_\ell}]_1$ with $\ell \geq 0$ and for all $1 \leq i \leq \ell$, a_i is a stack symbol in Σ and w_i is a collapsible stack in $\bigcup\limits_{1\leq k\leq n} \mathrm{Stacks}_k^n$.

An order-n stack can be represented naturally as an edge-labelled tree over the alphabet $\{[_{n-1}, \ldots, [_1,]_1, \ldots,]_{n-1}\} \uplus \Sigma$, with Σ-labelled edges having a second target to the tree representing the annotation. We do not use $[_n$ or $]_n$ since they would appear uniquely at the beginning and end of the stack. An example order-3 stack is given below, with only a few annotations shown (on a and c). The annotations are order-3 and order-2 respectively.



Given an order-n stack $w = [w_1 \ldots w_\ell]_n$, we define $top_{n+1}(w) = w$ and

$$\begin{array}{rcll} top_n([w_1 \ldots w_\ell]_n) & = & w_1 & \text{when } \ell > 0 \\ top_n([]_n) & = & []_{n-1} & \text{otherwise} \\ top_k([w_1 \ldots w_\ell]_n) & = & top_k(w_1) & \text{when } k < n \text{ and } \ell > 0 \end{array}$$

noting that $top_k(w)$ is undefined if $top_{k'}(w) = []_{k'-1}$ for any $k' > k$.

We write $u :_k v$ — where u is order-$(k-1)$ — to denote the stack obtained by placing u on top of the top_k stack of v. That is, if $v = [v_1 \ldots v_\ell]_k$ then $u :_k v = [uv_1 \ldots v_\ell]_k$, and if $v = [v_1 \ldots v_\ell]_{k'}$ with $k' > k$, $u :_k v = [(u :_k v_1)\, v_2 \ldots v_\ell]_{k'}$. This composition associates to the right. E.g., the stack $[[[a^w b]_1]_2]_3$ above can be written $u :_3 v$ where u is the order-2 stack $[[a^w b]_1]_2$ and v is the empty order-3 stack $[]_3$. Then $u :_3 u :_3 v$ is $[[[a^w b]_1]_2[[a^w b]_1]_2]_3$.

Operations on Order-n Collapsible Stacks The following operations can be performed on an order-n stack where *noop* is the null operation $noop(w) = w$.

$$\mathcal{O}_n \quad = \quad \{noop, pop_1\} \cup \left\{rew_a, push^k_a, copy_k, pop_k \mid a \in \Sigma \wedge 2 \leq k \leq n\right\}$$

We define each $o \in \mathcal{O}_n$ for an order-n stack w. Annotations are created by $push^k_a$, which pushes a character onto w and annotates it with $top_{k+1}(pop_k(w))$. This, in essence, attaches a closure to a new character.

1. We set $pop_k(u :_k v) = v$.
2. We set $copy_k(u :_k v) = u :_k u :_k v$.
3. We set $collapse_k\left(a^{u'} :_1 u :_{(k+1)} v\right) = u' :_{(k+1)} v$ when u is order-k and $1 \leq k < n$; and $collapse_n(a^u :_1 v) = u$ when u is order-n.
4. We set $push^k_b(w) = b^u :_1 w$ where $u = top_{k+1}(pop_k(w))$.
5. We set $rew_b(a^u :_1 v) = b^u :_1 v$.

For example, beginning with $[[a]_1[b]_1]_2$ and applying $push^2_c$ we obtain $[[c^{[[b]_1]_2}a]_1[b]_1]_2$. In this setting, the order-2 context information for the new character c is $[[b]_1]_2$. We can then apply $copy_2; collapse_2$ to get $[[c^{[[b]_1]_2}a]_1[c^{[[b]_1]_2}a]_1[b]_1]_2$ then $[[b]_1]_2$. That is, $collapse_k$ replaces the current top_{k+1} stack with the annotation attached to c.

Collapsible Pushdown Systems We are now ready to define collapsible PDS.

▶ **Definition 2.2** (Collapsible Pushdown Systems)**.** An order-n *collapsible pushdown system (n-CPDS)* is a tuple $\mathcal{C} = (\mathcal{P}, \Sigma, \mathcal{R})$ where $\mathcal{P}$ is a finite set of control states, Σ is a finite stack alphabet, and $\mathcal{R} \subseteq (\mathcal{P} \times \Sigma \times \mathcal{O}_n \times \mathcal{P})$ is a set of rules.

We write *configurations* of a CPDS as a pair $\langle p, w\rangle \in \mathcal{P} \times \mathrm{Stacks}^n_n$. We have a transition $\langle p, w\rangle \longrightarrow \langle p', w'\rangle$ via a rule (p, a, o, p') when $top_1(w) = a$ and $w' = o(w)$.

Consuming and Generating Rules We distinguish two kinds of rule or operation: a rule (p, a, o, p') or operation o is *consuming* if $o = pop_k$ or $o = collapse_k$ for some k. Otherwise, it is *generating.* We write $\mathcal{R}^{\mathcal{P},\Sigma}_{\mathcal{G}_n}$ for the set of generating rules of the form (p, a, o, p') such that $p, p' \in \mathcal{P}$ and $a \in \Sigma$, and $o \in \mathcal{O}_n$. We simply write $\mathcal{R}_{\mathcal{G}_n}$ when no confusion may arise.

2.2 Saturation for CPDS

Our algorithms for concurrent CPDSs build upon the saturation technique for CPDSs [9]. In essence, we represent sets of configurations C using a $\mathcal{P}$-stack automaton A reading stacks. We define such automata and their languages $\mathcal{L}(A)$ below. Saturation adds new transitions to A — depending on rules of the CPDS and existing transitions in A — to obtain A' representing configurations with a path to a configuration in C. I.e., given a CPDS $\mathcal{C}$ with control states $\mathcal{P}$ and a $\mathcal{P}$-stack automaton A_0, we compute $Pre^*_{\mathcal{C}}(A_0)$ which is the smallest set s.t. $Pre^*_{\mathcal{C}}(A_0) \supseteq \mathcal{L}(A_0)$ and $Pre^*_{\mathcal{C}}(A_0) \supseteq \{\langle p, w\rangle \mid \exists \langle p, w\rangle \longrightarrow \langle p', w'\rangle \text{ s.t. } \langle p', w'\rangle \in Pre^*_{\mathcal{C}}(A_0)\}$.

Stack Automata Sets of stacks are represented using order-n stack automata. These are alternating automata with a nested structure that mimics the nesting in a higher-order collapsible stack. We recall the definition below.

▶ **Definition 2.3** (Order-n Stack Automata)**.** An *order-n stack automaton* is a tuple $A = (\mathbb{Q}_n, \ldots, \mathbb{Q}_1, \Sigma, \Delta_n, \ldots, \Delta_1, \mathcal{F}_n, \ldots, \mathcal{F}_1)$ where Σ is a finite stack alphabet, $\mathbb{Q}_n, \ldots, \mathbb{Q}_1$ are disjoint, and

1. for all $2 \leq k \leq n$, we have $\mathbb{Q}_k$ is a finite set of states, $\mathcal{F}_k \subseteq \mathbb{Q}_k$ is a set of accepting states, and $\Delta_k \subseteq \mathbb{Q}_k \times \mathbb{Q}_{k-1} \times 2^{\mathbb{Q}_k}$ is a transition relation such that for all q and Q there is *at most one* q' with $(q, q', Q) \in \Delta_k$, and
2. $\mathbb{Q}_1$ is a finite set of states, $\mathcal{F}_1 \subseteq \mathbb{Q}_1$ is a set of accepting states, and the transition relation is $\Delta_1 \subseteq \bigcup\limits_{2 \leq k \leq n} \left(\mathbb{Q}_1 \times \Sigma \times 2^{\mathbb{Q}_k} \times 2^{\mathbb{Q}_1}\right)$.

States in $\mathbb{Q}_k$ recognise order-k stacks. Stacks are read from "top to bottom". A stack $u :_k v$ is accepted from q if there is a transition $(q, q', Q) \in \Delta_k$, written $q \xrightarrow{q'} Q$, such that u is accepted from $q' \in \mathbb{Q}_{(k-1)}$ and v is accepted from each state in Q. At order-1, a stack $a^u :_1 v$ is accepted from q if there is a transition (q, a, Q_{col}, Q) where u is accepted from all states in Q_{col} and v is accepted from all states in Q. An empty order-k stack is accepted by any state in $\mathcal{F}_k$. We write $w \in \mathcal{L}_q(A)$ to denote the set of all stacks w accepted from q. Note that a transition to the empty set is distinct from having no transition.

We show a part run using $q_3 \xrightarrow{q_2} Q_3 \in \Delta_3$, $q_2 \xrightarrow{q_1} Q_2 \in \Delta_2$, $q_1 \xrightarrow[Q_{col}]{a} Q_1 \in \Delta_1$.

$$q_3 \xrightarrow{[_2} q_2 \xrightarrow{[_1} q_1 \xrightarrow{a} Q_1 \xrightarrow{b} \cdots \xrightarrow{]_1} Q_2 \xrightarrow{]_2} Q_3 \qquad Q_{col} \xrightarrow{[_2} \cdots$$

Long-form Transitions We will often use a *long-form* notation (defined below) that captures nested sequences of transitions. E.g. we can write $q_3 \xrightarrow[Q_{col}]{a} (Q_1, Q_2, Q_3)$ to represent the use of $q_3 \xrightarrow{q_2} Q_3$, $q_2 \xrightarrow{q_1} Q_2$, and $q_1 \xrightarrow[Q_{col}]{a} Q_1$ for the first three transitions of the run above. Note that this latter long-form transition starts at the very beginning of the stack and reads its top_1 character. Formally, for a sequence of transitions $q \xrightarrow{q_{k-1}} Q_k, q_{k-1} \xrightarrow{q_{k-2}} Q_{k-1}, \ldots, q_1 \xrightarrow[Q_{col}]{a} Q_1$ in Δ_k to Δ_1 respectively, we write $q \xrightarrow[Q_{col}]{a} (Q_1, \ldots, Q_k)$.

$\mathcal{P}$-Stack Automata We define $\mathcal{P}$-automata [4] for CPDSs. Given control states $\mathcal{P}$, an *order-n $\mathcal{P}$-stack automaton* is an order-n stack automaton such that for each $p \in \mathcal{P}$ there exists a state $q_p \in \mathbb{Q}_n$. We set $\mathcal{L}(A) = \left\{\langle p, w\rangle \;\middle|\; w \in \mathcal{L}_{q_p}(A)\right\}$.

The Saturation Algorithm We recall the saturation algorithm. For a detailed explanation of the saturation function complete with examples, we refer the reader to our ICALP paper [9]. Here we present an abstracted view of the algorithm, relegating details that are not directly relevant to the remainder of the main article to the full version.

The saturation algorithm iterates a *saturation function* Π that adds new transitions to a given automaton. Beginning with A_0 representing a target set of configurations, we iterate $A_{i+1} = \Pi(A_i)$ until $A_{i+1} = A_i$. Once this occurs, we have that $\mathcal{L}(A_i) = Pre^*_{\mathcal{C}}(A_0)$.

We define Π in terms of a family of auxiliary saturation functions Π_r (defined in the full article) which return a set of long-form transitions to be added by saturation. When r is consuming, $\Pi_r(A)$ returns the set of long-form transitions to be added to A due to the rule r. When r is generating Π_r also takes as an argument a long-form transition t of A. Thus $\Pi_r(t, A)$ returns the set of long-form transitions that should be added to A as a result of the rule r combined with the transition t (and possibly other transitions of A).

For example, if $r = (p, a, rew_b, p')$ and $t = q_{p'} \xrightarrow[Q_{col}]{b} (Q_1, \ldots, Q_n)$ is a transition of A, then $\Pi_r(t, A)$ contains only the long-form transition $t' = q_p \xrightarrow[Q_{col}]{a} (Q_1, \ldots, Q_n)$. The idea is if $\langle p', b^u :_1 w\rangle$ is accepted by A via a run whose first (sequence of) transition(s) is t, then by adding t' we will be able to accept $\langle p, a^u :_1 w\rangle$ via a run beginning with t' instead of t. We have $\langle p, a^u :_1 w\rangle \in Pre^*_{\mathcal{C}}(A)$ since it can reach $\langle p', b^u :_1 w\rangle$ via the rule r.

▶ **Definition 2.4** (The Saturation Function Π)**.** For a CPDS with rules $\mathcal{R}$, and given an order-n stack automaton A_i we define $A_{i+1} = \Pi(A_i)$. The state-sets of A_{i+1} are defined implicitly by the transitions which are those in A_i plus, for each $r = (p, a, o, p') \in \mathcal{R}$, when

1. o is consuming and $t \in \Pi_r(A_i)$, then add t to A_{i+1},
2. o is generating, t is in A_i, and $t' \in \Pi_r(t, A)$, then add t' to A_{i+1}.

In ICALP 2012 we showed that saturation adds up to $\mathcal{O}(2 \uparrow_n (f(|\mathcal{P}|)))$ transitions, for some polynomial f, and that this can be reduced to $\mathcal{O}(2 \uparrow_{n-1} (f(|\mathcal{P}|)))$ (which is optimal) by restricting all Q_n to have size 1 when A_0 is "non-alternating at order-n". Since this property holds of all A_0 used here, we use the optimal algorithm for complexity arguments.

3 Extended Collapsible Pushdown Systems

To analyse concurrent systems, we extend CPDS following Atig [1]. Atig's extended PDSs allow words from arbitrary languages to be pushed on the stack. Our notion of extended CPDSs allows sequences of *generating operations* from a language $\mathcal{L}_g$ to be applied, rather than a single operation per rule. We can specify $\mathcal{L}_g$ by any system (e.g. a Turing machine).

▶ **Definition 3.1** (Extended CPDSs)**.** An order-n *extended CPDS* (*n-ECPDS*) is a tuple $\mathcal{C} = (\mathcal{P}, \Sigma, \mathcal{R})$ where $\mathcal{P}$ is a finite set of control states, Σ is a finite stack alphabet, and $\mathcal{R} \subseteq (\mathcal{P} \times \Sigma \times \mathcal{O}_n \times \mathcal{P}) \cup \left(\mathcal{P} \times \Sigma \times 2^{\left(\mathcal{R}^{\mathcal{P},\Sigma}_{\mathcal{G}_n}\right)^*} \times \mathcal{P}\right)$ is a set of rules.

As before, we have a transition $\langle p, w\rangle \longrightarrow \langle p', w'\rangle$ of an n-ECPDS via a rule (p, a, o, p') with $top_1(w) = a$ and $w' = o(w)$. Additionally, we have a transition $\langle p, w\rangle \longrightarrow \langle p', w'\rangle$ when we have a rule $(p, a, \mathcal{L}_g, p')$, a sequence $(p, a, o_1, p_1)(p_1, a_2, o_2, p_2)\ldots(p_{\ell-1}, a_\ell, o_\ell, p') \in \mathcal{L}_g$ and $w' = o_\ell(\cdots o_1(w))$. That is, a single extended rule may apply a sequence of stack updates in one step. A run of an ECPDS is a sequence $\langle p_0, w_0\rangle \longrightarrow \langle p_1, w_1\rangle \longrightarrow \cdots$.

3.1 Reachability Analysis

We adapt saturation for ECPDSs. In Atig's algorithm, an essential property is the decidability of $\mathcal{L}_g \cap \mathcal{L}(A)$ for some order-1 $\mathcal{P}$-stack automaton A and a language $\mathcal{L}_g$ appearing in a rule of the extended PDS. We need analogous machinery in our setting. For this, we first define a class of finite automata called *transition* automata, written $\mathcal{T}$. The states of these automata will be long-form transitions of a stack automaton $t = q \xrightarrow[Q_{col}]{a} (Q_1, \ldots, Q_n)$. Transitions $t \xrightarrow{r} t'$ are labelled by rules. We write $t \xrightarrow{\vec{r}}_* t'$ to denote a run over $\vec{r} \in (\mathcal{R}_{\mathcal{G}_n})^*$.

During the saturation algorithm we will build from A_i a transition automaton $\mathcal{T}$. Then, for each rule $(p, a, \mathcal{L}_g, p')$ we add to A_{i+1} a new long-form transition t if there is a word $\overrightarrow{r} \in \mathcal{L}_g$ such that $t \xrightarrow{\overrightarrow{r}}_* t'$ is a run of $\mathcal{T}$ and t' is already a transition of A_i.

For example, consider $(p, a, \mathcal{L}_g, p')$ where $\mathcal{L}_g = \{(p, a, rew_b, p')\}$. A transition

$$\left(q_p \xrightarrow[Q_{col}]{a} (Q_1, \ldots, Q_n)\right) \xrightarrow{(p,a,rew_b,p')} \left(q_{p'} \xrightarrow[Q_{col}]{b} (Q_1, \ldots, Q_n)\right)$$

will correspond to the fact that the presence of $q_{p'} \xrightarrow[Q_{col}]{b} (Q_1, \ldots, Q_n)$ in A_i causes $q_p \xrightarrow[Q_{col}]{a} (Q_1, \ldots, Q_n)$ to be added by Π. A run $t_1 \xrightarrow{r_1} t_2 \xrightarrow{r_2} t_3$ comes into play when e.g. $\mathcal{L}_g = \{r_1 r_2\}$. If the rule were split into two ordinary rules with intermediate control states, Π would first add t_2 derived from t_3, and then from t_2 derive t_1. In the case of extended CPDSs, the intermediate transition t_2 is not added to A_{i+1}, but its effect is still present in the addition of t_1. Below, we repeat the above intuition more formally. Fix a n-ECPDS $\mathcal{C} = (\mathcal{P}, \Sigma, \mathcal{R})$.

Transition Automata We build a transition automaton from a given $\mathcal{P}$-stack automaton A. Let A have order-n to order-1 state-sets $Q_n, \ldots, Q_1$ and alphabet Σ, let T_A be the set of all $q \xrightarrow[Q_{col}]{a} (Q_1, \ldots, Q_n)$ with $q \in Q_n$, for all k, $Q_k \subseteq \mathbb{Q}_k$, and for some k, $Q_{col} \subseteq \mathbb{Q}_k$.

▶ **Definition 3.2** (Transition Automata). Given an order-n $\mathcal{P}$-stack automaton A with alphabet Σ, and $t, t' \in T_A$, we define the transition automaton $\mathcal{T}^A_{t,t'} = \left(T_A, \mathcal{R}^{\mathcal{P},\Sigma}_{\mathcal{G}_n}, \delta, t, t'\right)$ such that $\delta \subseteq T_A \times \mathcal{R}^{\mathcal{P},\Sigma}_{\mathcal{G}_n} \times T_A$ is the smallest set such that $t_1 \xrightarrow{r} t_2 \in \delta$ if $t_1 \in \Pi_r(t_2, A)$.

We define $\mathcal{L}(\mathcal{T}^A_{t,t'}) = \left\{ \overrightarrow{r} \;\middle|\; t \xrightarrow{\overrightarrow{r}}_* t' \right\}$.

Extended Saturation Function We now extend the saturation function following the intuition explained above. For $t = q_p \xrightarrow[Q_{col}]{a} (Q_1, \ldots, Q_n)$, let $top_1(t) = a$ and $control(t) = p$.

▶ **Definition 3.3** (Extended Saturation Function Π). The extended Π is Π from Definition 2.4 plus for each extended rule $(p, a, \mathcal{L}_g, p') \in \mathcal{R}$ and t, t', we add t to A_{i+1} whenever **1.** $control(t) = p$ and $top_1(t) = a$, **2.** t' is a transition of A_i with $control(t') = p'$, and **3.** $\mathcal{L}_g \cap \mathcal{L}\left(\mathcal{T}^{A_i}_{t,t'}\right) \neq \emptyset$.

▶ **Theorem 3.4** (Global Reachability of ECPDS). *Given an ECPDS $\mathcal{C}$ and a $\mathcal{P}$-stack automaton A_0, the fixed point A of the extended saturation procedure accepts $Pre^*_{\mathcal{C}}(A_0)$.*

In order for the saturation algorithm to be effective, we need to be able to decide $\mathcal{L}_g \cap \mathcal{L}\left(\mathcal{T}^{A_i}_{t,t'}\right) \neq \emptyset$. We argue in the full paper that number of transitions added by extended saturation has the same upper bound as the unextended case.

4 Multi-Stack CPDSs

We define a general model of concurrent collapsible pushdown systems, which we later restrict. In the sequel, assume a bottom-of-stack symbol $\perp$ and define the "empty" stacks $\perp_0 = \perp$ and $\perp_{k+1} = [\perp_k]_{k+1}$. As standard, we assume that $\perp$ is neither pushed onto, nor popped from, the stack (though may be copied by $copy_k$).

▶ **Definition 4.1** (Multi-Stack Collapsible Pushdown Systems). An order-n *multi-stack collapsible pushdown system (n-MCPDS)* is a tuple $\mathcal{C} = (\mathcal{P}, \Sigma, \mathcal{R}_1, \ldots, \mathcal{R}_m)$ where $\mathcal{P}$ is a finite set of control states, Σ is a finite stack alphabet, and for each $1 \leq i \leq m$ we have a set of rules $\mathcal{R}_i \subseteq \mathcal{P} \times \Sigma \times \mathcal{O}_n \times \mathcal{P}$.

A configuration of $\mathcal{C}$ is a tuple $\langle p, w_1, \ldots, w_m \rangle$. There is a transition $\langle p, w_1, \ldots, w_m \rangle \longrightarrow \langle p', w_1, \ldots, w_{i-1}, w'_i, w_{i+1}, \ldots, w_m \rangle$ via $(p, a, o, p') \in \mathcal{R}_i$ when $a = top_1(w_i)$ and $w'_i = o(w_i)$.

We also need MCPD*Automata*, which are MCPDSs defining languages over an input alphabet Γ. For this, we add labelling input characters to the rules. Thus, a rule (p, a, γ, o, p') reads a character $\gamma \in \Gamma$. This is defined formally in the full paper.

We are interested in two problems for a given n-MCPDS $\mathcal{C}$.

▶ **Definition 4.2** (Control State Reachability Problem). Given control states $p_{\text{in}}, p_{\text{out}}$ of $\mathcal{C}$, decide if there is for some $w_1, \ldots, w_m$ a run $\langle p_{\text{in}}, \perp_n, \ldots, \perp_n \rangle \longrightarrow \cdots \longrightarrow \langle p_{\text{out}}, w_1, \ldots, w_m \rangle$.

▶ **Definition 4.3** (Global Control State Reachability Problem). Given a control state p_{out} of $\mathcal{C}$, construct a representation of the set of configurations $\langle p, w_1, \ldots, w_m \rangle$ such that there exists for some $w'_1, \ldots, w'_m$ a run $\langle p, w_1, \ldots, w_m \rangle \longrightarrow \cdots \longrightarrow \langle p_{\text{out}}, w'_1, \ldots, w'_m \rangle$.

We represent sets of configurations as follows. In the full paper we show it forms an effective boolean algebra, membership is linear time, and emptiness is in PSPACE.

▶ **Definition 4.4** (Regular Set of Configurations). A regular set R of configurations of a multi-stack CPDS $\mathcal{C}$ is definable via a finite set χ of tuples $(p, A_1, \ldots, A_m)$ where p is a control state of $\mathcal{C}$ and A_i is a stack automaton with designated initial state q_i for each i. We have $\langle p, w_1, \ldots, w_m \rangle \in R$ iff there is some $(p, A_1, \ldots, A_m) \in \chi$ such that $w_i \in \mathcal{L}_{q_i}(A_i)$ for each i.

Finally, we often partition runs of an MCPDS $\sigma = \sigma_1 \ldots \sigma_\ell$ where each σ_i is a sequence of configurations of the MCPDS. A transition from c to c' occurs in segment σ_i if c' is a configuration in σ_i. Thus, transitions from σ_i to σ_{i+1} are said to belong to σ_{i+1}.

5 Ordered CPDS

We generalise *ordered multi-stack pushdown systems* [7]. Intuitively, we can only remove characters from stack i whenever all stacks $j < i$ are empty.

▶ **Definition 5.1** (Ordered CPDS). An order-n *ordered CPDS* (n-OCPDS) is an n-MCPDS $\mathcal{C} = (\mathcal{P}, \Sigma, \mathcal{R}_1, \ldots, \mathcal{R}_m)$ such that a transition from $\langle p, w_1, \ldots, w_m \rangle$ using the rule r on stack i is permitted iff, when r is consuming, for all $1 \leq j < i$ we have $w_j = \perp_n$.

▶ **Theorem 5.2** (Decidability of Reachability Problems). *For n-OCPDSs the control state reachability problem and the global control state reachability problem are decidable.*

We outline the proofs below. In the full paper we show control state reachability uses $\mathcal{O}(2 \uparrow_{m(n-1)} (\ell))$ time, where ℓ is polynomial in the size of the OCPDS, and we have at most $\mathcal{O}(2 \uparrow_{mn} (\ell))$ tuples in the solution to the global problem. First observe that reachability can be reduced to reaching $\langle p_{\text{out}}, \perp_n, \ldots, \perp_n \rangle$ by clearing the stacks at the end of the run.

Control State Reachability Using our notion of ECPDS, we may adapt Atig's inductive algorithm for ordered PDSs [1] for the control state reachability problem. The induction is over the number of stacks. W.l.o.g. we assume that all rules $(p, \perp, o, p')$ of $\mathcal{C}$ have $o = push^n_a$.

In the base case, we have an n-OCPDS with a single stack, for which the global reachability problem is known to be decidable (e.g. [4]).

In the inductive case, we have an n-OCPDS $\mathcal{C}$ with m stacks. By induction, we can decide the reachability problem for n-OCPDSs with fewer than m stacks. We first show how to reduce the problem to reachability analysis of an extended CPDS, and then finally we show how to decide $\mathcal{L}_g \cap \mathcal{L}\left(\mathcal{T}^{A_i}_{t,t'}\right) \neq \emptyset$ using an n-OCPDS with $(m-1)$ stacks.

Consider the mth stack of $\mathcal{C}$. A run of $\mathcal{C}$ can be split into $\sigma_1\tau_1\sigma_2\tau_2\ldots\sigma_\ell\tau_\ell$. During the subruns σ_i, the first $(m-1)$ stacks are non-empty, and during τ_i, the first $(m-1)$ stacks are empty. Moreover, during each σ_i, only generating operations may occur on stack m.

We build an extended CPDS that directly models the mth stack during the τ_i segments where the first $(m-1)$ stacks are empty, and uses rules of the form $(p, a, \mathcal{L}_g, p')$ to encapsulate the behaviour of the σ_i sections where the first $(m-1)$ stacks are non-empty. The $\mathcal{L}_g$ attached to such a rule is the sequence of updates applied to the mth stack during σ_i.

We begin by defining, from the OCPDS $\mathcal{C}$ with m stacks, an OCPDA $\mathcal{C}^L$ with $(m-1)$ stacks. This OCPDA will be used to define the $\mathcal{L}_g$ described above. $\mathcal{C}^L$ simulates a segment σ_i. Since all updates to stack m in σ_i are generating, $\mathcal{C}^L$ need only track its top character, hence only keeps $(m-1)$ stacks. The top character of stack m is kept in the control state, and the operations that would have occurred on stack m are output.

▶ **Definition 5.3** ($\mathcal{C}^L$)**.** Given an n-OCPDS $\mathcal{C} = (\mathcal{P}, \Sigma, \mathcal{R}_1, \ldots, \mathcal{R}_m)$, we define $\mathcal{C}^L$ to be an n-OCPDA with $(m-1)$ stacks $\left(\mathcal{P} \times \Sigma, \Sigma, \mathcal{R}'_1 \cup \mathcal{R}', \mathcal{R}'_2, \ldots, \mathcal{R}'_{m-1}\right)$ over input alphabet $\mathcal{R}_{\mathcal{G}_n}$ where for all i

$$\mathcal{R}'_i = \{((p,a), b, (p, a, noop, p'), o, (p', a)) \mid a \in \Sigma \wedge (p, b, o, p') \in \mathcal{R}_i\}, \text{ and}$$

$$\begin{array}{rcl} \mathcal{R}' & = & \{((p,a), b, r, noop, (p', c)) \mid b \in \Sigma \wedge r = (p, a, rew_c, p') \in \mathcal{R}_m\} \cup \\ & & \{((p,a), b, r, noop, (p', a)) \mid b \in \Sigma \wedge r = (p, a, copy_k, p') \in \mathcal{R}_m\} \cup \\ & & \left\{((p,a), b, r, noop, (p', c)) \;\middle|\; b \in \Sigma \wedge r = \left(p, a, push^k_c, p'\right) \in \mathcal{R}_m\right\} \cup \\ & & \{((p,a), b, r, noop, (p', a)) \mid b \in \Sigma \wedge r = (p, a, noop, p') \in \mathcal{R}_m\} \; . \end{array}$$

We define the language $\mathcal{L}^{b,i}_{p,a,p'}\left(\mathcal{C}^L\right)$ to be the set of words $\gamma_1\ldots\gamma_\ell$ such that there exists a run of $\mathcal{C}^L$ over input $\gamma_1\ldots\gamma_\ell$ from $\langle (p,a), w_1, \ldots, w_{m-1}\rangle$ to $\langle (p', c), \bot_n, \ldots, \bot_n\rangle$ for some c, where $w_i = push^n_b(\bot_n)$ and $w_j = \bot_n$ for all $j \neq i$. This language describes the effect on stack m of a run σ_j from p to p'. (Note, by assumption, all σ_j start with some $push^n_b$.)

We now define the extended CPDS $\mathcal{C}^R$ that simulates $\mathcal{C}$ by keeping track of stack m in its stack and using extended rules based on $\mathcal{C}^L$ to simulate parts of the run where the first $(m-1)$ stacks are not all empty. Note, since all rules operating on $\bot$ (i.e. $(p, \bot, o, p')$) have $o = push^n_b$, rules from $\mathcal{R}_1, \ldots, \mathcal{R}_{m-1}$ may only fire during (or at the start of) the segments where the first $(m-1)$ stacks are non-empty (and thus appear in $\mathcal{R}_{\mathcal{L}_g}$ below).

▶ **Definition 5.4** ($\mathcal{C}^R$)**.** Given an n-OCPDS $\mathcal{C} = (\mathcal{P} \times \Sigma, \Sigma, \mathcal{R}_1, \ldots, \mathcal{R}_m)$ with m stacks, we define $\mathcal{C}^R$ to be an n-ECPDS such that $\mathcal{C}^R = (\mathcal{P}, \Sigma, \mathcal{R}')$ where $\mathcal{R}' = \mathcal{R}_m \cup \mathcal{R}_{\mathcal{L}_g}$ and

$$\mathcal{R}_{\mathcal{L}_g} = \left\{\left(p, a, \mathcal{L}^{b,i}_{p_1,a,p_2}\left(\mathcal{C}^L\right), p_2\right) \;\middle|\; a \in \Sigma \wedge (p, \bot, push^n_b, p_1) \in \mathcal{R}_i \wedge 1 \leq i < m\right\}$$

▶ **Lemma 5.5** ($\mathcal{C}^R$ simulates $\mathcal{C}$)**.** *Given an n-OCPDS $\mathcal{C}$ and control states p_{in}, p_{out}, we have $\langle p_{in}, w\rangle \in Pre^*_{\mathcal{C}^R}(A)$, where A is the $\mathcal{P}$-stack automaton accepting only the configuration $\langle p_{out}, \bot_n\rangle$ iff $\langle p_{in}, \bot_n, \ldots, \bot_n, w\rangle \longrightarrow \cdots \longrightarrow \langle p_{out}, \bot_n, \ldots, \bot_n\rangle$.*

Lemma 5.5 only gives an effective decision procedure if we can decide $\mathcal{L}_g \cap \mathcal{L}\left(\mathcal{T}^{A_i}_{t,t'}\right) \neq \emptyset$ for all rules $(p, a, \mathcal{L}_g, p')$ appearing in $\mathcal{C}^R$. For this, we use a standard product construction between the $\mathcal{C}^L$ associated with $\mathcal{L}_g$, and $\mathcal{T}^{A_i}_{t,t'}$. This gives an ordered CPDS with $(m-1)$ stacks, for which, by induction over the number of stacks, reachability (and emptiness) is decidable. Note, the initial transition of the construction sets up the initial stacks of $\mathcal{C}^L$.

▶ **Definition 5.6** ($\mathcal{C}_\emptyset$)**.** Given the non-emptiness problem $\mathcal{L}^{b,i}_{p_1,a,p_2}\left(\mathcal{C}^L\right) \cap \mathcal{L}\left(\mathcal{T}^{A_i}_{t,t'}\right) \neq \emptyset$, where $top_1(t) = a$, $\mathcal{C}^L = (\mathcal{P} \times \Sigma, \Sigma, \mathcal{R}_1, \ldots, \mathcal{R}_{m-1})$ and $\mathcal{T}^{A_i}_{t,t'} = (T_{A_i}, \mathcal{R}_{\mathcal{G}_n}, \delta, t, t')$, we define an

n-OCPDS $\mathcal{C}_\emptyset = \left(\mathcal{P}^\emptyset, \Sigma, \mathcal{R}^\emptyset_1, \ldots, \mathcal{R}^\emptyset_i \cup \mathcal{R}_{I/O}, \ldots, \mathcal{R}^\emptyset_{m-1}\right)$ where, for all $1 \leq i \leq (m-1)$,

$$\begin{aligned}
\mathcal{P}^\emptyset &= \{p_1, p_2\} \uplus \{(p, t_1) \mid t_1 \in T_{A_i} \wedge control(t_1) = p\} \ , \\
\mathcal{R}_{I/O} &= \{(p_1, \bot, push^n_b, (p_1, t))\} \cup \{((p_2, t), \bot, noop, p_2) \mid t \in T_{A_i}\} \ , \text{ and} \\
\mathcal{R}^\emptyset_i &= \{((p, t_1), c, o, (p', t_2)) \mid ((p, top_1(t_1)), c, r, o, (p', top_1(t_2))) \in \mathcal{R}_i \wedge (t_1, r, t_2) \in \Delta\}
\end{aligned}$$

▶ **Lemma 5.7** (Language Emptiness for OCPDS). *We have $\mathcal{L}^{b,i}_{p_1,a,p_2}(\mathcal{C}^L) \cap \mathcal{L}\left(\mathcal{T}^{A_i}_{t,t'}\right) \neq \emptyset$ iff, in $\mathcal{C}_\emptyset$ from Definition 5.6, we have that $\langle p_2, \bot_n, \ldots \bot_n\rangle$ is reachable from $\langle p_1, \bot_n, \ldots, \bot_n\rangle$.*

Global Reachability We sketch a solution to the global reachability problem, giving a full proof in the full paper. From Lemma 5.5 ($\mathcal{C}^R$ simulates $\mathcal{C}$) we gain a representation $A_m = Pre^*_{\mathcal{C}^R}(A)$ of the set of configurations $\langle p, \bot_n, \ldots, \bot_n, w_m\rangle$ that have a run to $\langle p_{\text{out}}, \bot_n, \ldots, \bot_n\rangle$. Now take any $\langle p, \bot_n, \ldots, \bot_n, w_{m-1}, w_m\rangle$ that reaches $\langle p_{\text{out}}, \bot_n, \ldots, \bot_n\rangle$. The run must pass some $\langle p', \bot_n, \ldots, \bot_n, w'_m\rangle$ with $\langle p', w'_m\rangle$ accepted by A_m. From the product construction above, one can (though not immediately) extract a tuple (p, A_{m-1}, A'_m) such that w_{m-1} is accepted by A_{m-1} and w_m is accepted by A'_m. We repeat this reasoning down to stack 1 and obtain a tuple of the form $(p, A_1, \ldots, A_m)$. We can only obtain a finite set of tuples in this manner, giving a solution to the global reachability problem.

6 Scope-Bounded CPDS

Recently, scope-bounded multi-pushdown systems were introduced [32] and their reachability problem was shown to be decidable. Furthermore, reachability for scope- and phase-bounding was shown to be incomparable [32]. Here we consider scope-bounded CPDS.

A run $\sigma = \sigma_1 \ldots \sigma_\ell$ of an MCPDS is *context-partitionable* when, for each σ_i, if a transition in σ_i is via $r \in \mathcal{R}_j$ on stack j, then all transitions of σ_i are via rules in $\mathcal{R}_j$ on stack j. A *round* is a context-partitioned run $\sigma_1 \ldots \sigma_m$, where during σ_i only $\mathcal{R}_i$ is used. A *round-partitionable* run can be partitioned $\sigma_1 \ldots \sigma_\ell$ where each σ_i is a round. A run of an SBCPDS is such that any character or stack removed from a stack must have been created at most ζ rounds earlier. For this, we define pop- and collapse-rounds for stacks. That is, we mark each stack and character with the round in which it was created. When we copy a stack via $copy_k$, the pop-round of the new copy of the stack is the current round. However, all stacks and characters within the copy of u keep the same pop- and collapse-round as in the original u.

E.g. take $[u]_2$ where $u = [ab]_1$, u and a have pop-round 2, and b has pop-round 1. Suppose in round 3 we use $copy_2$ to obtain $[uu]_2$. The new copy of u has pop-round 3 (the current round), but the a and b appearing in the copy of u still have pop-rounds 2 and 1 respectively. If the scope-bound is 2, the latest each a and the original u could be popped is in round 4, but the new u may be popped in round 5.

We will write ${}_{\mathfrak{p}}w$ for a stack w with pop-round $\mathfrak{p}$ and ${}_{\mathfrak{p},\mathfrak{c}}a$ for a character with pop-round $\mathfrak{p}$ and collapse-round $\mathfrak{c}$. Pop- and collapse-rounds will be sometimes omitted for clarity. Note, the outermost stack will always have pop-round 0. In particular, for all $u :_k v$ in the definition below, the pop-round of v is 0.

▶ **Definition 6.1** (Pop- and Collapse-Round). Given a round-partitioned run $\sigma_1 \ldots \sigma_\ell$ we define inductively the pop- and collapse-rounds. The pop- and collapse-round of each stack and character in the first configuration of σ_1 is 0. Take a transition $\langle p, w\rangle \longrightarrow \langle p', w'\rangle$ with $\langle p', w'\rangle$ in σ_z via a rule (p, a, o, p'). If $o = noop$ then $w = w'$, otherwise when

1. $o = copy_k$ and $w = {}_{\mathfrak{p}}u :_k v$, then $w' = {}_z u :_k ({}_{\mathfrak{p}}u :_k v)$ where ${}_z u = {}_z[{}_{\mathfrak{p}_1}u_1 \ldots {}_{\mathfrak{p}_\ell}u_\ell]_{k-1}$ when ${}_{\mathfrak{p}}u = {}_{\mathfrak{p}}[{}_{\mathfrak{p}_1}u_1 \ldots {}_{\mathfrak{p}_\ell}u_\ell]_{k-1}$.

2. $o = push_b^k$, then $w' = {}_{z,\mathfrak{c}}b^{\left({}_{\mathfrak{p}'}u\right)} :_1 w$ where ${}_{\mathfrak{p}'}u = top_{k+1}(pop_k(w))$ and $\mathfrak{c}$ is the pop-round of $top_k(w)$. (Note, when $k = n$, we know $\mathfrak{p}' = 0$ since the top_{n+1} stack is outermost.)
3. $o = pop_k$, when $w = u :_k v$ then $w' = v$.
4. We set $collapse_k\left(a^{\left({}_{\mathfrak{p}}u'\right)} :_1 u :_{(k+1)} v\right) = {}_{\mathfrak{p}}u' :_{(k+1)} v$ when u is order-k and $1 \leq k < n$; and $collapse_n\left(a^{({}_0u)} :_1 v\right) = {}_0u$ when u is order-n.
5. $o = rew_b$ and $w = {}_{\mathfrak{p},\mathfrak{c}}a^{\left({}_{\mathfrak{p}'}u\right)} :_1 v$, then $w' = {}_{\mathfrak{p},\mathfrak{c}}b^{\left({}_{\mathfrak{p}'}u\right)} :_1 v$.

▶ **Definition 6.2** (Scope-Bounded CPDS). A ζ-scope-bounded n-CPDS (n-SBCPDS) $\mathcal{C}$ is an order-n MCPDS whose runs are all runs of $\mathcal{C}$ that are round-partitionable, that is $\sigma_1 \dots \sigma_\ell$, such that for all z, if a transition in σ_z from $\langle p, w\rangle$ to $\langle p', w'\rangle$ is
1. a pop_k transition with $1 < k \leq n$ and $w = {}_{\mathfrak{p}}u :_k v$, then $z - \zeta \leq \mathfrak{p}$,
2. a pop_1 transition with $w = {}_{\mathfrak{p},\mathfrak{c}}a^u :_1 v$, then $z - \zeta \leq \mathfrak{p}$, or
3. a $collapse_k$ transition with $w = {}_{\mathfrak{p},\mathfrak{c}}a^u :_1 v$, then $z - \zeta \leq \mathfrak{c}$.

La Torre and Napoli's decidability proof for the order-1 case already uses the saturation method [32]. However, while La Torre and Napoli use a forwards-reachability analysis, we must use a backwards analysis. This is because the forwards-reachable set of configurations is in general not regular. We thus perform a backwards analysis for CPDS, resulting in a similar approach. However, the proofs of correctness of the algorithm are quite different.

▶ **Theorem 6.3** (Decidability of Reachability Problems). *For n-OCPDSs the control state reachability problem and the global control state reachability problem are decidable.*

In the full paper we show our non-global algorithm requires $\mathcal{O}(2 \uparrow_{n-1} (\ell))$ space, where ℓ is polynomial in ζ and the size of the SBCPDS, and we have at most $\mathcal{O}(2 \uparrow_n (\ell))$ tuples in the global reachability solution. La Torre and Parlato give an alternative control state reachability algorithm at order-1 using *thread interfaces*, which allows sequentialisation [21] and should generalise order-n, but, does not solve the global reachability problem.

Control State Reachability Fix initial and target control states p_{in} and p_{out}. The algorithm first builds a *reachability graph*, which is a finite graph with a certain kind of path iff p_{out} can be reached from p_{in}. To build the graph, we define layered stack automata. These have states q_p^i for each $1 \leq i \leq \zeta$ which represent the stack contents i rounds later. Thus, a layer automaton tracks the stack across ζ rounds, which allows analysis of scope-bounded CPDSs.

▶ **Definition 6.4** (ζ-Layered Stack Automata). A *ζ-layered stack automaton* is a stack automaton A such that $\mathbb{Q}_n = \left\{q_p^i \mid p \in \mathcal{P} \land 1 \leq i \leq \zeta\right\}$.

A state q_p^i is of layer i. A state q' labelling $q \xrightarrow{q'} Q$ has the same layer as q. We require that there is no $q \xrightarrow{q'} Q$ with $q'' \in Q$ where q is of layer i and q'' is of layer $j < i$. Similarly, there is no $q \xrightarrow[Q_{col}]{a} Q$ with $q' \in Q \cup Q_{col}$ where q is of layer i and q' is of layer $j < i$.

Next, we define several operations from which the reachability graph is constructed. The $\texttt{Predecessor}_j$ operation connects stack j between two rounds. We define for stack j

$$\texttt{Predecessor}_j(A, q_p, q_{p'}) = \texttt{Saturate}_j\left(\texttt{EnvMove}\left(\texttt{Shift}(A), q_{p_1}^1, q_{p_2}^2\right)\right)$$

where definitions of `Shift`, `EnvMove` and $\texttt{Saturate}_j$ are given in the full paper. `Shift` moves transitions in layer i to layer $(i+1)$. E.g. $q_p^1 \xrightarrow{q} \left\{q_{p'}^2\right\}$ would become $q_p^2 \xrightarrow{q} \left\{q_{p'}^3\right\}$. Moreover, transitions involving states in layer ζ are removed. This is because the stack elements in layer ζ will "go out of scope". `EnvMove` adds a new transition (analogously to a (p_1, a, rew_a, p_2) rule) corresponding to the control state change from p_1 to p_2 effected by the runs over the other stacks between the current round and the next (hence layers 1 and 2 in the definition above). $\texttt{Saturate}_j$ gets by saturation all configurations of stack j that can reach via $\mathcal{R}_j$ the stacks accepted from the layer-1 states of its argument (i.e. saturation using initial states $\left\{q^1_p \mid p \in \mathcal{P}\right\}$, which accept stacks from the next round).

The current layer automaton represents a stack across up to ζ rounds. The predecessor operation adds another round on to the front of this representation. A key new insight in our proofs is that if a transition goes to a layer i state, then it represents part of a run where the stack read by the transition is removed in i rounds time. Thus, if we add a transition at layer 0 (were it to exist) that depends on a transition of layer ζ, then the push or copy operation would have a corresponding pop $(\zeta+1)$ scopes away. Scope-bounding forbids this.

The Reachability Graph The reachability graph $\mathcal{G}^{p_{\text{out}}}_{\mathcal{C}} = (\mathcal{V}, \mathcal{E})$ has vertices $\mathcal{V}$ and edges $\mathcal{E}$. Firstly, $\mathcal{V}$ contains some *initial* vertices $(p_0, A_1, p_1, \ldots, p_{m-1}, A_m, p_m)$ where $p_m = p_{\text{out}}$, and for all $1 \leq i \leq m$ we have that A_i is the layer automaton $\texttt{Saturate}_i(A)$ where for all w, A accepts $\langle p_i, w\rangle$ from $q^1_{p_i}$. Furthermore, we require that there is some w such that $\langle p_{i-1}, w\rangle$ is accepted by A_i from $q^1_{p_i}$. That is, there is a run from $\langle p_{i-1}, w\rangle$ to p_i. Intuitively, initial vertices model the final round of a run to p_{out} with context switches at $p_0, \ldots, p_m$.

The complete set $\mathcal{V}$ is the set of all tuples $(p_0, A_1, p_1, \ldots, p_{m-1}, A_m, p_m)$ where there is some w such that $\langle p_{i-1}, w\rangle$ is accepted by A_i from state $q^1_{p_{i-1}}$. To ensure finiteness, we can bound A_i to at most N states. The value of N is $\mathcal{O}(2 \uparrow_{n-1} (\ell))$ where ℓ is polynomial in ζ and the size of $\mathcal{C}$. We give a full definition of N and proof in the full paper.

We have an edge from a vertex $(p_0, A_1, \ldots, A_m, p_m)$ to $(p'_0, A'_1, \ldots, A'_m, p'_m)$ whenever $p_m = p'_0$ and for all i we have $A_i = \texttt{Predecessor}_i\left(A'_i, q_{p_i}, q_{p'_{i-1}}\right)$. An edge means the two rounds can be concatenated into a run since the control states and stack contents match up.

▶ **Lemma 6.5** (Simulation by $\mathcal{G}^{p_{\text{out}}}_{\mathcal{C}}$). *Given a scope-bounded CPDS $\mathcal{C}$ and control states p_{in}, p_{out}, there is a run of $\mathcal{C}$ from $\langle p_{in}, w_1, \ldots, w_m\rangle$ to $\langle p_{out}, w'_1, \ldots, w'_m\rangle$ for some $w'_1, \ldots, w'_m$ iff there is a path in $\mathcal{G}^{p_{out}}_{\mathcal{C}}$ to a vertex $(p_0, A_1, \ldots, A_m, p_m)$ with $p_0 = p_{in}$ from an initial vertex where for all i we have $\langle p_{i-1}, w_i\rangle$ accepted from $q^1_{p_i}$ of A_i.*

Global Reachability The $(p_0, A_1, p_1, \ldots, p_{m-1}, A_m, p_m)$ in $\mathcal{G}^{p_{\text{out}}}_{\mathcal{C}}$ reachable from an initial vertex are finite in number. We know by Lemma 6.5 that there is such a vertex accepting all $\langle p_{i-1}, w_i\rangle$ iff $\langle p_0, w_1, \ldots, w_m\rangle$ can reach the target control state. Let χ be the set of tuples $(p_0, A_1, \ldots, A_m)$ for each reachable vertex as above, where A_i is restricted to the initial state $q^1_{p_{i-1}}$. This is a regular solution to the global control state reachability problem.

7 Conclusion

We have shown decidability of global reachability for ordered and scope-bounded collapsible pushdown systems (and phase-bounded in the full article). This leads to a challenge to find a general framework capturing these systems. Furthermore, we have only shown upper-bound results. Although, in the case of phase-bounded systems, our upper-bound matches that of Seth for CPDSs without collapse [29], we do not know if it is optimal. Obtaining matching lower-bounds is thus an interesting though non-obvious problem. Recently, a more relaxed notion of scope-bounding has been studied [20]. It would be interesting to see if we can extend our results to this notion. We are also interested in developing and implementing algorithms that may perform well in practice.

Acknowledgments Many thanks for initial discussions with Arnaud Carayol and to the referees for their helpful remarks. This work was supported by Fond. Sci. Math. Paris; AMIS [ANR 2010 JCJC 0203 01 AMIS]; FREC [ANR 2010 BLAN 0202 02 FREC]; VAPF (Région IdF); and the Engineering and Physical Sciences Research Council [EP/K009907/1].

References

1. M. F. Atig. Model-checking of ordered multi-pushdown automata. In *LMCS*, 8(3), 2012.
2. M. F. Atig, B. Bollig, and P. Habermehl. Emptiness of multi-pushdown automata is 2etime-complete. In *DLT*, 2008.
3. T. Ball and S. K. Rajamani. The SLAM project: Debugging system software via static analysis. In *POPL*, 2002.
4. A. Bouajjani, J. Esparza, and O. Maler. Reachability analysis of pushdown automata: Application to model-checking. In *CONCUR*, 1997.
5. A. Bouajjani, J. Esparza, and T. Touili. A generic approach to the static analysis of concurrent programs with procedures. *SIGPLAN Not.*, 38(1):62–73, 2003.
6. A. Bouajjani, M. Müller-Olm, and T. Touili. Regular symbolic analysis of dynamic networks of pushdown systems. *CONCUR*, 2005.
7. 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.
8. C. H. Broadbent, A. Carayol, M. Hague, and O. Serre. C-SHORe: A collapsible approach to verifying higher-order programs. To appear in *ICFP*, 2013.
9. C. H. Broadbent, A. Carayol, M. Hague, and O. Serre. A saturation method for collapsible pushdown systems. In *ICALP*, 2012.
10. A. Cyriac, P. Gastin, and K. N. Kumar. MSO decidability of multi-pushdown systems via split-width. In *CONCUR*, 2012.
11. W. Damm. The IO- and OI-hierarchies. *Theor. Comput. Sci.*, 20:95–207, 1982.
12. A. Finkel, B. Willems, and P. Wolper. A direct symbolic approach to model checking pushdown systems. In *INFINITY*, 1997.
13. M. Hague. Saturation of concurrent collapsible pushdown systems, 2013. arXiv:1310.2631 [cs.FL].
14. M. Hague, A. S. Murawski, C.-H. L. Ong, and O. Serre. Collapsible pushdown automata and recursion schemes. In *LICS*, 2008.
15. A. Heußner, J. Leroux, A. Muscholl, and G. Sutre. Reachability analysis of communicating pushdown systems. In *FOSSACS*, 2010.
16. V. Kahlon. Reasoning about threads with bounded lock chains. In *CONCUR*, 2011.
17. T. Knapik, D. Niwinski, P. Urzyczyn, and I. Walukiewicz. Unsafe grammars and panic automata. In *ICALP*, 2005.
18. N. Kobayashi. Higher-order model checking: From theory to practice. In *LICS*, 2011.
19. N. Kobayashi and A. Igarashi. Model-Checking Higher-Order Programs with Recursive Types In *ESOP*, 2013.
20. S. La Torre and M. Napoli. A temporal logic for multi-threaded programs. In *IFIP TCS*, 2012.
21. S. La Torre and G. Parlato. Scope-bounded multistack pushdown systems: Fixed-point, sequentialization, and tree-width. In *FSTTCS*, 2012.
22. P. Madhusudan and G. Parlato. The tree width of auxiliary storage. In *POPL*, 2011.
23. C.-H. L. Ong. On model-checking trees generated by higher-order recursion schemes. In *LICS*, 2006.
24. L. Ong. Recursion schemes, collapsible pushdown automata and higher-order model checking. In *LATA*, 2013.
25. S. Qadeer. The case for context-bounded verification of concurrent programs. In *SPIN*, 2008.
26. S. Qadeer and J. Rehof. Context-bounded model checking of concurrent software. In *TACAS*, 2005.
27. S. Schwoon. *Model-checking Pushdown Systems.* PhD thesis, Technical University of Munich, 2002.
28. K. Sen and M. Viswanathan. Model checking multithreaded programs with asynchronous atomic methods. In *CAV*, 2006.
29. A. Seth. Games on higher order multi-stack pushdown systems. In *RP*, 2009.
30. A. Seth. Global reachability in bounded phase multi-stack pushdown systems. In *CAV*, 2010.
31. S. L. Torre, P. Madhusudan, and G. Parlato. A robust class of context-sensitive languages. In *LICS*, 2007.
32. S. L. Torre and M. Napoli. Reachability of multistack pushdown systems with scope-bounded matching relations. In *CONCUR*, 2011.