Senescent Ground Tree Rewrite Systems

M. Hague

Royal Holloway University of London
matthew.hague@rhul.ac.uk

Abstract

Ground Tree Rewrite Systems with State are known to have an undecidable control state reachability problem. Taking inspiration from the recent introduction of scope-bounded multi-stack pushdown systems, we define *Senescent Ground Tree Rewrite Systems*. These are a restriction of ground tree rewrite systems with state such that nodes of the tree may no longer be rewritten after having witnessed an *a priori* fixed number of control state changes. As well as generalising scope-bounded multi-stack pushdown systems, we show — via reductions to and from reset Petri-nets — that these systems have an Ackermann-complete control state reachability problem. However, reachability of a regular set of trees remains undecidable.

Contents

1 Introduction

The study of reachability problems for infinite state systems, such as Turing machines, has often used strings to represent system states. In seminal work, Büchi showed the decidability of reachability for pushdown systems [36]. A state (or configuration) of a pushdown system is represented by a control state (from a finite set) and a stack over a given finite alphabet. In this case, a stack can be considered a word and one stack is obtained from another by replacing a prefix w of the stack with another word w'. In fact, the reachability problem is in P-time [9, 18].

Pushdown systems allow the control-flow of first-order programs to be accurately modelled [22] and have been well-studied in the automata-theoretic approach to software model checking (E.g. [9, 18, 16, 37]). Many scalable model checkers for pushdown systems have been implemented, and these tools (e.g. Bebop [7] and Moped [42]) are an essential back-end component of celebrated model checkers such as SLAM [6].

A natural and well-studied generalisation of these ideas is to use a tree representation of system states. This approach was first considered by Brainerd, who, generalising Büchi's result, showed decidability of reachability for Ground Tree Rewrite Systems[1] (GTRS) [11]. In these systems, each transition replaces a complete subtree of the state with another. Thus, these systems generalise pushdown systems and allow the analysis of tree manipulating programs. As in the pushdown case, reachability is solvable in P-time [28].

Unfortunately, these tree generalisations of pushdown automata do not allow a control state in their configurations. Instead, the pushdown system's control state must be encoded as the leaf of the tree. This is for good reason: when a control state external to the tree is permitted, reachability is immediately undecidable. This is because one can easily simulate a two-stack pushdown system with a tree that contains a branch for each stack. It is well known that a two-stack pushdown system can simulate a Turing machine, and thus reachability is undecidable.

However, due to the increasing importance of concurrent systems — where each thread requires its own stack — there has been renewed interest in identifying classes of multi-stack pushdown systems for which reachability becomes decidable. A seminal notion in this regard is that of *context-bounding*. This underapproximates a concurrent system by bounding the number of context switches that may occur [38]. It is based on the observation that most real-world bugs require only a small number of thread interactions [35]. By considering only context-bounded runs of a multi-stack pushdown system, the reachability problem becomes NP-complete.

[1] Also known as Ground Term Rewrite Systems

In recent work [43], Lin (formerly known as To) observed that a GTRS modelling a context-bounded multi-stack pushdown system has an underlying control state graph that is 1-weak. A 1-weak automaton is an automaton whose control state graph contains no cycles except for self-loops [30]. Intuitively, the control state is only used to manage context-switches, and hence a 1-weak control state graph suffices. Moreover, the reachability problem for GTRS with control states is decidable with such a control state graph [43]. Indeed, the problem remains NP-complete [27].

As well as the notion of context-bounding there are many more relaxed restrictions on multi-stack pushdown behaviours for which reachability also remains decidable. Of particular interest is *scope-bounding* [24]. In this setting, we fix a bound k and insist that an item may only be removed from the stack if it was pushed at most k context switches earlier. Thus, an arbitrary number of context switches may occur, and the underlying control state graph is no longer 1-weak. In this case, by relaxing the restriction on the control state behaviours, the complexity of reachability is increased from NP to PSPACE.

In this work we study how to generalise scope-bounding to GTRS with control states. We obtain a model of computation reminiscent of a tree growing in nature: it begins with a green shoot, which may grow and change. As this shoot ages, it becomes hardened and forms the trunk of the tree. From this trunk, new green shoots grow, and — via leaves that may fall and grow again — remain changeable. If a shoot lives long enough, it hardens and forms a new (fixed) branch of the tree.

Thus, we define *senescent ground tree rewrite systems*. The passage of time is marked by changes to the control state. If a node remains unchanged for a fixed number k of changes, it becomes unchangeable – that is, part of a hardened branch of the tree.

These systems naturally generalise scope-bounded pushdown systems while also allowing for additional features such as dynamic thread creation. To our knowledge, they also provide the most precise under-approximation of GTRS with control states currently known to have a decidable control state reachability problem, and thus may be used in the analysis of tree-manipulating programs.

We show, via inter-reductions with reset Petri-nets, that the control state reachability problem for senescent GTRS is Ackermann-complete while the reachability of a regular set of trees is undecidable. This increase in modelling power is in sharp contrast to the analogous restrictions for pushdown systems, where the increase is much more modest. That is, from The Faithful Gardner [17]:

> *To be poor and be without trees, is to be the most starved human being in the world. To be poor and have trees, is to be completely rich in ways that money can never buy.*

2 Related Work

Abdulla *et al.* [1] define a regular model checking algorithm for tree automatic structures. That is, the transition relation is given by a regular tree transducer. They give a reachability algorithm that is complete when there is a fixed bound k such that, during any run of the system, a node is changed at most k times. This has flavours of the systems we define here. However, in our model, a node may be changed an arbitrary number of times. In fact, we impose a limit on the extent to which a node may remain *unchanged*. It is not clear how the two models compare, and such a comparison is an interesting avenue of future work.

Atig *et al.* [5] consider a model of multi-stack pushdown systems with dynamic thread

creation. Decidability of the reachability problem is obtained by allowing each thread to be active at most k times during a run. As we show in Section 5.1, we can consider each thread to be a branch of the tree, and context switches correspond to control state changes. However, while in Atig *et al.*'s model a thread may be active for *any* k context switches (as long as it's inactive for the others), in our model a thread will begin to suffer restrictions after the k next context switches (though may be active for an arbitrary number). Perhaps counter-intuitively, our restriction actually increases expressivity: Atig *et al.* show inter-reducibility between reachability in their model and Petri-net coverability, while in our model the more severe restriction allows us to inter-reduce with coverability of *reset* Petri-nets.

The scope-bounded restriction has recently been relaxed for multi-stack pushdown systems by La Torre and Napoli [25]. In their setting, a character that is popped from a particular stack may must have been pushed within k active contexts of that stack. In particular, this allows for an unbounded number of context switches to occur between a push and a pop, as long as the stack involved is only active in up to k of those contexts. However, it is unclear what such a relaxation would mean in the context of senescent GTRS.

GTRS have been extensively studied as generators of graphs (e.g. [28]) and and are known to have decidable verification problems for repeated reachability [28], first-order logic [15], confluence [14], &c. However, LTL and CTL model checking are undecidable [10, 20, 28]. They also have intimate connections (e.g. [28, 20]) with the Process Rewrite Systems Hierarchy [31].

There are several differing restrictions to multi-stack pushdown systems with decidable verification problems. Amongst these are phase-bounded [44] and ordered [12] (corrected in [4]) pushdown systems. There are also generic frameworks — that bound the tree- [29] or split-width [13] of the interactions between communication and storage — that give decidability for all communication architectures that can be defined within them.

3 Preliminaries

We write $\mathbb{N}$ to denote the set of natural numbers and $\mathbb{N}_+$ to denote the set of strictly positive natural numbers. Given a word language $\mathcal{L} \subseteq \Sigma^*$ for some alphabet Σ and a word $w \in \Sigma^*$, let $w \cdot \mathcal{L} = \{ww' \mid w' \in \mathcal{L}\}$. For a given set S, let $|S|$ denote the cardinality of the set.

In the cases when the dimension is clear, we will write $\vec{0}$ to denote the tuple $(0, \ldots, 0)$ and $\vec{i}$ to denote the tuple $(n_1, \ldots, n_m)$ where all $n_j = 0$ for all $j \neq i$ and $n_i = 1$.

We will denote by $\mathbf{F}_\omega$ both the Ackermann function and the class of problems solvable in $\mathbf{F}_\omega$-time. Following Schmitz and Schnoebelen [39], we have the class $\mathbf{F}_\omega$ of problems computable in Ackermannian time. This class is closed under primitive-recursive reductions.

3.1 Trees and Automata

3.1.1 Regular Automata and Parikh Images

A *regular automaton* is a tuple $\mathcal{A} = (\mathcal{Q}, \Gamma, \Delta, q_0, \mathcal{F})$ where $\mathcal{Q}$ is a finite set of states, Γ is a finite output alphabet, $\Delta \subseteq \mathcal{Q} \times \Gamma \times \mathcal{Q}$ is a transition relation, $q_0 \in \mathcal{Q}$ is an initial state and $\mathcal{F} \subseteq \mathcal{Q}$ is a set of final states.

We write $q \xrightarrow{a} q'$ to denote a transition $(q, a, q') \in \Delta$. A run from $q_1 \in \mathcal{Q}$ over a word $w = a_1 \ldots a_h$ is a sequence

$$q_1 \xrightarrow{a_1} \cdots \xrightarrow{a_h} q_{h+1} \ .$$

A run is *accepting* whenever $q_{h+1} \in \mathcal{F}$. The language $\mathcal{L}(\mathcal{A})$ of $\mathcal{A}$ is the set of words $w \in \Gamma^*$ such that there is an accepting run of $\mathcal{A}$ over w from q_0.

For a word $w \in \Gamma^*$ for some alphabet Γ, we define $|w|_\gamma$ to be the number of occurrences of γ in w. Given a fixed linear ordering $\gamma_1, \ldots, \gamma_m$ over $\Gamma = \{\gamma_1, \ldots, \gamma_m\}$ and a word $w \in \Gamma^*$, we define $\textsc{Parikh}(w) = \left(|w|_{\gamma_1}, \ldots, |w|_{\gamma_m}\right)$. Given a language $\mathcal{L} \subseteq \Gamma^*$, we define $\textsc{Parikh}(\mathcal{L}) = \{\textsc{Parikh}(w) \mid w \in \mathcal{L}\}$. Finally, given a regular automaton $\mathcal{A}$, we define $\textsc{Parikh}(\mathcal{A}) = \textsc{Parikh}(\mathcal{L}(\mathcal{A}))$.

3.1.2 Trees

A *ranked alphabet* is a finite set of characters Σ together with a rank function $\texttt{rank} : \Sigma \mapsto \mathbb{N}$. A *tree domain* $D \subset \mathbb{N}_+^*$ is a nonempty finite subset of $\mathbb{N}_+^*$ that is both *prefix-closed* and *younger-sibling-closed*. That is, if $ni \in D$, then we also have $n \in D$ and, for all $1 \leq j \leq i$, $nj \in D$ (respectively). A *tree* over a ranked alphabet Σ is a pair $T = (D, \lambda)$ where D is a tree domain and $\lambda : D \mapsto \Sigma$ such that for all $n \in D$, if $\lambda(n) = a$ and $\texttt{rank}(a) = m$ then n has exactly m children (i.e. $nm \in D$ and $n(m+1) \notin D$). Let $\textsc{Trees}(\Sigma)$ denote the set of trees over Σ.

Given a node n and trees $T_1, \ldots, T_m$, we will often write $n(T_1, \ldots, T_m)$ to denote the tree with root node n and left-to-right child sub-trees $T_1, \ldots, T_m$. When n is labelled a, we may also write $a(T_1, \ldots, T_m)$ to denote the same tree. We will often simply write a to denote the tree with a single node labelled a. Finally, let $\mathcal{E}$ denote the empty tree.

3.1.3 Context Trees

A *context tree* over the alphabet Σ with context variables $x_1, \ldots, x_m$ is a tree $C = (D, \lambda)$ over $\Sigma \uplus \{x_1, \ldots, x_m\}$ such that for each $1 \leq i \leq m$ we have $\texttt{rank}(x_i) = 0$ and there exists a unique *context node* n_i such that $\lambda(n_i) = x_i$. We will denote such a tree $C[x_1, \ldots, x_m]$.

Given trees $T_i = (D_i, \lambda_i)$ for each $1 \leq i \leq m$, we denote by $C[T_1, \ldots, T_m]$ the tree T' obtained by filling each variable x_i with the tree T_i. That is, $T' = (D', \lambda')$ where $D' = D \cup n_1 \cdot D_1 \cup \cdots \cup n_m \cdot D_m$ and

$$\lambda'(n) = \begin{cases} \lambda(n) & n \in D \wedge \forall i. n \neq n_i \\ \lambda_i(n') & n = n_i n' \end{cases}.$$

3.1.4 Tree Automata

A *bottom-up nondeterministic tree automaton* (NTA) over a ranked alphabet Σ is a tuple $\mathcal{T} = (\mathcal{Q}, \Delta, \mathcal{F})$ where $\mathcal{Q}$ is a finite set of states, $\mathcal{F} \subseteq \mathcal{Q}$ is a set of final (accepting) states, and Δ is a finite set of rules of the form $(q_1, \ldots, q_m) \xrightarrow{a} q$ where $q_1, \ldots, q_m, q \in \mathcal{Q}$, $a \in \Sigma$ and $\texttt{rank}(a) = m$. A *run* of $\mathcal{T}$ on a tree $T = (D, \lambda)$ is a mapping $\rho : D \mapsto \mathcal{Q}$ such that for all $n \in D$ labelled $\lambda(n) = a$ with $\texttt{rank}(a) = m$ we have $(\rho(n1), \ldots, \rho(nm)) \xrightarrow{a} \rho(n)$. It is accepting if $\rho(\varepsilon) \in \mathcal{F}$. The *language* defined by a tree automaton $\mathcal{T}$ over alphabet Σ is a set $\mathcal{L}(\mathcal{T}) \subseteq \textsc{Trees}(\Sigma)$ over which there exists an accepting run of $\mathcal{T}$. A set of trees $\mathcal{L}$ is *regular* iff there is a tree automaton $\mathcal{T}$ such that $\mathcal{L}(\mathcal{T}) = \mathcal{L}$.

For any tree T, let $\mathcal{T}_T$ be a tree automaton accepting only the tree T.

3.2 Reset Petri-Nets

We give a simplified presentation of reset Petri-nets as counter machines with increment, decrement and reset operations. This can easily been seen to be equivalent to the standard definition [3].

Given a set $X = \{x_1, \ldots, x_m\}$ of counter variables, we define the set $\textsc{Op}_X$ of counter operations to be $\{incr(x), decr(x), res(x) \mid x \in X\}$.

Definition 3.1 (Reset Petri Nets)**.** *A* reset Petri net *is a tuple $\mathcal{N} = (\mathcal{Q}, X, \Delta)$ where $\mathcal{Q}$ is a finite set of control states, X is a finite set of counter variables, and $\Delta \subseteq \mathcal{Q} \times 2^{\textsc{Op}_X} \times \mathcal{Q}$ is a transition relation.*

A configuration of a reset Petri net is a pair (q, π) where $q \in \mathcal{Q}$ is a control state and $\pi : X \to \mathbb{N}$ is a marking assigning values to counter variables. We write $p \underset{\widetilde{o}}{\to} p'$ to denote a rule $(p, \widetilde{o}, p') \in \Delta$ and omit the set notation when the set of counter operations is a singleton. There is a transition $(p, \pi) \longrightarrow (p', \pi')$ whenever we have $p \underset{\widetilde{o}}{\to} p' \in \Delta$ and there are markings π_1, π_2 such that

- we have
$$\pi_1(x) = \begin{cases} \pi(x) - 1 & \text{if } decr(x) \in \widetilde{o} \text{ and } \pi_1(x) > 0 \\ \pi(x) & \text{if } decr(x) \notin \widetilde{o} \end{cases}$$
and
- we have
$$\pi_2(x) = \begin{cases} 0 & \text{if } res(x) \in \widetilde{o} \\ \pi_1(x) & \text{if } res(x) \notin \widetilde{o} \end{cases}$$
and
- we have
$$\pi'(x) = \begin{cases} \pi_2(x) + 1 & \text{if } incr(x) \in \widetilde{o} \\ \pi_2(x) & \text{if } incr(x) \notin \widetilde{o} \end{cases}$$

Note, in particular, operations are applied in the order $decr(x), res(x), incr(x)$, and if x is 0, attempting to apply $decr(x)$ will cause the Petri net to become stuck. We write $(q, \pi) \longrightarrow^* (q', \pi')$ whenever there is a run $(q, \pi) \longrightarrow \cdots \longrightarrow (q', \pi')$ of $\mathcal{N}$.

Given two markings π and π', we say π *covers* π', written $\pi' \leq \pi$ whenever, for all x we have $\pi'(x) \leq \pi(x)$.

Definition 3.2 (Coverability Problem)**.** *Given a reset Petri-net $\mathcal{N}$, and configurations (q, π) and (q', π') the* coverability problem *is to decide whether there exists a run $(q, \pi) \longrightarrow^* (q', \pi'')$ of $\mathcal{N}$ such that $\pi' \leq \pi''$.*

The coverability problem for reset Petri nets is decidable via the Karp-Miller algorithm [23] whose complexity is bounded by $\mathbf{F}_\omega$ [32]. In fact, the coverability problem for reset Petri nets is $\mathbf{F}_\omega$-complete [40, 41]. In contrast, the reachability problem (defined below) is undecidable [3].

Definition 3.3 (Reachability Problem)**.** *Given a reset Petri-net $\mathcal{N}$, and configurations (q, π) and (q', π') the* reachability problem *is to decide whether there exists a run $(q, \pi) \longrightarrow^* (q', \pi')$ of $\mathcal{N}$.*

In the following, we will write π_0 for the marking assigning zero to all counters.

3.3 Ground Tree Rewrite Systems with State

In this work, we will actually consider a generalisation of GTRS where regular automata appear in the rewrite rules. In this way, a single rewrite rule may correspond to an infinite number of rewrite rules containing concrete trees on their left- and right-hand sides. Such an extension is frequently considered (e.g. [14, 28, 27]). It can be noted that our lower bound results only use tree automata that accept a singleton set of trees, and thus we do not increase our lower bounds due to this generalisation.

3.3.1 Basic Model

A Ground Tree Rewrite System with State maintains a tree over a given alphabet Σ and a control state from a finite set. Each transition may update the control state and rewrite a part of the tree. Rewriting a tree involves matching a sub-tree of the current tree and replacing it with a new tree. Note, that since we are considering ranked trees, a sub-tree cannot be erased by a rewrite rule, since this would make the tree inconsistent w.r.t the ranks of the tree labels.

Definition 3.4 (Ground Tree Rewrite System with State)**.** *A* ground tree rewrite system with state *(sGTRS) is a tuple $G = (\mathcal{P}, \Sigma, \mathcal{R})$ where $\mathcal{P}$ is a finite set of control states, Σ is a finite ranked alphabet, and $\mathcal{R}$ is a finite set of rules of the form $(p_1, \mathcal{T}_1) \longrightarrow (p_2, \mathcal{T}_2)$ where $p_1, p_2 \in \mathcal{P}$ and $\mathcal{T}_1, \mathcal{T}_2$ are NTAs over Σ such that $\mathcal{E} \notin \mathcal{L}(\mathcal{T}_1) \cup \mathcal{L}(\mathcal{T}_2)$.*

A *configuration* of a sGTRS is a pair $(p, T) \in \mathcal{P} \times \textsc{Trees}(\Sigma)$. We have a *transition* $(p_1, T_1) \longrightarrow (p_2, T_2)$ whenever there is a rule $(p_1, \mathcal{T}_1) \longrightarrow (p_2, \mathcal{T}_2) \in \mathcal{R}$ such that $T_1 = C[T_1']$ for some context C and tree $T_1' \in \mathcal{L}(\mathcal{T}_1)$ and $T_2 = C[T_2']$ for some tree $T_2' \in \mathcal{L}(\mathcal{T}_2)$.

A *run* of an sGTRS is a sequence $(p_1, T_1) \longrightarrow \cdots \longrightarrow (p_h, T_h)$ such that for all $1 \leq i < h$ we have $(p_i, T_i) \longrightarrow (p_{i+1}, T_{i+1})$ is a transition of G. We write $(p, T) \longrightarrow^* (p', T')$ whenever there is a run from (p, T) to (p', T').

We are interested in both the control state reachability problem and the regular reachability problem.

Definition 3.5 (Control State Reachability Problem)**.** *Given an sGTRS G, an initial configuration (p_{src}, T_{src}) of G and a target control state p_{snk}, the* control state reachability problem *asks whether there is a run $(p_{src}, T_{src}) \longrightarrow^* (p_{snk}, T)$ of G for some tree T.*

Definition 3.6 (Regular Reachability Problem)**.** *Given an sGTRS G, an initial configuration (p_{src}, T_{src}) of G, a target control state p_{snk}, and tree automaton $\mathcal{T}$, the* regular reachability problem *is to decide whether there exists a run $(p_{src}, T_{src}) \longrightarrow^* (p_{snk}, T)$ for some $T \in \mathcal{L}(\mathcal{T})$.*

3.3.2 Output Symbols

We may also consider sGTRSs with output symbols.

Definition 3.7 (Ground Tree Rewrite System with State and Outputs)**.** *A* ground tree rewrite system with state and outputs *is a tuple $G = (\mathcal{P}, \Sigma, \Gamma, \mathcal{R})$ where $\mathcal{P}$ is a finite set of control states, Σ is a finite ranked alphabet, Γ is a finite alphabet of output symbols, and $\mathcal{R}$ is a finite set of rules of the form $(p_1, \mathcal{T}_1) \xrightarrow{\gamma} (p_2, \mathcal{T}_2)$ where $p_1, p_2 \in \mathcal{P}$, $\gamma \in \Gamma$, and $\mathcal{T}_1, \mathcal{T}_2$ are NTAs over Σ such that $\mathcal{E} \notin \mathcal{L}(\mathcal{T}_1) \cup \mathcal{L}(\mathcal{T}_2)$.*

As before, a *configuration* of an sGTRS is a pair $(p, T) \in \mathcal{P} \times \textsc{Trees}(\Sigma)$. We have a *transition* $(p_1, T_1) \xrightarrow{\gamma} (p_2, T_2)$ whenever there is a rule $(p_1, \mathcal{T}_1) \xrightarrow{\gamma} (p_2, \mathcal{T}_2) \in \mathcal{R}$ such that $T_1 = C[T_1']$ for some context C and tree $T_1' \in \mathcal{L}(\mathcal{T}_1)$ and $T_2 = C[T_2']$ for some tree $T_2' \in \mathcal{L}(\mathcal{T}_2)$. A *run* over

$\gamma_1 \dots \gamma_{h-1}$ is a sequence $(p_1, T_1) \xrightarrow{\gamma_1} \cdots \xrightarrow{\gamma_{h-1}} (p_h, T_h)$ such that for all $1 \leq i < h$ we have $(p_i, T_i) \xrightarrow{\gamma_i} (p_{i+1}, T_{i+1})$ is a transition of G. We write $(p, T) \xrightarrow{\gamma_1 \dots \gamma_h} (p', T')$ whenever there is a run from (p, T) to (p', T') over the sequence of output symbols $\gamma_1 \dots \gamma_h$. Let ε denote the empty output symbol.

3.3.3 Weakly Extended Ground Tree Rewrite Systems

The control state and regular reachability problems for sGTRS are known to be undecidable [10, 20]. The problems become NP-complete for *weakly-synchronised* sGTRS [27], where the underlying control state graph (where there is an edge between p_1 and p_2 whenever there is a transition $(p_1, \mathcal{T}_1) \longrightarrow (p_2, \mathcal{T}_2)$) may only have cycles of length 1 (i.e. self-loops).

More formally, we define the *underlying control graph* of a sGTRS $G = (\mathcal{P}, \Sigma, \Gamma, \mathcal{R})$ as a tuple $(\mathcal{P}, \Delta)$ where $\Delta = \left\{ (p, p') \;\middle|\; (p, \mathcal{T}) \xrightarrow{\gamma} (p', \mathcal{T}') \in \mathcal{R} \right\}$. Note, the underlying control graph of a sGTRS without output symbols can be defined by simply omitting Γ and γ.

Definition 3.8 (Weakly Extended GTRS [27])**.** *An sGTRS (with or without output symbols) is* weakly extended *if its underlying control graph $(\mathcal{P}, \Delta)$ is such that all paths*

$$(p_1, p_2)\,(p_2, p_3) \dots (p_{h-2}, p_{h-1})\,(p_{h-1}, p_h) \in \Delta^*$$

with $p_1 = p_h$ satisfy $p_i = p_1$ for all $1 \leq i \leq h$.

A key result of Lin is that the Parikh image of a weakly extended sGTRS with output symbols can be represented by an existential Presburger formula that is constructible in polynomial time. We use this result to obtain regular automata representing the possible outputs of weakly extended sGTRSs, which will be used later in our decidability proofs. In the following lemma, fix an arbitrary linear ordering over the output alphabet of G.

Lemma 3.1 (Parikh Image of Weakly Extended sGTRS)**.** *Given a weakly extended sGTRS G with outputs Γ, control states p_1 and p_2 and tree automata $\mathcal{T}_1$ and $\mathcal{T}_2$, we can construct a regular automaton $\mathcal{A}$ with outputs Γ such that we have some trees $T_1 \in \mathcal{L}(\mathcal{T}_1)$ and $T_2 \in \mathcal{L}(\mathcal{T}_2)$ and a run*

$$(p_1, T_1) \xrightarrow{\gamma_1 \dots \gamma_h} (p_2, T_2)$$

with $\textsc{Parikh}(\gamma_1 \dots \gamma_h) = \vec{v}$ iff $\vec{v} \in \textsc{Parikh}(\mathcal{A})$. Moreover, the size of $\mathcal{A}$ is at most triply exponential in the size of G.

Proof. Directly from Lemma 2 in Lin 2012 [27] we can construct in polynomial time an existential Presburger formula φ with $|\Gamma|$ free variables such that we have some trees $T_1 \in \mathcal{L}(\mathcal{T}_1)$ and $T_2 \in \mathcal{L}(\mathcal{T}_2)$ and a run

$$(p_1, T_1) \xrightarrow{\gamma_1 \dots \gamma_h} (p_2, T_2)$$

with $\textsc{Parikh}(\gamma_1 \dots \gamma_h) = \vec{v}$ iff $\vec{v}$ satisfies φ. We know from Ginsburg and Spanier [19] that the set of satisfying assignments to an existential Presburger formula can be described by a semilinear set (and vice-versa). That is, there exists some m and for all $1 \leq i \leq m$ there are vectors of natural numbers $\vec{u}_i$ and $\vec{u}_i^1, \dots, \vec{u}_i^{n_i}$ such that for all $\vec{v}$, $\vec{v}$ satisfies φ iff there exists some $1 \leq i \leq m$ and constants $\mu_1, \dots, \mu_{n_i} \in \mathbb{N}$ such that $\vec{v} = \vec{u}_i + \mu_1 \cdot \vec{u}_i^1 + \cdots + \mu_{n_i} \cdot \vec{u}_i^{n_i}$. In fact, via algorithms of Pottier [34], it is possible to obtain vectors such that the size of the values appearing in $\vec{u}_i$ and $\vec{u}_i^1, \dots, \vec{u}_i^{n_i}$ are at most doubly exponential in the size of φ (via an exponential translation of φ into DNF, then applying Pottier to gain a bound exponential in the size of the DNF — see Haase [21] or Piskac [33]). Since φ is polynomial in the size of G,

we know that that each vector $\vec{u}_i$ or $\vec{u}_i^j$ has elements at most doubly exponential in size, and since there are at most a triply exponential number of such sets of vectors, we know that m is at most triply exponential in the size of G.

It is straightforward to build, from a semilinear set, a regular automaton $\mathcal{A}$ such that PARIKH($\mathcal{A}$) is equivalent to the set: for each i we have a branch in $\mathcal{A}$ first outputting the appropriate number of characters to describe $\vec{u}_i$, and then passing through a succession of loops each outputting characters describing some $\vec{u}_i^j$. Such an automaton will be at most triply exponential in the size of G. □

4 Senescent Ground Tree Rewrite Systems with State

In this paper we generalise weakly-synchronised sGTRS to define senescent ground tree rewrite systems by incorporating ideas from scope-bounded multi-stack pushdown systems [24], where stack characters may only be accessed if they were created less than a fixed number of context switches previously.

Intuitively, during each transition of a run that changes the control state, the nodes in the tree "age" by one timestep. When the nodes reach a certain (fixed) age, they become fossilised and may no longer be changed by any future transitions.

4.1 Model Definition

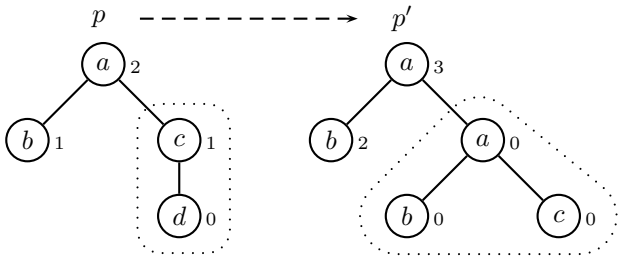
Given a run $(p_1, T_1) \longrightarrow \cdots \longrightarrow (p_h, T_h)$ of an sGTRS, let $C_1, \ldots, C_{h-1}$ be the sequence of tree contexts used in the transitions from which the run was constructed. That is, for all $1 \leq i < h$, we have $T_i = C_i[T_i^{\text{out}}]$ and $T_{i+1} = C_i\big[T_{i+1}^{\text{in}}\big]$ where $(p_i, \mathcal{T}_i) \longrightarrow (p_{i+1}, \mathcal{T}_i')$ was the rewrite rule used in the transition and $T_i^{\text{out}} \in \mathcal{L}(\mathcal{T}_i)$, $T_{i+1}^{\text{in}} \in \mathcal{L}(\mathcal{T}_i')$ were the trees that were used in the tree update.

For a given position (p_i, T_i) in the run and a given node n in the domain of T_i, the *birthdate* of the node is the largest $1 \leq j \leq i$ such that n is in the domain of $C_j\big[T_j^{\text{in}}\big]$ and n is in the domain of $C_j[x]$ only if its label is x. The *age* of a node is the cardinality of the set $\{i' \mid j \leq i' < i \wedge p_{i'} \neq p_{i'+1}\}$. That is, the age is the number of times the control state changed between the jth and the ith configurations in the run. This is illustrated in Figure 1.

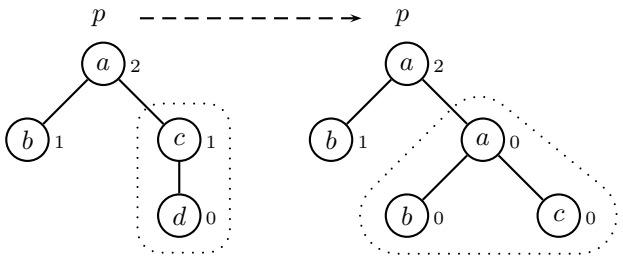
Figure 1 shows two transitions of a senescent GTRS. A configuration is written as its control state (p or p') with the tree appearing below. The label of each node appears in the centre of the node, while the ages of each node appears to the right. The parts of the tree rewritten by the transition appear inside the dotted lines. Figure 1a shows a transition where the control state is changed. This change causes the nodes that are not rewritten to increase their age by 1. The rewritten nodes are given the age 0. Figure 1b shows a transition that does not change the control state. Notice that, in this case, the nodes that are not rewritten maintain the same age.

A lifespan restricted run with a lifespan of k is a run such that each transition $(p_i, C_i[T_i^{\text{out}}]) \longrightarrow \big(p_{i+1}, C_i\big[T_{i+1}^{\text{in}}\big]\big)$ has the property that all nodes n in the domain of $C_i[T_i^{\text{out}}]$ but only in the domain of $C_i[x]$ if the label is x have an age of at most k. For example, the transitions in Figure 1 require a lifespan ≥ 1 since the oldest node that is rewritten by the transitions has age 1.

Definition 4.1 (Senescent Ground Tree Rewrite Systems)**.** *A* senescent ground tree rewrite system *with* lifespan k *is an sGTRS* $G = (\mathcal{P}, \Sigma, \mathcal{R})$ *where runs are* lifespan restricted *with a lifespan of* k*.*



(a) A transition changing the control state.



(b) A transition that does not change the control state.

Figure 1: Transitions of a senescent GTRS.

We will study the control state reachability problem and the regular reachability problem for senescent GTRS. We will show in Theorem 6.1 that the control state reachability problem is $\mathbf{F}_\omega$-complete, and in Theorem 5.3 that the regular reachability is undecidable.

Definition 4.2 (Control State Reachability Problem)**.** *Given a senescent GTRS G with lifespan k, initial configuration (p_{src}, T_{src}), and target control state p_{snk}, the* control state reachability problem *is to decide whether there exists a lifespan restricted run*

$$(p_{src}, T_{src}) \longrightarrow \cdots \longrightarrow (p_{snk}, T)$$

for some T.

Definition 4.3 (Regular Reachability Problem)**.** *Given a senescent GTRS G with lifespan k, initial configuration (p_{src}, T_{src}), target control state p_{snk}, and tree automaton $\mathcal{T}$, the* regular reachability problem *is to decide whether there exists a lifespan restricted run*

$$(p_{src}, T_{src}) \longrightarrow \cdots \longrightarrow (p_{snk}, T)$$

for some $T \in \mathcal{L}(\mathcal{T})$.

One might expect that decidability of control state reachability would imply decidability of regular reachability: one could simply encode the tree automaton into the senescent GTRS. However, it is not possible to enforce conditions on the final tree (e.g. that all leaf nodes are labelled by initial states of the tree automaton) when only the final control state can be specified.

```
shared int nextID = 0;

critical int getID() {
    return nextID++;
}

void run(int myID, int siblingID) {
    switch * {
        case:
            int id1 = getID();
            int id2 = getID();
            spawn(id1, run(id1, id2));
            spawn(id2, run(id2, id1));
            break;
        case:
            if (siblingID >= 0)
                sendMessage(siblingID);
        case:
            if (siblingID >= 0)
                receiveMessage(siblingID);
    }
}

void main() {
    int id = getID();
    spawn(id, run(id, -1));
}
```

Figure 2: A simple program with thread creation.

4.2 Example

4.2.1 A Simple Concurrent Program

We present a simple example of how a senescent GTRS may be used to model a simple concurrent system. Consider the toy program in Figure 2. This simple (outline) program has dynamic thread creation.

Beginning from the main function, the program creates a thread using a spawn function that takes as an argument the ID to assign to the new thread, and the function to run in the new thread. IDs are obtained using a getID function. This function is declared "critical" to ensure that no two threads execute the function simultaneously. It uses a shared variable to ensure that a fresh ID is returned after each call.

The run function describes the behaviour of each process. It takes as an argument its own ID and the ID of another process with which it may communicate. The switch statement represents a non-deterministic choice between each of the cases: the process creates two new child processes who may communicate with each other; or the process communicates with its sibling.

In order to ensure reliable communication, a relevant property may be that each thread has

a unique ID. With a standard C-like semantics, the program in Figure 2 does not satisfy this property: when the value of nextID surpasses the maximum value that can be stored in an int, it will loop back around to 0. Thus, two processes may share the same ID. This is a bug in the program.

4.2.2 Modelling the Program as a Senescent GTRS

We may model the above program as a senescent GTRS. For simplicity, we will inline all function calls (we show in Section 5.1 how to model threads with stacks, and hence function calls). We will also abstract away the sendMessage and receiveMessage functions into silent actions (since we are only interested in whether two threads share the same ID).

We define the senescent GTRS $G = (\mathcal{P}, \Sigma, \mathcal{R})$. Let m be the maximum value of an int. The set of control states will be

$$\begin{array}{rcl} \mathcal{P} & = & \{0, \ldots, m\} \cup \\ & & \{(i, (i_1, j_1)) \mid 0 \leq i, i_1 \leq m \wedge -1 \leq j_1 \leq m\} \cup \\ & & \{has(0), \ldots, has(m)\} \cup \\ & & \{p_E\} \ . \end{array}$$

The control states in $\{0, \ldots, m\}$ will track the value of the nextID variable. States of the form $(i, (i_1, j_1))$ indicate that nextID is i and a new thread with ID i_1 and sibling ID j_1 should be started. Control states of the form $has(i)$ will be used to detect whether two threads exist with the same ID. The final control state p_E indicates that an ID error has been detected.

The alphabet will be the set

$$\begin{array}{rcl} \Sigma & = & \{(i_1, j_1) \mid -1 \leq i_1, j_1 \leq m\} \cup \\ & & \{(i_1, j_1, i_2, j_2)_1, (i_1, j_1, i_2, j_2)_2 \mid 0 \leq i_1, i_2, j_1 \leq m \wedge -1 \leq j_2 \leq m\} \cup \\ & & \{*, \bullet, \jmath\} \ . \end{array}$$

The senescent GTRS will maintain a tree with a single leaf node per thread. Each leaf corresponding to a thread will be of the form (i_1, j_1) indicating the values of myID and siblingID respectively. The labels $(i_1, j_1, i_2, j_2)_1$ and $(i_1, j_1, i_2, j_2)_2$ will be used respectively for the first and second calls to spawn: i_2 and j_2 will store the values returned by getID before the spawn actions are called.

Note, if we wanted to model function calls using a pushdown system, we would use a branch representing the call stack rather than simple leaves (as in Section 5.1).

The label $\bullet$ will label internal nodes of the tree, while $*$ will label a unique leaf node from which the tree may be expanded to accommodate new threads. Finally, $\jmath$ indicates a sleeping thread.

The rules $\mathcal{R}$ of G will be of several kinds: internal thread actions, spawn actions, and error detection actions.

- The internal actions (including the abstracted send and receive actions) will simply be rules of the form $\left(p, \mathcal{T}_{(i_1,j_1)}\right) \longrightarrow \left(p, \mathcal{T}_{(i_1,j_1)}\right)$ for all $p \in \mathcal{P}$ and $-1 \leq i_1, j_1, \leq m$. Note that applying these rules does not change the structure of given configuration, but it does reset the age of each leaf node to zero, preventing the node from fossilising. Likewise, we can also have rules $(p, \mathcal{T}_a) \longrightarrow (p, \mathcal{T}_a)$ for each $p \in \mathcal{P}$ and each $a \in \Sigma \setminus \{\bullet\}$.
- The spawn actions will be modelled in several steps.
  - First we get the value of id1 by $\left(i, \mathcal{T}_{(i_1,j_1)}\right) \longrightarrow \left(i', \mathcal{T}_{(i_1,j_1,i,-1)_1}\right)$ where $i' = (i + 1) \bmod m$.

– Next we get the value of id2 using $\big(i, \mathcal{T}_{(i_1,j_1,i_2,-1)_1}\big) \longrightarrow \big(i', \mathcal{T}_{(i_1,j_1,i_2,i)_1}\big)$ (where $i' = (i+1) \bmod m$).

– We then perform the first spawn action using two rules. First we use

$$\big(i, \mathcal{T}_{(i_1,j_1,i_2,j_2)_1}\big) \longrightarrow \big((i,(i_2,j_2)), \mathcal{T}_{(i_1,j_1,i_2,j_2)_2}\big)$$

to spawn the first thread, and then

$$\big(i, \mathcal{T}_{(i_1,j_1,i_2,j_2)_2}\big) \longrightarrow \big((i,(j_2,i_2)), \mathcal{T}_{(i_1,j_1)}\big)$$

to spawn the second.

Note that each spawn request changes the control state to $(i,(i_1,j_1))$. The actual creation of each new thread is via rules of the form

$$((i,(i_1,j_1)), \mathcal{T}_*) \longrightarrow \big(i, \mathcal{T}_{\bullet((i_1,j_1),*)}\big) \ .$$

- Finally, we will detect multiple IDs via two sets of rules. In the first a thread can question whether any other thread has the same ID. That is, we have $\big(i, \mathcal{T}_{(i_1,j_1)}\big) \longrightarrow \big(has(i_1), \mathcal{T}_{\mathbb{D}}\big)$. That is, the node requests if any thread has the same ID, and then goes to sleep (so that it does not answer its own request). The request can be answered with a rule $\big(has(i), \mathcal{T}_{(i,j_1)}\big) \longrightarrow \big(p_E, \mathcal{T}_{(i,j_1)}\big)$, taking the GTRS to control state p_E, indicating that an error has occurred.

The above defined system will always have a run to p_E from the initial configuration $\big(1, \mathcal{T}_{\bullet((0,-1,,)*)}\big)$ which represents the system state of the initial spawn action. This holds even with a lifespan of 1: the first process simply performs spawn actions until the nextID reaches m and wraps around to 0; then by starting another two new processes and moving to $has(0)$ the error can be detected.

5 Modelling Power of Senescent GTRSs

We show in this section that senescent GTRSs at least capture scope-bounded multi-stack pushdown systems. Essentially just by encoding each stack as a different branch of the tree. We also show that we can encode coverability and reachability of a reset Petri net. The former can be reduced to control state reachability, whereas the latter is reducible to regular reachability. Thus, regular reachability is undecidable.

5.1 Scope-Bounded Pushdown Systems

A senescent GTRS may quite naturally model a scope-bounded multi-stack pushdown system. Scope-bounded multi-pushdown systems were first introduced by La Torre and Napoli [24], where they were shown to have a PSPACE-complete control state reachability problem. This is in contrast to the control state reachability problem for senescent GTRS which we will show to be $\mathbf{F}_\omega$-complete. Hence, senescent GTRS represent a significant increase in modelling power. We first define multi-stack and scope-bounded pushdown systems, before comparing them with senescent GTRS.

5.1.1 Model Definition

A multi-stack pushdown system consists of, at any one moment, a control state and a fixed number z of stacks over an alphabet Σ. Runs of a scope-bounded multi-stack pushdown system are organised into *rounds*, where each round consists of z *phases* and during the ith phase, stack operations may only occur on the ith stack. This can be thought of as several threads running on a round robin scheduler. The *scope-bound* k is a restriction on which characters may be removed from a stack. That is, a character may only be removed if it was pushed within the previous k rounds.

Definition 5.1 (Multi-Stack Pushdown Systems)**.** *A* multi-stack pushdown system *is a tuple* $\mathbb{P} = (\mathcal{P}, \Sigma, \mathcal{R}_1, \ldots, \mathcal{R}_z)$ *where* $\mathcal{P}$ *is a finite set of control states,* Σ *is a finite set of stack characters, and for each* $1 \leq i \leq z$ *we have* $\mathcal{R}_i = \mathcal{R}_i^{push} \cup \mathcal{R}_i^{int} \cup \mathcal{R}_i^{pop}$ *is a set of rules where* $\mathcal{R}_i^{push} \subseteq \mathcal{P} \times \mathcal{P} \times \Sigma$ *is a set of push rules,* $\mathcal{R}_i^{int} \subseteq \mathcal{P} \times \mathcal{P}$ *is a set of internal rules, and* $\mathcal{R}_i^{pop} \subseteq \mathcal{P} \times \Sigma \times \mathcal{P}$ *is a set of pop rules.*

A *configuration* of a multi-stack pushdown system is a tuple $(p, w_1, \ldots, w_z)$ where $p \in \mathcal{P}$ and for all $1 \leq i \leq z$ we have $w_1 \in \Sigma^*$. We have a transition on stack i, written

$$(p, w_1, \ldots, w_z) \longrightarrow_i (p', w_1, \ldots, w_{i-1}, w_i', w_{i+1}, \ldots, w_z)$$

whenever we have

1. a rule $(p, p', a) \in \mathcal{R}_i^{push}$ and $w_i' = aw_i$, or
2. a rule $(p, p') \in \mathcal{R}_i^{int}$ and $w_i' = w_i$, or
3. a rule $(p, a, p') \in \mathcal{R}_i^{pop}$ and $aw_i' = w_i$.

We write $\longrightarrow_i^*$ for the transitive closure of $\longrightarrow_i$ and $\longrightarrow$ for the union of all $\longrightarrow_i$. Take a *run*

$$\sigma_0 \longrightarrow \cdots \longrightarrow \sigma_h$$

of a multi-stack pushdown system and suppose that the ith configuration σ_i is the configuration $(p_i, w_1, \ldots, w_{j-1}, aw_j, w_{j+1}, \ldots, w_z)$. We say that a was pushed at configuration $\sigma_{i'}$ whenever $i' \leq i$ is the largest index such that we have

$$\begin{aligned} \sigma_{i'-1} = \quad & (p_{i'-1}, w_1', \ldots, w_{j-1}', w_j, w_{j+1}', \ldots, w_z') \\ & \longrightarrow_j \\ & (p_{i'}, w_1', \ldots, w_{j-1}', aw_j, w_{j+1}', \ldots, w_z') \quad = \sigma_{i'} \ . \end{aligned}$$

Note that, by convention, a was pushed at configuration σ_0 if no such i' exists.

A *round* $\sigma_0 \longrightarrow_R \sigma_z$ of a multi-stack pushdown system is a sequence

$$\sigma_0 \longrightarrow_1^* \sigma_1 \longrightarrow_2^* \cdots \longrightarrow_z^* \sigma_z$$

where each σ_i is a configuration.

Finally, a *k-scope-bounded* run of a multi-stack pushdown system is a sequence of rounds

$$\sigma_1 \longrightarrow_R \cdots \longrightarrow_R \sigma_h$$

for some h such that for all j and all pop transitions

$$(p, w_1, \ldots, w_{i-1}, aw_i, w_{i+1}, \ldots, w_z) \longrightarrow_i (p', w_1, \ldots, w_{i-1}, w_i, w_{i+1}, \ldots, w_z)$$

occurring during the round $\sigma_j \longrightarrow_R \sigma_{j+1}$ we have that a was pushed during the round $\sigma_{j'} \longrightarrow_R \sigma_{j'+1}$ where $(j - k) \leq j' \leq j$.

Definition 5.2 (Scope-bounded Multi-Stack Pushdown System)**.** *We define a* k-scope-bounded multi-stack pushdown system *to be a multi-stack pushdown system together with a scope-bound* k*.*

We thus define the control state reachability problem for scope-bounded multi-stack pushdown systems.

Definition 5.3 (Control State Reachability)**.** *Given a scope-bounded multi-stack pushdown system* $\mathbb{P}$ *with scope-bound* k *and* z *stacks, control states* p_{src} *and* p_{snk}*, and initial stack character* $\bot$*, the* control state reachability problem *is to decide whether there exists a* k*-scope-bounded run*

$$(p_{src}, \bot, \ldots, \bot) \longrightarrow \cdots \longrightarrow (p_{snk}, w_1, \ldots, w_z)$$

for some stacks $w_1, \ldots, w_z$*.*

5.1.2 Reduction to Senescent GTRS

We show that the control state reachability problem for scope-bounded multi-stack pushdown systems can be simply reduced to the control state reachability problem for senescent GTRS. The reduction is a straightforward extension of the standard method for encoding a pushdown system with an sGTRS with a single control state, which was generalised to *context-bounded* multi-stack pushdown systems by Lin [27].

Without loss of generality, we will assume a stack symbol $\bot$ that is the bottom-of-stack symbol. It is neither pushed onto, nor popped from the stack. It will also be the initial stack character in the control state reachability problem.

Furthermore, by abuse of notation, for a stack $w = a_1 \ldots a_m$ we write $w(T)$ for the tree $a_m(\cdots a_1(T))$.

A single-stack pushdown system can be modelled as follows. A configuration (p, w) is encoded as a tree containing a single path. Consider the tree $w(p)$. Since the rules of the pushdown system only depend on and change the control state and the top of the stack, they can be encoded as tree rewriting operations. For example, the push rule (p, p', a) can be modelled by matching the subtree p and replacing it with $a(p')$.

To extend this to multi-stack pushdown systems with z stacks, we maintain a tree whose root is a node with z children, where each child encodes a stack. However, in this case, the control state must be stored in the control state of the sGTRS since the tree rewriting rules can only rewrite sub-trees.

Fortunately, the structure of a scope-bounded run allows us to choose an encoding that is more economical with the use of the senescent GTRS's control state. We will use the pushdown system's control state as a kind of "token" to indicate which stack is currently active in the round. That is, it will appear as a leaf of the branch containing the currently active stack. To move to the next stack (i.e. use $\longrightarrow_{i+1}$ instead of $\longrightarrow_i$) the control state of the senescent GTRS will be used to transfer the pushdown system's control state to the next branch. Thus, a k-scope-bounded multi-stack pushdown system will be modelled by a senescent GTRS with a lifespan of $k \cdot z$. This is natural since a round of a scope bounded pushdown system contains z communications, and hence k rounds contain $k \cdot z$ communications.

Definition 5.4 ($G_{\mathbb{P}}$)**.** *Given a* k*-scope-bounded multi-stack pushdown system* $\mathbb{P}$ *as the tuple*

$(\mathcal{P}, \Sigma, \mathcal{R}_1, \ldots, \mathcal{R}_z)$ we define the senescent GTRS $G_{\mathbb{P}} = (\mathcal{P}_S, \Sigma_S, \mathcal{R}_S)$ with lifespan $k \cdot z$ where

$$\begin{aligned} \mathcal{P}_S &= (\mathcal{P} \times \{1, \ldots, z\}) \cup \{p_{src}, p_{snk}\} \\ \Sigma_S &= \Sigma \cup \mathcal{P} \cup \{\bullet, \square_1, \ldots, \square_z\} \\ \mathcal{R}_S &= \mathcal{R}_{src/snk} \cup \mathcal{R}_{PDS} \cup \mathcal{R}_{Switch} \end{aligned}$$

and $\mathcal{R}_{src/snk}$ is the set

$$\{(p_{src}, \mathcal{T}_{\square_z}) \longrightarrow ((p_{src}, 1), \mathcal{T}_{\square_z}), ((p_{snk}, 1), \mathcal{T}_{p_{snk}}) \longrightarrow (p_{snk}, \mathcal{T}_{p_{snk}})\}$$

and $\mathcal{R}_{PDS}$ is the set

$$\begin{Bmatrix} ((p,i), \mathcal{T}_{p_1}) \longrightarrow \big((p,i), \mathcal{T}_{a(p_2)}\big) \;\Big|\; \begin{array}{c} 1 \le i \le z \wedge \\ p \in \mathcal{P} \wedge \\ (p_1, p_2, a) \in \mathcal{R}_i \end{array} \end{Bmatrix} \cup$$
$$\begin{Bmatrix} ((p,i), \mathcal{T}_{p_1}) \longrightarrow ((p,i), \mathcal{T}_{p_2}) \;\Big|\; \begin{array}{c} 1 \le i \le z \wedge \\ p \in \mathcal{P} \wedge \\ (p_1, p_2) \in \mathcal{R}_i^{int} \end{array} \end{Bmatrix} \cup$$
$$\begin{Bmatrix} \big((p,i), \mathcal{T}_{a(p_1)}\big) \longrightarrow ((p,i), \mathcal{T}_{p_2}) \;\Big|\; \begin{array}{c} 1 \le i \le z \wedge \\ p \in \mathcal{P} \wedge \\ (p_1, a, p_2) \in \mathcal{R}_i^{pop} \end{array} \end{Bmatrix}$$

and finally $\mathcal{R}_{Switch}$ is the set

$$\{((p,i), \mathcal{T}_{p'}) \longrightarrow ((p', (i \bmod z) + 1), \mathcal{T}_{\square_i}) \mid 1 \le i \le z\} \cup$$
$$\{((p,i), \mathcal{T}_{\square_i}) \longrightarrow ((p,i), \mathcal{T}_p) \mid 1 \le i \le z\} \ .$$

Theorem 5.1 (Scope-Bounded to Senescent GTRS)**.** *The control state reachability problem for scope-bounded multi-stack pushdown systems can be reduced to the control state reachability problem for senescent GTRS.*

Proof. Given a k-scope-bounded multi-stack pushdown system $\mathbb{P} = (\mathcal{P}, \Sigma, \mathcal{R}_1, \ldots, \mathcal{R}_z)$ we obtain the senescent GTRS $G_{\mathbb{P}}$ with lifespan $k \cdot z$ as in Definition 5.4 ($G_{\mathbb{P}}$). It is almost direct to obtain from a run

$$(p_{src}, \bot, \ldots, \bot) \longrightarrow \cdots \longrightarrow (p_{snk}, w_1, \ldots, w_z)$$

of $\mathbb{P}$ a run

$$(p_{src}, \bullet(\bot(\square_1), \ldots, \bot(\square_z))) \longrightarrow \cdots \longrightarrow (p_{snk}, \bullet(T_1, \ldots, T_z))$$

of $G_{\mathbb{P}}$ and vice versa for some $w_1, \ldots, w_z$ and $T_1, \ldots, T_z$.

To go from $\mathbb{P}$ to $G_{\mathbb{P}}$ we divide the run into rounds and the rounds into sub-runs

$$(p, w_1, \ldots, w_{i-1}, w_i, w_{i+1}, \ldots, w_z) \longrightarrow_i^* (p', w_1, \ldots, w_{i-1}, w_i', w_{i+1}, \ldots, w_z) \ .$$

We can obtain by straightforward induction (using rules from $\mathcal{R}_{PDS}$ and the fact we never remove $\bot$ from a stack) a run

$$\begin{array}{c} ((p,i), \bullet(w_1(\square_1), \ldots, w_{i-1}(\square_{i-1}), w_i(p), w_{i+1}(\square_{i+1}), \ldots, w_z(\square_z))) \\ \longrightarrow^* \\ ((p,i), \bullet(w_1(\square_1), \ldots, w_{i-1}(\square_{i-1}), w_i'(p'), w_{i+1}(\square_{i+1}), \ldots, w_z(\square_z))) \end{array}$$

of $G_{\mathbb{P}}$. Note that this holds true even for empty runs. Then by topping and tailing with transitions from $\mathcal{R}_{Switch}$ we obtain.

$$\begin{gathered}((p,i),\bullet(w_1(\square_1),\ldots,w_{i-1}(\square_{i-1}),w_i(\square_i),w_{i+1}(\square_{i+1}),\ldots,w_z(\square_z)))\\ \longrightarrow^*\\ ((p',i'),\bullet(w_1(\square_1),\ldots,w_{i-1}(\square_{i-1}),w'_i(\square_i),w_{i+1}(\square_{i+1}),\ldots,w_z(\square_z)))\end{gathered}$$

where $i' = (i \bmod z) + 1$.

Thus, combining these runs, from a round

$$(p,w_1,\ldots,w_z) \longrightarrow_1^* \cdots \longrightarrow_z^* (p',w'_1,\ldots,w'_z)$$

of $\mathbb{P}$ we obtain a run

$$((p,1),\bullet(w_1(\square_1),\ldots,w_z(\square_z))) \longrightarrow^* ((p',1),\bullet(w'_1(\square_1),\ldots,w'_z(\square_z)))$$

of $G_{\mathbb{P}}$.

To complete the direction, we must now combine rounds of the run of $\mathbb{P}$ into a run of $G_{\mathbb{P}}$. To do this, we simply concatenate the runs obtained for each round. We have to be careful that in doing so we respect the lifespan $k \cdot z$ of $G_{\mathbb{P}}$. Indeed, this is a simple consequence of the scope-bound of $\mathbb{P}$: since we never remove a character that was pushed k rounds earlier, we know that the top character must have been pushed at most k rounds earlier, and thus there are fewer than $k \cdot z$ control state changes of $G_{\mathbb{P}}$ since the birth date of the corresponding node in the tree (note also that the leaf node is rewritten every z control state changes, and hence does not become fossilised). We thus obtain an almost complete run of $G_{\mathbb{P}}$: all that remains is to append and concatenate transitions from $\mathcal{R}_{src/snk}$, resulting in a run of the required form.

In the opposite direction one need only observe that all runs of $G_{\mathbb{P}}$ must be of the form constructed during the proof of the direction above. Hence, applying the above reasoning in reverse obtains a run of $\mathbb{P}$ as required. □

5.2 Reset Petri-Nets

We show that the coverability and reachability problems for reset Petri-nets can be reduced to the control state and regular reachability problems for senescent GTRS respectively. The idea is that the control state of the reset Petri-net can be directly encoded by the control state of the senescent GTRS. To keep track of the marking for each counter x, we maintain a tree with $\pi(x)$ leaf nodes labelled x. Decrementing a counter is then a case of rewriting a leaf node x to the empty tree, while incrementing the counter requires adding a new leaf node. To avoid leaf nodes becoming fossilised, we allow all leaf nodes to rewrite to themselves in (almost) every control state. We can then reset a counter by forcing the GTRS to change control states k times without allowing the counter to refresh; thus, all x nodes become fossilised and the counter is effectively set to zero.

5.2.1 Coverability

We will begin with coverability. Without loss of generality, we assume that we aim to cover the zero marking. Moreover, we assume that in each rule $q \xrightarrow[\widetilde{o}]{} q'$ we have at most a single counter operation in $\widetilde{o}$. In the following definition, we use $*$ to label an open node which may spawn new counter-labelled nodes, $\bullet$ to label internal nodes of the tree, and $\eth$ to label counter nodes that have been disactivated by a decrement operation.

Definition 5.5 ($G_{\mathcal{N}}$)**.** *Given a reset Petri-net $\mathcal{N} = (\mathcal{Q}, X, \Delta)$ we define the senescent GTRS $G_{\mathcal{N}} = (\mathcal{P}, \Sigma, \mathcal{R})$ with lifespan 1 where*

$$
\begin{aligned}
\mathcal{P} =& \mathcal{Q} \cup \left\{ \times_q^x \mid x \in X \wedge q \in \mathcal{Q} \right\} \\
\Sigma =& X \cup \{*, \bullet, \mathbb{D}\} \\
\mathcal{R} =& \left\{ (p, \mathcal{T}_x) \longrightarrow (p, \mathcal{T}_x) \;\middle|\; x \in X \wedge p \in \mathcal{P} \wedge \forall q \in \mathcal{Q}. p \neq \times_q^x \right\} \cup \\
& \{ (p, \mathcal{T}_*) \longrightarrow (p, \mathcal{T}_*) \mid p \in \mathcal{P} \} \cup \\
& \left\{ (q, \mathcal{T}_*) \longrightarrow \left(q', \mathcal{T}_{\bullet(x,*)}\right) \;\middle|\; q \xrightarrow[\{incr(x)\}]{} q' \in \Delta \right\} \cup \\
& \left\{ (q, \mathcal{T}_x) \longrightarrow \left(q', \mathcal{T}_{\mathbb{D}}\right) \;\middle|\; q \xrightarrow[\{decr(x)\}]{} q' \in \Delta \right\} \cup \\
& \left\{ \begin{array}{l} (q, \mathcal{T}_*) \longrightarrow \left(\times_{q'}^x, \mathcal{T}_*\right), \\ \left(\times_{q'}^x, \mathcal{T}_*\right) \longrightarrow (q', \mathcal{T}_*) \end{array} \;\middle|\; q \xrightarrow[\{res(x)\}]{} q' \in \Delta \right\} \cup \\
& \left\{ (q, \mathcal{T}_*) \longrightarrow (q', \mathcal{T}_*) \;\middle|\; q \underset{\emptyset}{\to} q' \in \Delta \right\}
\end{aligned}
$$

Theorem 5.2 (Coverability to Control State Reachability)**.** *The coverability problem for reset Petri-nets can be reduced to the control state reachability problem for senescent GTRS.*

Proof. Given a reset Petri-net $\mathcal{N} = (\mathcal{Q}, X, \Delta)$ we obtain the senescent GTRS $G_{\mathcal{N}} = (\mathcal{P}, \Sigma, \mathcal{R})$ with lifespan 1 from Definition 5.5 ($G_{\mathcal{N}}$). We show there exists a run

$$(q_1, \pi_1) \longrightarrow \cdots \longrightarrow (q_h, \pi_h)$$

of $\mathcal{N}$ where $\pi_1 = \pi_0$ iff there is a run

$$(q_1, T_1) \longrightarrow^* (q_2, T_2) \longrightarrow^* \cdots \longrightarrow^* (q_h, T_h)$$

of $G_{\mathcal{N}}$ where $T_1 = *$. This implies our theorem.

First consider the direction from $\mathcal{N}$ to $G_{\mathcal{N}}$. We induct from $i = 1$ to $i = h$. We maintain the induction invariant that T_i has exactly $\pi_i(x)$ leaf nodes of age 0 labelled by x for each counter $x \in X$. Furthermore, there is exactly one leaf node labelled $*$, and this node has age 1. In the base case the invariant is immediate.

Now assume the invariant for i. Consider the transition

$$(q_i, \pi_i) \underset{\widetilde{o}}{\to} (q_{i+1}, \pi_{i+1})$$

of the run of $\mathcal{N}$. We show the existence of a run

$$(q_i, T_i) \longrightarrow^* (q_{i+1}, T_{i+1})$$

satisfying the invariant. We perform a case split on $\widetilde{o}$. In the following, when we say "refresh the leaf nodes" we mean that we execute for each counter x and for each leaf node (with age < 2) labelled by x a rule $(q_{i+1}, \mathcal{T}_x) \longrightarrow (q_{i+1}, \mathcal{T}_x)$ to set the age of each leaf node back to 0, and finally we fire $(q_{i+1}, \mathcal{T}_*) \longrightarrow (q_{i+1}, \mathcal{T}_*)$ to set the age of the node labelled $*$ to 0.

1. When $\widetilde{o} = \emptyset$ we can fire the rule $(q_i, \mathcal{T}_*) \longrightarrow (q_{i+1}, \mathcal{T}_*)$ and since $\pi_i = \pi_{i+1}$ we simply refresh the leaf nodes to obtain the induction invariant.

2. When $\widetilde{o} = \{incr(x)\}$ we fire the rule $(q_i, \mathcal{T}_*) \longrightarrow \left(q_{i+1}, \mathcal{T}_{\bullet(x,*)}\right)$ to obtain T_{i+1} with the correct number of leaf nodes concordant with π_{i+1} and then refresh the leaf nodes to obtain the invariant.

3. When $\widetilde{o} = \{decr(x)\}$ we know from the invariant that since $\pi_i(x) > 0$ that there is a leaf labelled x of age 0. We fire the rule $(q_i, \mathcal{T}_x) \longrightarrow \left(q_{i+1}, \mathcal{T}_{\emptyset}\right)$ to obtain T_{i+1} with the correct number of leaf nodes concordant with π_{i+1} and then refresh the leaf nodes to obtain the invariant.

4. When $\widetilde{o} = \{res(x)\}$ we fire the rule $(q_i, \mathcal{T}_*) \longrightarrow \left(\times^x_{q_{i+1}}, \mathcal{T}_*\right)$ and then refresh all leaf nodes except those labelled x, which cannot be reset. Thus all leaf nodes have age 0 except those labelled x which have age 1. Then we fire $\left(\times^x_{q_{i+1}}, \mathcal{T}_*\right) \longrightarrow (q_{i+1}, \mathcal{T}_*)$. Note that all leaf nodes labelled x now have age 2 and are fossilised. Then we refresh all leaf nodes (that are young enough) to obtain T_{i+1} concordant with π_{i+1} and obtain the invariant.

Thus, by induction, we obtain a run as required.

In the other direction, we take a run

$$(q_1, T_1) \longrightarrow^* \cdots \longrightarrow^* (q_h, T_h)$$

where $T_1 = *$ and for all configurations between (q_i, T_i) and (q_{i+1}, T_{i+1}) (if there are any) there is some x such that such that the control state of the configurations is of the form $\times^x_{q_{i+1}}$. That all runs are of this form follows easily from the definition of $G_{\mathcal{N}}$. We build a run

$$(q_1, \pi_1) \longrightarrow \cdots \longrightarrow (q_h, \pi_h)$$

of $\mathcal{N}$ where $\pi_1 = \pi_0$. For technical convenience we assume that $\mathcal{N}$ can perform "no-op" transitions that change neither the control state nor the marking. One can simply remove these transitions from the final run to obtain a run of $\mathcal{N}$.

We induct from $i = 1$ to $i = h$. We maintain the induction invariant $\pi(x)$ is greater than or equal to the number of leaf nodes of age ≤ 1 labelled by x for each counter $x \in X$. In the base case the invariant is immediate.

Now consider the first transition on the run

$$(q_i, T_i) \longrightarrow^* (q_{i+1}, T_{i+1}) \ .$$

There are several cases depending on the rule used by the transition.

1. When the rule is $(q_i, \mathcal{T}_x) \longrightarrow (q_{i+1}, \mathcal{T}_x)$ or $(q_i, \mathcal{T}_*) \longrightarrow (q_{i+1}, \mathcal{T}_*)$ we have $q_i = q_{i+1}$. Since this transition can only set the age of some leaf node of age ≤ 1 to 0 we extend the run of $\mathcal{N}$ with a no-op transition which maintains the invariant.

2. When the rule is $(q_i, \mathcal{T}_*) \longrightarrow \left(q_{i+1}, \mathcal{T}_{\bullet(x,*)}\right)$ we have a rule $q_i \xrightarrow[\{incr(x)\}]{} q_{i+1}$ by definition. We extend the run of $\mathcal{N}$ by firing this rule to obtain π_{i+1}. For all $x' \neq x$ we know that the number of leaf nodes labelled x' of age ≤ 1 can only be reduced, and hence $\pi_{i+1}(x')$ remains larger or equal. For x the number of leaf nodes may increase by at most 1, but since we fired $incr(x)$ the invariant is maintained.

3. When the rule is $(q_i, \mathcal{T}_x) \longrightarrow \left(q_{i+1}, \mathcal{T}_{\emptyset}\right)$ we have a rule $q_i \xrightarrow[\{decr(x)\}]{} q_{i+1}$ by definition. We extend the run of $\mathcal{N}$ by firing this rule to obtain π_{i+1}. This is possible since we know

there is at least one leaf node of age ≤ 1 labelled x and hence $\pi_i(x) > 0$ by induction. Then for all $x' \neq x$ we know that the number of leaf nodes labelled x' of age ≤ 1 can only be reduced, and hence $\pi_{i+1}(x')$ remains larger or equal. For x the number of leaf nodes decreases by at least 1, and hence the invariant is maintained.

4. When the rule is $(q_i, \mathcal{T}_*) \longrightarrow \left(\times^x_{q_{i+1}}, \mathcal{T}_*\right)$ we know by definition that the transition reaching (q_{i+1}, T_{i+1}) is via the rule $\left(\times^x_{q_{i+1}}, \mathcal{T}_*\right) \longrightarrow (q_{i+1}, \mathcal{T}_*)$ and all transitions in between are of the form $\left(\times^x_{q_{i+1}}, \mathcal{T}_{x'}\right) \longrightarrow \left(\times^x_{q_{i+1}}, \mathcal{T}_{x'}\right)$ where $x' \neq x$ or $\left(\times^x_{q_{i+1}}, \mathcal{T}_*\right) \longrightarrow \left(\times^x_{q_{i+1}}, \mathcal{T}_*\right)$. Hence the number of leaf nodes of age ≤ 1 in T_{i+1} will be zero with the label x and less than or equal to the number in T_i when the label is some other x'. Thus, by firing the transition $q_i \xrightarrow[\{res(x)\}]{} q_{i+1}$ to obtain (q_{i+1}, π_{i+1}) we maintain the invariant.

5. When the rule is $(q_i, \mathcal{T}_*) \longrightarrow (q_{i+1}, \mathcal{T}_*)$ the number of leaf nodes with any label of age ≤ 1 can only decrease. By definition we have a rule $q_i \underset{\emptyset}{\rightarrow} q_{i+1}$ which we fire to obtain (q_{i+1}, π_{i+1}) and maintain the invariant.

Thus we are done. □

5.2.2 Reachability

Finally, using a slight extension of the reduction used for coverability, we can show that the reachability problem reduces to the regular reachability problem for GTRS.

The proof is by a minor extension of the coverability reduction. Naively, since the reduction uses leaf nodes to store the value of the counters, we could simply test reachability with respect to a tree automaton $\mathcal{T}$ that accepts trees where the number of leaf nodes labelled by each counter matches the target marking of the counter. However, this does not work since the reset actions are encoded by forcing leaf nodes to become fossilised. Hence, the number of leaf nodes labelled by a counter will not match the actual marking of the counter.

To overcome this problem we make two modifications to the encoding. First, when a counter is being reset, we give all leaves labelled by that counter the opportunity to rewrite themselves to $\mathfrak{d}$. Furthermore, when we have reached the target control state, we have the possibility to make a non-deterministic guess that the target marking has also been reached. At this point we let all active leaf nodes labelled by a counter x to rewrite themselves to be labelled by $\overline{x}$. We then define the target tree automaton $\mathcal{T}$ to accept trees where the number of leaves labelled $\overline{x}$ matches the target marking value of the counter, and, moreover, there are no leaves labelled by a counter x. The second condition ensures that no node labelled x allowed itself to become fossilised while labelled x (in particular, during a reset, all nodes rewrote themselves to $\mathfrak{d}$). Similarly, after guessing that the target configuration had been reached, all nodes labelled x rewrote themselves to $\overline{x}$ thus ensuring that the tree accurately represents the true counter values.

Definition 5.6 ($G'_{\mathcal{N}}$). *Given a reset Petri-net $\mathcal{N} = (\mathcal{Q}, X, \Delta)$, let $G_{\mathcal{N}} = (\mathcal{P}, \Sigma, \mathcal{R})$ be the*

senescent GTRS from Definition 5.5 ($G_{\mathcal{N}}$). We define $G'_{\mathcal{N}} = (\mathcal{P}', \Sigma', \mathcal{R}')$ with lifespan 1 where

$$\begin{array}{rcl} \mathcal{P}' & = & \mathcal{P} \cup \{p'_{snk}\} \\ \Sigma' & = & \Sigma \cup \{\overline{x} \mid x \in X\} \\ \mathcal{R}' & = & \mathcal{R} \cup \\ & & \{(p_{snk}, \mathcal{T}_*) \longrightarrow (p'_{snk}, \mathcal{T}_*)\} \cup \\ & & \{(p'_{snk}, \mathcal{T}_x) \longrightarrow (p'_{snk}, \mathcal{T}_{\overline{x}}) \mid x \in X\} \cup \\ & & \left\{\left(\times^x_q, \mathcal{T}_x\right) \longrightarrow \left(\times^x_q, \mathcal{T}_{\text{D}}\right) \mid x \in X \wedge q \in \mathcal{P}\right\} \end{array}$$

Theorem 5.3 (Reachability to Regular Reachability)**.** *The reachability problem for reset Petri-nets can be reduced to the regular reachability problem for senescent GTRS and is thus undecidable.*

Proof. Given a reset Petri-net $\mathcal{N} = (\mathcal{Q}, X, \Delta)$ we obtain the senescent GTRS $G'_{\mathcal{N}} = (\mathcal{P}, \Sigma, \mathcal{R})$ with lifespan 1 from Definition 5.6 ($G'_{\mathcal{N}}$). Let p_{src} be the initial control state, and (without loss of generality) let π_0 be the initial marking. Then, let p_{snk} be the target control state and π be the target marking.

First, define $\mathcal{T}$ to be the tree automaton accepting all trees of the following form:

1. all internal nodes are labelled $\bullet$, and
2. there is one leaf node labelled $*$, and
3. for each $x \in X$ there are exactly $\pi(x)$ leaf nodes labelled $\overline{x}$, and
4. all other leaf nodes are labelled D.

It is straightforward to construct such a $\mathcal{T}$.

We first show that the existence of a run

$$(p_{src}, \pi_0) \longrightarrow^* (p_{snk}, \pi)$$

of $\mathcal{N}$ implies there is a run

$$(p_{src}, T_1) \longrightarrow^* (p'_{snk}, T_2)$$

of $G_{\mathcal{N}}$ where $T_1 = *$ and $T_2 \in \mathcal{L}(\mathcal{T})$.

We can use almost the same proof as the same direction in the proof of Theorem 5.2 (Coverability to Control State Reachability), with a minor modification to the induction hypothesis. That is, instead of maintaining for each x that there are exactly $\pi_i(x)$ leaf nodes *of age* 0 labelled by x, we maintain the stronger property that there are exactly $\pi_i(x)$ leaf nodes labelled by x. To do so, we need only update the handling of the reset transition. That is, when handling $res(x)$ we fire, for each leaf node labelled x, the rule $\left(\times^x_q, \mathcal{T}_x\right) \longrightarrow \left(\times^x_q, \mathcal{T}_{\text{D}}\right)$. Thus, there are no leaf nodes labelled x.

We therefore obtain a run to (p_{snk}, T) such that T has all internal nodes labelled by $\bullet$, one node labelled by $*$, exactly $\pi(x)$ leaf nodes labelled x for each x, and all other leaf nodes labelled D. To complete the direction, we fire the rule $(p_{snk}, \mathcal{T}_*) \longrightarrow (p'_{snk}, \mathcal{T}_*)$ followed by $(p'_{snk}, \mathcal{T}_x) \longrightarrow (p'_{snk}, \mathcal{T}_{\overline{x}})$ for each leaf node labelled by some x. Whence, we reach the configuration (p'_{snk}, T_2) with $T_2 \in \mathcal{L}(\mathcal{T})$ as required.

The other direction is also similar to the coverability case. We take a run

$$(q_1, T_1) \longrightarrow^* \cdots \longrightarrow^* (q_h, T_h) \longrightarrow (p'_{snk}, T_h) \longrightarrow^* (p'_{snk}, T)$$

where $q_1 = p_{src}$, $T_1 = *$, $q_h = p_{snk}$, $T \in \mathcal{L}(\mathcal{T})$, and for all configurations between (q_i, T_i) and (q_{i+1}, T_{i+1}) (if there are any) there is some x such that such that the control state of the configurations is of the form $\times^x_{q_{i+1}}$. We then build a run

$$(q_1, \pi_1) \longrightarrow \cdots \longrightarrow (q_h, \pi_h)$$

of $\mathcal{N}$ where $\pi_1 = \pi_0$. Again, we assume that $\mathcal{N}$ can perform "no-op" transitions. We induct from $i = 1$ to $i = h$. We maintain the induction invariant $\pi(x)$ equal to the number of leaf nodes labelled by x for each counter $x \in X$. In the base case the invariant is immediate.

Now consider the first transition on the run

$$(q_i, T_i) \longrightarrow^* (q_{i+1}, T_{i+1}) \ .$$

There are several cases depending on the rule used by the transition. In all cases, it is key to observe that no leaf node labelled by x can become fossilised in the run of $G'_{\mathcal{N}}$: if such a leaf were to become fossilised it would still be present in T and thus we could not have $T \in \mathcal{L}(\mathcal{T})$.

1. When the rule is $(q_i, \mathcal{T}_x) \longrightarrow (q_{i+1}, \mathcal{T}_x)$ or $(q_i, \mathcal{T}_*) \longrightarrow (q_{i+1}, \mathcal{T}_*)$ we have $q_i = q_{i+1}$. Since this transition can only set the age of some leaf node of age ≤ 1 to 0 we extend the run of $\mathcal{N}$ with a no-op transition which maintains the invariant.

2. When the rule is $(q_i, \mathcal{T}_*) \longrightarrow \left(q_{i+1}, \mathcal{T}_{\bullet(x,*)}\right)$ we have a rule $q_i \xrightarrow[\{incr(x)\}]{} q_{i+1}$ by definition. We extend the run of $\mathcal{N}$ by firing this rule to obtain π_{i+1} satisfying the invariant.

3. When the rule is $(q_i, \mathcal{T}_x) \longrightarrow \left(q_{i+1}, \mathcal{T}_{\mathbb{D}}\right)$ we have a rule $q_i \xrightarrow[\{decr(x)\}]{} q_{i+1}$ by definition. We extend the run of $\mathcal{N}$ by firing this rule to obtain π_{i+1}. This is possible since we know there is at least one leaf node x which cannot be fossilised (as remarked above) and hence $\pi_i(x) > 0$ by induction.

4. When the rule is $(q_i, \mathcal{T}_*) \longrightarrow \left(\times^x_{q_{i+1}}, \mathcal{T}_*\right)$ we know by definition that the transition reaching (q_{i+1}, T_{i+1}) is via the rule $\left(\times^x_{q_{i+1}}, \mathcal{T}_*\right) \longrightarrow (q_{i+1}, \mathcal{T}_*)$ and all transitions in between are of the form $\left(\times^x_{q_{i+1}}, \mathcal{T}_{x'}\right) \longrightarrow \left(\times^x_{q_{i+1}}, \mathcal{T}_{x'}\right)$ where $x' \neq x$, $\left(\times^x_{q_{i+1}}, \mathcal{T}_*\right) \longrightarrow \left(\times^x_{q_{i+1}}, \mathcal{T}_*\right)$, or $\left(\times^x_{q_{i+1}}, \mathcal{T}_x\right) \longrightarrow \left(\times^x_{q_{i+1}}, \mathcal{T}_{\mathbb{D}}\right)$. Indeed, we know that a rule of the first form must be fired for all leaf nodes labelled by some $x' \neq x$, and a rule of the third form must be fired for each leaf labelled x. If this were not the case then some leaf labelled by a counter would become fossilised, preventing T from being accepted by $\mathcal{T}$. Hence, the number of leaf nodes labelled x is zero and the number labelled by some other counter x' is the same as in T_i. Thus, we have the invariant.

5. When the rule is $(q_i, \mathcal{T}_*) \longrightarrow (q_{i+1}, \mathcal{T}_*)$ then by definition we have a rule $q_i \xrightarrow[\emptyset]{} q_{i+1}$ which we fire to obtain (q_{i+1}, π_{i+1}) and maintain the invariant.

Note that the rules $(p_{snk}, \mathcal{T}_*) \longrightarrow (p'_{snk}, \mathcal{T}_*)$ and $(p'_{snk}, \mathcal{T}_x) \longrightarrow (p'_{snk}, \mathcal{T}_{\overline{x}})$ can only be fired during the final stage of the run reaching (p'_{snk}, T). Moreover, observe that in the final phase, in order for T to be accepted, all leaf nodes labelled x must have been rewritten to $\overline{x}$ and hence their number must match π. Thus, in T_h the number of leaf nodes labelled by x must have matched π. Thus, upon reaching (p_h, π_h) we have a run of the Petri-net as required. □

6 Reachability Analysis of Senescent GTRSs

For the following section, fix a senescent GTRS $G = (\mathcal{P}, \Sigma, \mathcal{R})$ with lifespan k. Furthermore, fix an ordering $r_1, \ldots, r_\ell$ on the rules in $\mathcal{R}$. Thus, we will use each rule $r \in \mathcal{R}$ as an index (that is, we use r instead of i when $r = r_i$). Notice that ℓ denotes the number of rules in G.

Without loss of generality, we assume that all tree automata appearing in the rules $\mathcal{R}$ of G accept at least one tree (rules not satisfying this condition can be discarded since they cannot be applied).

We show, using ideas from [26], that the control state reachability problem for senescent GTRSs is decidable and is $\mathbf{F}_\omega$-complete.

Theorem 6.1 (Ackermann-Completeness of Reachability)**.** *It is the case that the control state reachability problem for senescent GTRS is $\mathbf{F}_\omega$-complete.*

Proof. $\mathbf{F}_\omega$-hardness follows from Theorem 5.2 (Coverability to Control State Reachability) and the $\mathbf{F}_\omega$-hardness of the coverability problem for reset Petri-nets [40, 41].

The upper bound is obtained in the following sections. In outline, given a senescent GTRS G we obtain from Definition 6.4 a reset Petri-net $\mathcal{N}_G$ triply-exponential in the size of G. From Lemma 6.1 we know that we can decide the control state reachability problem for G via a coverability problem over $\mathcal{N}_G$. Since $\mathbf{F}_\omega$ is closed under all primitive-recursive reductions, we have our upper bound as required. □

6.1 Independent Sub-Tree Interfaces

Our algorithm will non-deterministically construct a representation of a run of G witnessing the reachability property. A key idea is that during the guessed run, certain sub-trees may operate independently of one another.

That is, suppose we have a tree consisting of a root node n with a left sub-tree T_1 and a right sub-tree T_2. If it is the case that, during the run, the complete tree rooted at n is never matched by the LHS of a rewrite rule, then we may say that T_1 and T_2 develop independently: any rewrite rule applied during the run either matches a sub-tree of T_1 or a sub-tree of T_2, but never depends on both the contents of T_1 and T_2. Thus, the interaction between T_1 and T_2 is only via the changes to the control state.

When a rewrite rule r is applied to the tree, rewriting a sub-tree T_1 to T_2, there are two possibilities: either T_2 develops independently of the rest of the tree for the remainder of the run, or T_2 appears as a strict sub-tree of a later rewrite rule application. In the former case, a new independent sub-tree has been generated, while, in the later, no new independent tree has been created. When T_2 is independent we say that an independent sub-tree has been generated via rule r.

Adapting the *thread interfaces* introduced by La Torre and Parlato in their analysis of scope-bounded multi-stack pushdown systems [26], we define a notion of *independent sub-tree interfaces*, which we will refer to simply as *interfaces*.

Definition 6.1 (Independent Sub-Tree Interfaces)**.** *For a senescent GTRS with lifespan k and rewrite rule-set $\{r_1, \ldots, r_\ell\}$, an* independent sub-tree interface *α is defined to be a sequence $(p_1, b_1, \vec{\eta}_1) \ldots (p_m, b_m, \vec{\eta}_m)$ of triples in $\mathcal{P} \times \{0, 1\} \times \mathbb{N}^\ell$ with $m \leq k$.*

An interface α describes the external effect of the evolution of a sub-tree over up to k control state changes. A sequence $\alpha = (p_1, b_1, \vec{\eta}_1), \ldots, (p_m, b_m, \vec{\eta}_m)$ describes the sequence of control state changes $p_1, \ldots, p_m$ witnessed by the sub-tree before it becomes fossilised (or ceases to

change for the remainder of the run). The component b_i indicates whether the subtree effected the control state change (via the application of a rewrite rule modifying both the tree and the control state) or whether the control state change is supposed to have been made by an external independent sub-tree.

The final component $\vec{\eta}_i = (\eta_i^{r_1}, \ldots, \eta_i^{r_\ell})$ indicates how many new independent sub-trees are generated during the lifespan of the sub-tree. That is, during the run described by α, we have η_i^r independent sub-trees generated using rule r after the control state has been changed to p_i but before the change to control state p_{i+1} takes place. Note if a rule both changes the control state and generates a new independent sub-tree, the sub-tree is considered to have been generated *after* the control state has changed.

6.2 Examples of Independent Sub-Tree Interfaces

In the following, by abuse of notation, let $(p_1, T_1) \longrightarrow (p_2, T_2)$ for given control states p_1, p_2 and trees T_1, T_2 denote a rule $(p_1, \mathcal{T}_1) \longrightarrow (p_2, \mathcal{T}_2)$ where $\mathcal{L}(\mathcal{T}_1) = \{T_1\}$ and $\mathcal{L}(\mathcal{T}_2) = \{T_2\}$. Also, recall $\vec{0}$ to denotes the tuple $(0, \ldots, 0)$ and $\vec{i}$ denotes the tuple where all components are 0 except the ith, which is 1.

Consider a senescent GTRS with rules $\{r_1, \ldots, r_5\}$ where

$$\begin{aligned} r_1 &= (p_1, T_0) \longrightarrow (p_2, n(T_1, T_2)) \\ r_2 &= (p_2, T_1) \longrightarrow \left(p_2, T_1^1\right) \\ r_3 &= (p_2, T_2) \longrightarrow \left(p_3, T_2^1\right) \\ r_4 &= \left(p_3, T_2^1\right) \longrightarrow \left(p_4, T_2^2\right) \\ r_5 &= \left(p_4, T_1^1\right) \longrightarrow \left(p_5, T_1^2\right) \ . \end{aligned}$$

Now consider the run formed from $r_1, \ldots, r_5$ in sequence,

$$\begin{aligned} (p_1, T_0) &\longrightarrow (p_2, n(T_1, T_2)) \longrightarrow \left(p_2, n\left(T_1^1, T_2\right)\right) \\ &\longrightarrow \left(p_3, n\left(T_1^1, T_2^1\right)\right) \longrightarrow \left(p_4, n\left(T_1^1, T_2^2\right)\right) \longrightarrow \left(p_5, n\left(T_1^2, T_2^2\right)\right) \ . \end{aligned}$$

Below we present several alternative decompositions of the above run into interfaces. In the first decomposition, we take a lifespan of 5. In this case, we may simply have the decomposition

$$\left(p_1, 0, \vec{0}\right), \quad \left(p_2, 1, \vec{0}\right), \quad \left(p_3, 1, \vec{0}\right), \quad \left(p_4, 1, \vec{0}\right), \quad \left(p_5, 1, \vec{0}\right)$$

indicating that no new independent sub-trees are considered to have been generated, and thus, all control state changes are effected by the evolution of the original tree. Note that $b_1 = 0$ since the control state was initially p_1.

However, the above run can also be decomposed if the lifespan is set to 4. One such decomposition can be obtained by considering the application of the rule r_1 to generate $n(T_1, T_2)$, where $n(T_1, T_2)$ is a new independent sub-tree. Using $\triangle$ to denote an independent sub-tree that has been generated, we can decompose the run into two runs

$$(p_1, T_0) \longrightarrow (p_2, \triangle)$$

and the run of the generated independent sub-tree

$$\begin{aligned} (p_2, n(T_1, T_2)) &\longrightarrow \left(p_2, n\left(T_1^1, T_2\right)\right) \\ &\longrightarrow \left(p_3, n\left(T_1^1, T_2^1\right)\right) \longrightarrow \left(p_4, n\left(T_1^1, T_2^2\right)\right) \longrightarrow \left(p_5, n\left(T_1^2, T_2^2\right)\right) \ . \end{aligned}$$

These two runs give rise to two independent sub-tree interfaces that can be combined to represent the original run.

$$\begin{array}{llll} \left(p_1, 0, \vec{0}\right), & \left(p_2, 1, \vec{1}\right) & & \\ & \left(p_2, 0, \vec{0}\right), & \left(p_3, 1, \vec{0}\right), & \left(p_4, 1, \vec{0}\right), \quad \left(p_5, 1, \vec{0}\right) \end{array}$$

The upper interface comes from the first part of the decomposed run, and the lower interface represents the second part. Note, the lifespan of 4 is respected and $\vec{1}$ indicates that an independent sub-tree has been generated as the RHS of r_1.

Finally, we observe that the evolution of T_1^1 and T_2^1 are independent. Hence, we could be more eager in our generation of independent sub-trees. That is, we can decompose the original run into the following runs.

$$(p_1, T_0) \longrightarrow (p_2, \triangle)$$

and

$$(p_2, n(T_1, T_2)) \longrightarrow (p_2, n(\triangle, T_2)) \longrightarrow (p_3, n(\triangle, \triangle))$$

where the evolution of T_1^1 is given by

$$\left(p_2, T_1^1\right) \longrightarrow \left(p_3, T_1^1\right) \longrightarrow \left(p_4, T_1^1\right) \longrightarrow \left(p_5, T_1^2\right)$$

and the evolution of T_2^1 by

$$\left(p_3, T_2^1\right) \longrightarrow \left(p_4, T_2^2\right) \longrightarrow \left(p_5, T_2^2\right) \ .$$

Note, the control state change to p_5 was effected by the evolution of T_1^1 and the change to p_4 by the evolution of T_2^1. The respective interfaces for the above runs, aligned to suggest how they combine, are

$$\begin{array}{llll} \left(p_1, 0, \vec{0}\right), & \left(p_2, 1, \vec{1}\right) & & \\ & \left(p_2, 0, \vec{2}\right), & \left(p_3, 1, \vec{4}\right) & \\ & \left(p_2, 0, \vec{0}\right), & \left(p_3, 0, \vec{0}\right), & \left(p_4, 0, \vec{0}\right), \quad \left(p_5, 1, \vec{0}\right) \\ & & \left(p_3, 0, \vec{0}\right), & \left(p_4, 1, \vec{0}\right), \quad \left(p_5, 0, \vec{0}\right) \ . \end{array}$$

Each column represents a single control state change. It is important that in each column there is exactly one independent sub-tree for which $b_i = 1$. That is, each control state change is performed by exactly one independent sub-tree.

6.3 Representing Interfaces

In this section we show that interfaces α can be generated as the Parikh image of regular automata.

For each rule $r \in \mathcal{R}$ and sequence $(p_1, b_1), \ldots, (p_m, b_m)$ with $m \leq k$ we will build a regular automaton $\mathcal{A}$ over the alphabet

$$\Gamma_I = \{(r, i) \mid r \in \mathcal{R} \wedge 1 \leq i \leq m\} \ .$$

By abuse of notation, for a run over a word $w \in \Sigma_I^*$, we define

$$\textsc{Parikh}(w) = (\vec{\eta}_1, \ldots, \vec{\eta}_m)$$

where for all $1 \leq i \leq m$ we have $\vec{\eta}_i = (\eta_i^{r_1}, \ldots, \eta^{r_\ell})$ and $\eta_i^r = |w|_{(r,i)}$. This generalises to $\textsc{Parikh}(\mathcal{A})$ in the natural way.

In particular, we build $\mathcal{A}$ such that, if $(\vec{\eta}_1, \ldots, \vec{\eta}_m)$ is an element of $\textsc{Parikh}(\mathcal{A})$ then there is an independent sub-tree interface

$$(p_1, b_1, \vec{\eta}_1), \ldots, (p_m, b_m, \vec{\eta}_m)$$

beginning with a tree $T \in \mathcal{L}(\mathcal{T}_1)$ where $r = (p, \mathcal{T}) \longrightarrow (p_1, \mathcal{T}_1)$.

We obtain the above regular automaton as follows. First, from G, r and $(p_1, b_1), \ldots, (p_m, b_m)$ we build a weakly extended sGTRS G_I that simulates a run of G from a subtree appearing on the RHS of r, passing precisely the control states $p_1, \ldots, p_m$ and only effecting a control state change with a rule in G if $b_i = 1$ (else G_I guesses the control state change). The output of this sGTRS gives us information on the independent sub-trees created during the run. Then, using Lemma 3.1 (Parikh Image of Weakly Extended sGTRS) we obtain a regular automaton as required.

Definition 6.2 (G_I)**.** *Given a senescent GTRS $G = (\mathcal{P}, \Sigma, \mathcal{R})$ with lifespan k, an $r \in \mathcal{R}$ and sequence $(p_1, b_1), \ldots, (p_m, b_m)$ with $m \leq k$ we construct a weakly extended sGTRS $G_I = (\mathcal{P}_I, \Sigma_I, \Gamma_I, \mathcal{R}_I)$ where, letting $\mathcal{T}$ be the tree automaton on the RHS of r,*

$$\begin{aligned}
\mathcal{P}_I &= \{(p_1, b_1, 1), \ldots, (p_m, b_m, m)\} \\
\Sigma_I &= \Sigma \uplus \{\triangle, \diamond\} \\
\Gamma_I &= \{(r, i) \mid r \in \mathcal{R} \wedge 1 \leq i \leq m\}
\end{aligned}$$

and $\mathcal{R}_I$ is the set

$$\begin{array}{l}
\left\{((p_1, b_1, 1), \mathcal{T}_\diamond) \xrightarrow{\varepsilon} ((p_1, b_1, 1), \mathcal{T})\right\} \cup \\
\left\{((p_i, b_i, i), \mathcal{T}_1) \xrightarrow{\varepsilon} ((p_i, b_i, i), \mathcal{T}_2) \;\middle|\; \begin{array}{c} 1 \leq i \leq m \wedge \\ (p_i, \mathcal{T}_1) \longrightarrow (p_i, \mathcal{T}_2) \in \mathcal{R} \end{array}\right\} \cup \\
\left\{((p_i, b_i, i), \mathcal{T}_1) \xrightarrow{\varepsilon} ((p_{i+1}, 1, i+1), \mathcal{T}_2) \;\middle|\; \begin{array}{c} 1 \leq i < m \wedge b_{i+1} = 1 \wedge \\ (p_i, \mathcal{T}_1) \longrightarrow (p_{i+1}, \mathcal{T}_2) \in \mathcal{R} \end{array}\right\} \cup \\
\left\{((p_i, b_i, i), \mathcal{T}_a) \xrightarrow{\varepsilon} ((p_{i+1}, 0, i+1), \mathcal{T}_a) \;\middle|\; \begin{array}{c} 1 \leq i < m \wedge b_{i+1} = 0 \wedge \\ a \in \Sigma_I \wedge a \textit{ has arity } 0 \end{array}\right\} \cup \\
\left\{((p_i, b_i, i), \mathcal{T}_1) \xrightarrow{(r,i)} ((p_i, b_i, i), \mathcal{T}_\triangle) \;\middle|\; \begin{array}{c} 1 \leq i \leq m \wedge \\ r = (p_i, \mathcal{T}_1) \longrightarrow (p_i, \mathcal{T}_2) \in \mathcal{R} \end{array}\right\} \cup \\
\left\{((p_i, b_i, i), \mathcal{T}_1) \xrightarrow{(r,i+1)} ((p_{i+1}, 1, i+1), \mathcal{T}_\triangle) \;\middle|\; \begin{array}{c} 1 \leq i < m \wedge b_{i+1} = 1 \wedge \\ r = (p_i, \mathcal{T}_1) \longrightarrow (p_{i+1}, \mathcal{T}_2) \\ \wedge\ r \in \mathcal{R} \end{array}\right\}
\end{array}$$

and both $\triangle$ and $\diamond$ have arity 0.

In the above definition, we use $\diamond$ to be the starting label of G_I, and the first set of rules contains only the rule generating a (independent sub-)tree that could have been created by rule r. The second set of rules simply simulates the rules of G that do not change the control state. The next two sets of rules take care of the cases where either the control state change is effected by the independent sub-tree under consideration ($b_i = 1$), or whether the control state change is effected by another (independent) part of the tree ($b_i = 0$). The final two sets of rules take care of the generation of new independent sub-trees. That is, when applying a rule of G,

instead of the new tree appearing in the current tree, a place-holder tree (accepted by $\mathcal{T}_\triangle$) is created. Note, since $\triangle$ is a new label, the place-holder sub-tree cannot be rewritten during the remainder a run of G_I.

Using G_I we are able to build a regular representation of the independent sub-trees generated during a run matching a given interface.

Definition 6.3 ($\mathcal{A}_I$)**.** *Given a senescent GTRS $G = (\mathcal{P}, \Sigma, \mathcal{R})$ with lifespan k, an $r \in \mathcal{R}$ and sequence $(p_1, b_1), \ldots, (p_m, b_m)$ with $m \leq k$ we construct G_I as above, and then via Lemma 3.1 (Parikh Image of Weakly Extended sGTRS) a regular automaton $\mathcal{A}_I$ such that there is a run*

$$((p_1, b_1, 1), T_1) \xrightarrow{w} ((p_m, b_m, m), T_2)$$

where $T_1 \in \mathcal{L}(\mathcal{T}_\diamond)$ and T_2 is any tree iff Parikh$(w) \in$ Parikh$(\mathcal{A}_I)$.

6.4 Reduction to Reset Petri-Nets

6.4.1 Interface Summaries

We reduce the control state reachability problem for senescent GTRSs to the coverability problem for reset Petri nets. To do so, we construct a reset Petri net whose control states hold a sequence $(p_1, b_1) \ldots (p_m, b_m)$ where $m \leq k$. It will also have a set of counters

$$X_G = \{x_i^r \mid 1 \leq i \leq m\} \ .$$

We will refer to this sequence as an *interface summary*. Such a summary will summarise the combination of a number of interfaces. Each p_i indicates that the ith next control state is p_i (with p_1 being the current control state), and b_i will indicate whether an independent sub-tree has already been generated to account for the control state change. The value of each counter x_i^r indicates how many independent sub-trees are generated using rule r between the ith and $(i+1)$th control state by the combination of the thread interfaces in the summary.

There are two operations we perform on the interface summary: addition and resolution.

Addition Addition refers to the addition of a thread interface to a given summary. Suppose we have a summary $((p_1, b_1), \ldots, (p_m, b_m), \pi)$ where π gives the valuation of the counters. Now suppose we want to add to the summary the effect of the evolution of an independent sub-tree with interface

$$(p'_1, b'_1, \vec{\eta}_1) \ldots (p'_{m'}, b'_{m'}, \vec{\eta}_{m'}) \ .$$

We first require the two sequences $(p_1, b_1) \ldots (p_m, b_m)$ and $(p'_1, b'_1) \ldots (p'_{m'}, b'_{m'})$ to be *compatible*. There are two conditions for this.

1. They must agree on their control states. That is, for all $1 \leq i \leq min(m, m')$ we have $p_i = p'_i$.
2. At most one independent sub-tree can effect a control state change. That is, for all $1 \leq i \leq min(m, m')$ we do not have $b_i = b'_i = 1$.

We first define the addition only over the states and bits, that is, we define

$$(p_1, b_1), \ldots, (p_m, b_m) \ +\!+ \ (p'_1, b'_1) \ldots (p'_{m'}, b'_{m'})$$

when the two are compatible to be,

1. when $m \leq m'$,
$$(p_1, b''_1) \dots (p_m, b''_m) \left(p'_{m+1}, b'_{m+1}\right) \dots (p'_{m'}, b'_{m'})$$
2. and when $m > m'$,
$$(p_1, b''_1) \dots (p_{m'}, b''_{m'}) (p_{m'+1}, b_{m'+1}) \dots (p_m, b_m)$$

where in both cases $b''_i = 1$ if $b_i = 1$ or $b'_i = 1$, and otherwise $b''_i = 0$.

Then, the addition,
$$(\beta, \pi) \ +\!+ \ (p'_1, b'_1, \vec{\eta}_1) \dots (p'_{m'}, b'_{m'}, \vec{\eta}_{m'})$$
when the two are compatible is (β', π') where
$$\beta' = \beta \ +\!+ \ (p'_1, b'_1) \dots (p'_{m'}, b_{m'})$$
and for all r and i
$$\pi'(x_i^r) = \begin{cases} \pi(x_i^r) + \eta_i^r & i \leq m' \\ \pi(x_i^r) & i > m' \end{cases}$$
That is, we add the sub-trees generated to the appropriate counters of the Petri net.

Resolution Addition of thread interfaces to the summary handles the evolution of new independent sub-trees generated on the run between the current control state p_1 and the next p_2. Once all such trees have been accounted for, we can perform *resolution*. That is, we remove the completed first round from the summary. Note that this can only be done if $b_2 = 1$, that is, some independent sub-tree has taken responsibility for the change to the next control state p_2.

We thus define
$$\textsc{Res}(((p_1, b_1), (p_2, b_2), \dots, (p_m, b_m), \pi)) = ((p_2, b_2), \dots, (p_m, b_m), \pi')$$
when $b_2 = 1$ and where
$$\pi'(x_i^r) = \pi\left(x_{i+1}^r\right)$$
for all $1 \leq i < m$, and
$$\pi'(x_m^r) = 0 \ .$$

6.4.2 Reduction to Coverability

We define a reset Petri-net that has a positive solution to the coverability problem iff the control state reachability problem for the given senescent GTRS G is also positive.

For technical convenience, we assume $r_1 = (p_{src}, \mathcal{T}_1) \longrightarrow (p_{src}, \mathcal{T}_2)$ where $\mathcal{T}_1$ accepts no trees and $\mathcal{T}_2$ accepts only the initial tree T_{src}. The assumption of such a rule does not allow more runs of G since $\mathcal{T}_1$ matches no trees.

Initial Configuration The Petri-net begins in a configuration
$$((p_{src}, 1), \pi_{src})$$
where
$$\pi_{src}(x_i^r) = \begin{cases} 1 & \text{if } i = 1 \text{ and } r = r_1 \\ 0 & \text{otherwise .} \end{cases}$$
This means that the Petri net is simulating a configuration of the senescent GTRS where the control state is p_{src} and the only independent sub-tree that can be generated is the T_{src}.

Addition of New Interfaces The Petri net can then begin simulating execution as follows. It will non-deterministically guess the independent sub-tree interface of the initial tree during a satisfying run of the reachability problem. It will do this by subtracting 1 from the variable $x_1^{r_1}$ then guessing a sequence $(p_1, b_1) \dots (p_m, b_m)$. Since there are only a finite number of possibilities for such a sequence, the guess can be made in the control state. To fully guess an interface, however, the Petri net must also guess the values of $\vec{\eta}_i$ for each $1 \leq i \leq m$. To do this it will simulate (in its control state) the automaton $\mathcal{A}$ generated by Definition 6.3 ($\mathcal{A}_I$), but, instead of outputting a symbol (r, i), it will increment the counter x_i^r.

In the manner described above, the Petri net can update its control state and counter values to perform an addition

$$((p_1, b_1), \dots, (p_m, b_m), \pi) \ +\!+ \ (p'_1, b'_1, \vec{\eta}_1) \dots (p'_{m'}, b'_{m'}, \vec{\eta}_{m'})$$

for the interface summary it is currently storing in its control state and counters, and a guessed new interface generated from some available independent sub-tree.

Resolving The Current Interface Summary Given a configuration

$$((p_1, b_1), \dots, (p_m, b_m), \pi)$$

the Petri net can non-deterministically decide whether to add another interface to the summary, or whether (if $b_2 = 1$) perform a resolution step.

To perform resolution the Petri net first updates the control state to obtain the sequence $(p_2, b_2), \dots, (p_m, b_m)$ (that is, deletes the first tuple), and then updates its marking to

$$\pi'(x_i^r) = \pi\big(x_{i+1}^r\big)$$

for all $1 \leq i < m$, and

$$\pi'(x_m^r) = 0 \ .$$

It does this incrementally from $i = 1$ to $i = m$. For each given i it first uses reset transitions to zero each counter x_i^r. Then, when $i < m$, it performs a loop for each counter, decrementing x_{i+1}^r and incrementing x_i^r. It repeats this loop a non-deterministic number of times before moving to the next counter.

Note that this is not a faithful implementation of the resolution operation since the Petri net cannot ensure that it transfers x_{i+1}^r to x_i^r in its entirety, merely that $x_i^r \leq x_{i+1}^r$. However, "forgetting" the existence of independent sub-trees merely restricts the number of runs and does not add new behaviours. Hence such an inaccuracy is benign (since it is still possible to transfer all sub-trees). The reset operation is used to ensure that no leakage occurs between each i.

Formal Definition We give the formal definition of the reset Petri net $\mathcal{N}_G$ that simulates G with respect to the control state reachability problem. For each $\beta = (p_1, b_1) \dots (p_m, b_m)$ with $1 \leq m \leq k$ and rule $r \in \mathcal{R}$, let $\mathcal{A}_\beta^r = \left(\mathcal{Q}_\beta^r, \Gamma_I, \Delta_\beta^r, q_\beta^r, \left\{f_\beta^r\right\}\right)$ be the regular automaton obtained via Definition 6.3 ($\mathcal{A}_I$) and without loss of generality assume $\mathcal{A}_\beta^r$ has the unique initial state q_β^r and final state f_β^r. We assume for all r and β that $\mathcal{A}_\beta^r$ have disjoint state sets.

Definition 6.4 ($\mathcal{N}_G$)**.** *Given the senescent GTRS G (with notation and assumptions as described in this section), we define the reset Petri net $\mathcal{N}_G = (\mathcal{Q}_G, X_G, \Delta_G)$ where X_G is defined*

above and

$$\begin{aligned}\mathcal{S} =& \{(p_1, b_1) \dots (p_m, b_m) \in (\mathcal{P} \times \{0,1\})^m \mid 1 \leq m \leq k\} \\ \mathcal{Q}_G =& \mathcal{S} \cup \left\{(\beta, q) \in \mathcal{S} \times \mathcal{Q}^r_{\beta'} \mid \beta' \in \mathcal{S} \wedge r \in \mathcal{R}\right\} \cup \left\{\lhd^\beta_i \mid \beta \in \mathcal{S} \wedge 1 \leq i \leq k\right\} \\ \Delta_G =& \Delta_{\text{ADD}} \cup \Delta_{\text{RES}}\end{aligned}$$

where

$$\begin{aligned}\Delta_{\text{ADD}} = &\left\{\beta \xrightarrow[decr(x^r_1)]{} (\beta_1, q^r_{\beta_2}) \mid r \in \mathcal{R} \wedge \beta, \beta_1, \beta_2 \in \mathcal{S} \wedge \beta_1 = \beta \mathbin{++} \beta_2\right\} \cup \\ &\left\{(\beta, q) \xrightarrow[incr(x^r_i)]{} (\beta, q') \;\middle|\; \beta \in \mathcal{S} \wedge \exists r', \beta' \text{ s.t. } q \xrightarrow{(r,i)} q' \in \Delta^{r'}_{\beta'}\right\} \cup \\ &\left\{(\beta, f^r_{\beta'}) \underset{\emptyset}{\to} \beta \mid r \in \mathcal{R} \wedge \beta' \in \mathcal{S}\right\}\end{aligned}$$

and

$$\begin{aligned}\Delta_{\text{RES}} = &\left\{\beta \underset{\widetilde{o}}{\to} \lhd^{\beta'}_1 \;\middle|\; \begin{array}{c}\beta, \beta' \in \mathcal{S} \wedge \\ \beta' = (p_2, b_2) \dots (p_m, b_m) \wedge \\ \beta = (p_1, b_1)\, \beta' \wedge b_2 = 1 \wedge \\ \widetilde{o} = \{res(x^r_1) \mid r \in \mathcal{R}\}\end{array}\right\} \cup \\ &\left\{\lhd^\beta_i \xrightarrow[\{decr(x^r_{i+1}), incr(x^r_i)\}]{} \lhd^\beta_i \mid \beta \in \mathcal{S} \wedge 1 \leq i < k\right\} \cup \\ &\left\{\lhd^\beta_i \underset{\widetilde{o}}{\to} \lhd^\beta_{i+1} \;\middle|\; \beta \in \mathcal{S} \wedge 1 \leq i < k \wedge \widetilde{o} = \left\{res(x^r_{i+1}) \mid r \in \mathcal{R}\right\}\right\} \cup \\ &\left\{\lhd^\beta_k \underset{\emptyset}{\to} \beta \mid \beta \in \mathcal{S}\right\}\end{aligned}$$

Note that the size of $\mathcal{N}_G$ is dominated by the size of the regular automata $\mathcal{A}^r_\beta$. Thus, the size of $\mathcal{N}_G$ is triply exponential in the size of G.

6.5 Correctness of Reduction

We prove that the control state reachability problem for G has a positive solution iff the coverability problem for $\mathcal{N}_G$, initial configuration $((p_{src}, 1), \pi_{src})$, and target configuration $((p_{snk}, 1), \pi_0)$ is also positive.

We prove each direction in the sections that follow.

Lemma 6.1 (Correctness of Reduction)**.** *For a given senescent GTRS G with lifespan k, control states p_{src} and p_{snk}, and tree T_{src}, there is a lifespan restricted run*

$$(p_{src}, T_{src}) \longrightarrow \cdots \longrightarrow (p_{snk}, T)$$

for some T of G iff there is a run

$$((p_{src}, 1), \pi_{src}) \longrightarrow^* ((p_{snk}, 1), \pi)$$

of $\mathcal{N}_G$ for some $\pi_0 \leq \pi$.

Proof. From Lemma 6.2 and Lemma 6.3 below. □

In the following, let the marking $\pi - r$ be the marking

$$(\pi - r)\left(x_j^{r'}\right) = \begin{cases} \pi\left(x_j^{r'}\right) & r' \neq r \vee j > 1 \\ \pi\left(x_j^{r'}\right) - 1 & r' = r \wedge j = 1 \end{cases}.$$

6.5.1 From Senescent GTRS to Reset Petri Nets

Lemma 6.2. *For a given senescent GTRS G with lifespan k, control states p_{src} and p_{snk}, and tree T_{src}, there is a lifespan restricted run*

$$(p_{src}, T_{src}) \longrightarrow \cdots \longrightarrow (p_{snk}, T)$$

for some T of G only if there is a run

$$((p_{src}, 1), \pi_{src}) \longrightarrow^* ((p_{snk}, 1), \pi)$$

of $\mathcal{N}_G$ for some $\pi_0 \leq \pi$.

Proof. The proof proceeds in two steps. We begin with a run

$$(p_1, T_1) \longrightarrow \cdots \longrightarrow (p_h, T_h)$$

of G. First we deconstruct this run into independent sub-trees, coupled with their interfaces. From this deconstruction, we then build a run of $\mathcal{N}_G$.

The deconstruction is a sequence

$$(p_1, id_1, run_1, int_1) \ldots (p_h, id_h, run_h, int_h)$$

where for all $1 \leq i \leq h$ we have that id_i assigns to each node of T_i a natural number indicating the ID of the independent sub-tree the node currently belongs to, run_i maps each ID (natural number) to a run of some G_I (where the appropriate $r, p_1, b_1, \ldots, p_m, b_m$ are defined on-the-fly) which is the evolution of the independent sub-tree in the run up to i, and finally int_i maps each ID to an independent sub-tree interface also representing the run up to i.

We build this sequence by induction (simultaneously arguing its existence). We begin with id_1 mapping each node in T_1 to 1, then

$$run_1(1) = ((p_1, 0, 1), T_1)$$
$$int_1(1) = \left(p_1, 0, \vec{0}\right) .$$

Now, inductively take $(p_i, id_i, run_i, int_i)$ and consider the transition

$$(p_i, C[T]) \longrightarrow (p_{i+1}, C[T'])$$

where $T_i = C[T]$ and $T_{i+1} = C[T']$ and the transition is via rule $r \in \mathcal{R}$.

Let n_p be the parent of T in T_i, should it exist (it does not exist if $T = C[T]$). There are now two cases.

1. There is some $j > i$ such that in the transition

$$\left(p_j, C'\left[T^1\right]\right) \longrightarrow \left(p_{j+1}, C'\left[T^2\right]\right)$$

of the run, the node n_p appears in T^1.

In this case, T' cannot form a new independent sub-tree since it appears as part of the run over the sub-tree including n_p.

We define, for each n in T_{i+1},

$$id_{i+1}(n) = \begin{cases} id_i(n) & n \text{ is in } C \\ id_i(n_p) & n \text{ is in } T' \end{cases} .$$

There are now two further cases, depending on whether the control state is changed by the transition.

(a) When $p_i = p_{i+1}$ we define, for each j in the image of id_{i+1},

$$run_{i+1}(j) = \begin{cases} run_i(j) & j \neq id_i(n_p) \\ \rho & j = id_i(n_p) \end{cases}$$
$$int_{i+1}(j) = int_i(j)$$

where we define ρ as follows. We know $run_i(id_i(n_p))$ is a run to some configuration $((p_i, b, i'), C'[T])$ where there is some C'' such that $T_i = C''[C'[T]]$. We define

$$\rho = run_i(id_i(n_p)) \xrightarrow{\varepsilon} ((p_{i+1}, b, i'), C'[T'])$$

which can be seen to be a transition of G_I due to $r \in \mathcal{R}$.

Note, in all other cases we keep the same run as in run_i. Since $p_i = p_{i+1}$ we maintain (for all independent sub-trees that have not expired their lifespan) that run_i tracks the run up to i.

(b) When $p_i \neq p_{i+1}$ we define $run_{i+1}(j)$ for each j in the image of id_{i+1}. There are two cases.

When $j \neq id_i(n_p)$ we know $run_i(j)$ is a run to some configuration $((p, b, i'), T'')$. If $i' = k$ then the nodes in T'' can no longer be rewritten after the control state change (they are fossilised). Hence, we define $run_{i+1}(j) = run_i(j)$. When $i' < k$, we define

$$run_{i+1}(j) = run_i(j) \xrightarrow{\varepsilon} ((p_{i+1}, 0, i' + 1), T'') \ .$$

Such a transition is always possible by the definition of G_I.

When $j = id_i(n_p)$ we know $run_i(j)$ is a run to some configuration $((p_i, b, i'), C'[T])$ where there is some C'' such that $T_i = C''[C'[T]]$. We define

$$\rho = run_i(j) \xrightarrow{\varepsilon} ((p_{i+1}, 1, i' + 1), C'[T'])$$

which can be seen to be a transition of G_I due to $r \in \mathcal{R}$.

Finally, all j we define $int_i(j)$ when $int_i(j)$ has k tuples, and otherwise,

$$int_{i+1}(j) = \begin{cases} int_i(j)\left(p_{i+1}, 1, \vec{0}\right) & j = id_i(n_p) \\ int_i(j)\left(p_{i+1}, 0, \vec{0}\right) & j \neq id_i(n_p) \end{cases} .$$

2. Either $T = C[T]$, or there is no $j > i$ such that in the transition

$$\left(p_j, C'\left[T^1\right]\right) \longrightarrow \left(p_{j+1}, C'\left[T^2\right]\right)$$

of the run, the node n_p appears in T^1.

In this case, T' can form a new independent sub-tree since its parent node is not read during the remainder of the run.

Let j_* be a natural number not in the image of id_i. We define, for each n in T_{i+1},

$$id_{i+1}(n) = \begin{cases} id_i(n) & n \text{ is in } C \\ j_* & n \text{ is in } T' \end{cases}.$$

There are now two further cases, depending on whether the control state is changed by the transition. In the following, recall $\vec{e}$ is the vector that is zero in all components except the eth, which is 1.

(a) When $p_i = p_{i+1}$ we define, for each j in the image of id_{i+1},

$$run_{i+1}(j) = \begin{cases} run_i(j) & j \neq id_i(n_p) \wedge j \neq j_* \\ \rho & j = id_i(n_p) \\ ((p_{i+1}, 0, 1), T') & j = j_* \end{cases}$$

$$int_{i+1}(j) = \begin{cases} int_i(j) & j \neq id_i(n_p) \wedge j \neq j_* \\ \alpha & j = id_i(n_p) \\ \left(p_{i+1}, 0, \vec{0}\right) & j = j_* \end{cases}$$

where we define ρ and α as follows.

First, for ρ, we know $run_i(id_i(n_p))$ is a run to some configuration $((p_i, b, i'), C'[T])$ where there is some C'' such that $T_i = C''[C'[T]]$. We define

$$\rho = run_i(id_i(n_p)) \xrightarrow{(r,i')} ((p_{i+1}, b, i'), C'[\triangle])$$

which can be seen to be a transition of G_I due to $r \in \mathcal{R}$.

Note, in all other cases we keep the same run as in run_i. Since $p_i = p_{i+1}$ we maintain (for all independent sub-trees that have not expired their lifespan) that run_i tracks the run up to i.

Next, we define α. Let $int_i(n_p) = \alpha'(p_i, b, \vec{\eta})$ for some α', b and $\vec{\eta}$. We define

$$\alpha = \alpha'(p_i, b, \vec{\eta} + \vec{e})$$

when $r = r_e$.

(b) When $p_i \neq p_{i+1}$ we define $run_{i+1}(j)$ for each j in the image of id_{i+1}. There are three cases.

When $j = j_*$, we have $run_{i+1}(j) = (p_{i+1}, T')$.

When $j = id_i(n_p)$ we know $run_i(j)$ is a run to some configuration $((p_i, b, i'), C'[T])$ where there is some C'' such that $T_i = C''[C'[T]]$. We define

$$\rho = run_i(j) \xrightarrow{(r,i+1)} ((p_{i+1}, 1, i' + 1), C'[\triangle])$$

which can be seen to be a transition of G_I due to $r \in \mathcal{R}$.
When $j \neq id_i(n_p)$ and $j \neq j_*$ we know $run_i(j)$ is a run to some configuration $((p, b, i'), T'')$. If $i' = k$ then the nodes in T'' can no longer be rewritten after the control state change (they are fossilised). Hence, we define $run_{i+1}(j) = run_i(j)$. When $i' < k$, we define

$$run_{i+1}(j) = run_i(j) \xrightarrow{\varepsilon} ((p_{i+1}, 0, i' + 1), T'') \ .$$

Such a transition is always possible by the definition of G_I.
Finally, we define for all j that $int_i(j)$ when $int_i(j)$ has k tuples, and otherwise, when $r = r_e$,

$$int_{i+1}(j) = \begin{cases} \left(p_{i+1}, 0, \vec{0}\right) & j = j_* \\ int_i(j)\,(p_{i+1}, 1, \vec{e}) & j = id_i(n_p) \\ int_i(j)\left(p_{i+1}, 0, \vec{0}\right) & j \neq id_i(n_p) \wedge j \neq j_* \ . \end{cases}$$

We have now defined our deconstruction

$$(p_1, id_1, run_1, int_1) \dots (p_h, id_h, run_h, int_h)$$

from which we will construct a run

$$((p_{src}, 1), \pi_{src}) \longrightarrow (\beta_1, \pi_1) \longrightarrow \cdots \longrightarrow (\beta_{h'}, \pi_{h'})$$

of $\mathcal{N}_G$, where $\beta_{h'} = (p_{snk}, 1)$ and $\pi_0 \leq \pi_{h'}$ as required.

Note that the independent sub-tree interfaces in the decomposition of the run are given by int_h. We will now iterate over the run of G from $i = 1$ to $i = h$, building the required run of $\mathcal{N}_G$ as we go. First, let ρ_1 be the run

$$((p_{src}, 1), \pi_{src}) \longrightarrow^* (\beta, \pi)$$

where, recalling that the ID 1 denotes the independent sub-tree corresponding to the evolution of T_{src} and the independent sub-trees it generates, we have

$$(\beta, \pi) = ((p_{src}, 1), \pi_0) \ +\!+ \ int_h(1) \ .$$

It remains to prove that such a run exists. For this, take the G_I defined by the initial rule r_1 (assumed in Section 6.4.2), and the sequence $\beta' = (p_1, b_1) \dots (p_m, b_m)$ where $int_h(1) = (p_1, b_1, \vec{\eta}_1) \dots (p_m, b_m, \vec{\eta}_m)$. Note that in this case β differs from β' only in the value of b_1 (it is 0 in β' and 1 in β). Further, observe that by construction $run_h(1)$ gives a run of G_I outputting w such that $\textsc{Parikh}(w) = (\vec{\eta}_1, \dots, \vec{\eta}_m)$. From this we obtain a run of the accompanying regular automaton $\mathcal{A}_I$ with the same Parikh image and thus a run

$$((p_{src}, 1), \pi_{src}) \xrightarrow[decr\left(x_1^{r_1}\right)]{} \left(\left(\beta, q_{\beta'}^{r_1}\right), \pi_0\right) \longrightarrow^* \left(\left(\beta, f_{\beta'}^{r_1}\right), \pi\right) \longrightarrow (\beta, \pi)$$

of $\mathcal{N}_G$ as required.

Now, inductively assume a run ρ_i. Let (β, π) be the final configuration of ρ_i. We extend ρ_i to build ρ_{i+1} by considering the transition

$$(p_i, T_i) \longrightarrow (p_{i+1}, T_{i+1})$$

of the run of G. As in the above deconstruction, there are two cases to consider.

1. When the transition does not generate a new independent sub-tree, there are two further cases depending on whether a control state change occurs.
   (a) When $p_i = p_{i+1}$ we simply define $\rho_{i+1} = \rho_i$.
   (b) When $p_i \neq p_{i+1}$ we perform a resolution step. That is, we define ρ_{i+1} to be a run
   $$\rho_i \longrightarrow^* (\beta', \pi')$$
   where, recalling ρ_i ends with the configuration (β, π), we have $(\beta', \pi') = \textsc{Res}((\beta, \pi))$. That such a run
   $$\rho_i \longrightarrow \left(\lhd_1^{\beta'}, \pi\right) \longrightarrow^* \left(\lhd_k^{\beta'}, \pi'\right) \longrightarrow (\beta', \pi')$$
   of $\mathcal{N}_G$ exists follows in a straightforward manner from the transitions in $\Delta_{\textsc{Res}}$. Note that to start the resolution, we require $b_2 = 1$. This will always be the case because exactly one independent sub-tree is responsible for firing r and its interface was added when the sub-tree was generated. Similarly, the compatibility conditions come from the construction of run_h from a valid run.
2. When a new independent sub-tree is generated, we again have two cases depending on whether the control state changes. Let j_* be the ID of the new sub-tree.
   (a) When $p_i = p_{i+1}$ we proceed in the same manner as we defined ρ_1. That is, take the G_I defined by the rule r responsible for the transition, and the sequence $\beta_I = (p_1, b_1) \dots (p_m, b_m)$ where $int_h(j_*) = (p_1, b_1, \vec{\eta}_1) \dots (p_m, b_m, \vec{\eta}_m)$. Again, observe that by construction $run_h(j_*)$ gives a run of G_I outputting w such that $\textsc{Parikh}(w) = (\vec{\eta}_1, \dots, \vec{\eta}_m)$. From this we obtain a run of the accompanying regular automaton $\mathcal{A}_I$ with the same Parikh image and thus a run
   $$(\beta, \pi) \xrightarrow[decr(x_1^r)]{} \left(\left(\beta', q_{\beta_I}^{r_1}\right), \pi - r\right) \longrightarrow^* \left(\left(\beta', f_{\beta_I}^{r_1}\right), \pi'\right) \longrightarrow (\beta', \pi')$$
   of $\mathcal{N}_G$ where
   $$(\beta', \pi') = (\beta, \pi - r) \ +\!+ \ int_h(j_*) \ .$$
   That $\pi - r$ exists follows from the fact that by construction of ρ_i we have that $\pi(x_1^r)$ contains the number of independent sub-trees generated by r between the change to p_i and the change to the next control state. Hence, since we only decrement x_1^r when such a sub-tree is generated, we do not fall below 0.
   (b) When $p_i \neq p_{i+1}$ we peform a resolution, a subtraction and an addition in the same way as the previous cases. That is, we build a run
   $$\begin{aligned}\rho_i &\longrightarrow \left(\lhd_1^{\beta^1}, \pi\right) \longrightarrow^* \left(\lhd_k^{\beta^1}, \pi^1\right) \longrightarrow \left(\beta^1, \pi^1\right) \\ &\longrightarrow \left(\left(\beta^2, q_{\beta_I}^{r_1}\right), \pi^1 - r\right) \longrightarrow^* \left(\left(\beta^2, f_{\beta_I}^{r_1}\right), \pi^2\right) \longrightarrow \left(\beta^2, \pi^2\right)\end{aligned}$$
   where $\left(\beta^1, \pi^1\right) = \textsc{Res}((\beta, \pi))$ and
   $$\left(\beta^2, \pi^2\right) = \left(\beta^1, \pi^1 - r\right) \ +\!+ \ int_h(j_*) \ .$$
   The existence of such a run follows by the same arguments presented above.

Thus, we construct ρ_h to some configuration (β, π) where, necessarily, $\pi_0 \leq \pi$ and $\beta = (p_m, 1)$ where by assumption $p_m = p_{snk}$. This follows since β was built by additions of interfaces that did not pass beyond p_m to some p_{m+1}, and the resolution steps occured at the points where the control state changed. The second component of the tuple is 1 since by construction some sub-tree interface was responsible for every control state change. Thus, we are done. □

6.5.2 From Reset Petri Nets to Senescent GTRS

Lemma 6.3. *For a given senescent GTRS G with lifespan k, control states p_{src} and p_{snk}, and tree T_{src}, there is a lifespan restricted run*

$$(p_{src}, T_{src}) \longrightarrow \cdots \longrightarrow (p_{snk}, T)$$

for some T of G if there is a run

$$((p_{src}, 1), \pi_{src}) \longrightarrow^* ((p_{snk}, 1), \pi)$$

of $\mathcal{N}_G$ for some $\pi_0 \leq \pi$.

Proof. Take a run

$$(\beta_1, \pi_1) \longrightarrow^* (\beta_2, \pi_2) \longrightarrow^* \cdots \longrightarrow^* (\beta_h, \pi_h)$$

of $\mathcal{N}_G$ where $(\beta_1, \pi_1), \ldots, (\beta_h, \pi_h)$ are all configurations with a control state in $\mathcal{S}$ that occur on the run.

By induction from $i = 1$ to $i = h$ we build a run of G witnessing the control state reachability property. We assume by induction that for $\beta_i = (p_1, b_1) \ldots (p_m, b_m)$ that for all $m < j \leq k$ and $r' \in \mathcal{R}$, we have $\pi_i\left(x_j^{r'}\right) = 0$. Moreover, we have contexts $C_1, \ldots, C_m$ that are intuitively the trees corresponding to the runs of G built so far (where C_i is paired with the run over control state p_i), but where the independent sub-trees (represented by $\triangle$ from the runs of the G_I used to build the run) are context variables. Since the runs were built to leave these independent sub-trees uninspected, they can be replaced by any tree matching the rule which generated them. More precisely, we have contexts $C_1, \ldots, C_m$ such that for all sequences $\vec{T}_1, \ldots, \vec{T}_m$ where for all $1 \leq j \leq m$ we have

$$\vec{T}_j = \vec{T}_{r_1,j}, \ldots, \vec{T}_{r_\ell,j}$$

and for all r

$$\vec{T}_{r,j} = T^1_{r,j}, \ldots, T^{\pi\left(x_j^r\right)}_{r,j} \in \mathcal{L}(\mathcal{T})^{\pi\left(x_j^r\right)}$$

where $\mathcal{T}$ is the tree automaton on the RHS of r, we have a run ρ_i of G reaching a configuration

$$\left(p_1, C_1\left[\vec{T}_1\right]\right) .$$

and for all $1 < j \leq m$, if $b_j = 1$, we have a run

$$\left(p_{j-1}, C_{j-1}\left[\vec{T}_1, \ldots, \vec{T}_{j-1}\right]\right) \longrightarrow^* \left(p_j, C_j\left[\vec{T}_1, \ldots, \vec{T}_j\right]\right)$$

of G and if $b_j = 0$, we have a run

$$\left(p_j, C_{j-1}\left[\vec{T}_1, \ldots, \vec{T}_{j-1}\right]\right) \longrightarrow^* \left(p_j, C_j\left[\vec{T}_1, \ldots, \vec{T}_j\right]\right) .$$

That is, we have a run to $\left(p_m, C_m\left[\vec{T}_1, \ldots, \vec{T}_m\right]\right)$ which may have single transition gaps where control state changes occur (in which case $b_j = 0$).

In the base case we have $\beta_1 = (p_{src}, 1)$ and $\pi_1 = \pi_{src}$ and the induction hypothesis is satisfied by

$$\rho_1 = (p_{src}, C_1[T_1])$$

where C_1 is the context containing only a root node labelled by a variable and T_1 is necessarily T_{src}.

Now we consider the inductive step. Take the run of $\mathcal{N}_G$

$$(\beta_i, \pi_i) \longrightarrow^* (\beta_{i+1}, \pi_{i+1})$$

and fix $\beta_i = (p_1, b_1) \ldots (p_m, b_m)$ and $C_1, \ldots, C_m$ by induction. There are two cases: either the run uses transition in Δ_{ADD} or Δ_{RES}.

1. If the run uses Δ_{ADD} rules then we have a run of the form

$$\begin{aligned}(\beta_i, \pi_i) &\xrightarrow[decr(x_1^r)]{} \left(\left(\beta_{i+1}, q_\beta^r\right), \pi_i - r\right) \\ &\longrightarrow^* \left(\left(\beta_{i+1}, f_\beta^r\right), \pi_{i+1}\right) \longrightarrow (\beta_{i+1}, \pi_{i+1})\end{aligned}$$

where $\beta_{i+1} = \beta_i \mathbin{++} \beta$. Moreover, for the G_I defined from r and $\beta = (p'_1, b'_1) \ldots (p'_{m'}, b'_{m'})$, we have a run with the Parikh image $(\vec{\eta}_1, \ldots, \vec{\eta}_{m'})$ from (p'_1, T) where T is accepted by the RHS of r and the run has the interface

$$(p'_1, b'_1, \vec{\eta}_1) \ldots (p'_{m'}, b'_{m'}, \vec{\eta}_{m'})$$

such that

$$(\beta_{i+1}, \pi_{i+1}) = (\beta_i, \pi_i - r) \mathbin{++} (p'_1, b'_1, \vec{\eta}_1) \ldots (p'_{m'}, b'_{m'}, \vec{\eta}_{m'}) \ .$$

From the run over G_I, by creating context variable nodes when (r', j) characters are output, we obtain contexts $C'_1, \ldots, C'_{m'}$ such that for all $\vec{T}'_1, \ldots, \vec{T}'_{m'}$ where for all $1 \leq j \leq m$ we have

$$\vec{T}'_j = \vec{T}'_{r_1,j}, \ldots, \vec{T}'_{r_\ell,j}$$

and for all r'

$$\vec{T}'_{r',j} = Y^1_{r',j}, \ldots, Y^{\eta^{r'}_j}_{r',j} \in \mathcal{L}(\mathcal{T}')^{\eta^{r'}_j}$$

where $\mathcal{T}'$ is the tree automaton on the RHS of r', we have a run

$$(p'_1, T) \longrightarrow^* \left(p'_1, C'_1\left[\vec{T}'_1\right]\right)$$

and for all $1 < j \leq m$, if $b'_j = 1$, we have a run

$$\left(p'_{j-1}, C'_{j-1}\left[\vec{T}'_1, \ldots, \vec{T}'_{j-1}\right]\right) \longrightarrow^* \left(p'_j, C'_j\left[\vec{T}'_1, \ldots, \vec{T}'_j\right]\right)$$

of G and if $b'_j = 0$, we have a run

$$\left(p'_j, C'_{j-1}\left[\vec{T}'_1, \ldots, \vec{T}'_{j-1}\right]\right) \longrightarrow^* \left(p'_j, C'_j\left[\vec{T}'_1, \ldots, \vec{T}'_j\right]\right) \ .$$

Using $C_1, \ldots, C_m$ and $C'_1, \ldots, C'_{m'}$ we can establish the induction hypothesis for $(i+1)$. We write

$$C_j\left[\ldots, T^1_{r,1}, \ldots\right]$$

to single out $T^1_{r,1}$ in $C_j\left[\vec{T}_1, \ldots, \vec{T}_j\right]$. Note that since the induction hypothesis holds for all $T^1_{r,1}$ we can select it to match T accepted by the RHS of r used in the definition of $C'_1, \ldots C'_{m'}$ above. Although there are some details we take care of below, the essential idea is to obtain new contexts, satisfying the induction hypothesis, by inserting C'_j in place of $T^1_{r,1}$ in C_j.

There are two (similar) cases, depending on whether $m \leq m'$. In both cases we have that the first $m_{min} = min(m, m')$ tuples of β_{i+1} are

$$(p_1, b''_1) \ldots \left(p_{m_{min}}, b''_{m_{min}}\right),$$

where $b''_j = 1$ iff either $b_1 = 1$ or $b'_1 = 1$. Note, since β_i and β are compatible, there is agreement on the control states. Let x be the context variable corresponding to the position of $T^1_{r,1}$ in each $C_j\left[\vec{T}_1, \ldots, \vec{T}_j\right]$ (using the same x in each context). We write $C_j[\ldots, x, \ldots]$ to isolate this variable, leaving all other variables untouched. To satisfy the induction for all $1 \leq j \leq m_{min}$ we take the context $C_j\left[\ldots, C'_j, \ldots\right]$ (with a suitable variable ordering to make the comparison with π_{i+1}).

Then, we first build ρ_{i+1} by concatenating to ρ_i the run

$$\left(p_1, C_1\left[\ldots, T^1_{r,1}, \ldots\right]\right) \longrightarrow^* \left(p_1, C_1\left[\ldots, C'_1\left[\vec{T}'_1\right], \ldots\right]\right)$$

by simply appending the run to C'_1 above to ρ_i.

Next, for all $1 < j \leq m_{min}$, if $b''_j = 1$, we can, if $b_j = 1$ (implying $b'_j = 0$), build

$$\begin{aligned}\left(p_{j-1}, C_{j-1}\left[\ldots, C'_{j-1}\left[\vec{T}'_1, \ldots, \vec{T}'_{j-1}\right], \ldots\right]\right) &\longrightarrow^* \\ \left(p_j, C_j\left[\ldots, C'_{j-1}\left[\vec{T}'_1, \ldots, \vec{T}'_{j-1}\right], \ldots\right]\right) &\longrightarrow^* \\ \left(p_j, C_j\left[\ldots, C'_j\left[\vec{T}'_1, \ldots, \vec{T}'_j\right], \ldots\right]\right)\end{aligned}$$

and if $b_j = 0$ (implying $b'_j = 1$), build

$$\begin{aligned}\left(p_{j-1}, C_{j-1}\left[\ldots, C'_{j-1}\left[\vec{T}'_1, \ldots, \vec{T}'_{j-1}\right], \ldots\right]\right) &\longrightarrow^* \\ \left(p_j, C_{j-1}\left[\ldots, C'_j\left[\vec{T}'_1, \ldots, \vec{T}'_j\right], \ldots\right]\right) &\longrightarrow^* \\ \left(p_j, C_j\left[\ldots, C'_j\left[\vec{T}'_1, \ldots, \vec{T}'_j\right], \ldots\right]\right).\end{aligned}$$

The remaining case is when $b''_j = b'_j = b_j = 0$ and we build

$$\begin{aligned}\left(p_j, C_{j-1}\left[\ldots, C'_{j-1}\left[\vec{T}'_1, \ldots, \vec{T}'_{j-1}\right], \ldots\right]\right) &\longrightarrow^* \\ \left(p_j, C_{j-1}\left[\ldots, C'_j\left[\vec{T}'_1, \ldots, \vec{T}'_j\right], \ldots\right]\right) &\longrightarrow^* \\ \left(p_j, C_j\left[\ldots, C'_j\left[\vec{T}'_1, \ldots, \vec{T}'_j\right], \ldots\right]\right).\end{aligned}$$

The remainder of the cases are below.

(a) When $m \leq m' \ldots$ we have

$$\beta_{i+1} = (p_1, b''_1) \ldots (p_m, b''_m) \left(p'_{m+1}, b'_{m+1}\right) \ldots (p'_{m'}, b'_{m'})$$

and it remains to define contexts for $(m+1) \leq j \leq m'$, which we set for each j to be

$$C_m[\ldots, C'_j, \ldots]$$

with the runs, when $b'_j = 1$,

$$\left(p'_{j-1}, C_m\left[\ldots, C'_{j-1}\left[\vec{T}'_1, \ldots, \vec{T}'_{j-1}\right], \ldots\right]\right) \longrightarrow^* \left(p'_j, C_m\left[\ldots, C'_j\left[\vec{T}'_1, \ldots, \vec{T}'_j\right], \ldots\right]\right)$$

and if $b_j = 0$, we have a run

$$\left(p'_j, C_m\left[\ldots, C'_{j-1}\left[\vec{T}'_1, \ldots, \vec{T}'_{j-1}\right], \ldots\right]\right) \longrightarrow^* \left(p'_j, C_m\left[\ldots, C'_j\left[\vec{T}'_1, \ldots, \vec{T}'_j\right], \ldots\right]\right).$$

(b) When $m > m'$ we have

$$\beta_{i+1} = (p_1, b''_1) \ldots (p_{m'}, b''_{m'}) \left(p_{m'+1}, b_{m'+1}\right) \ldots (p_m, b'_m)$$

and it remains to define contexts for $(m'+1) \leq j \leq m'$, which we set for each j to be

$$C_j[\ldots, C_{m'}, \ldots]$$

with the runs, when $b_j = 1$,

$$\left(p_{j-1}, C_{j-1}\left[\ldots, C'_{m'}\left[\vec{T}'_1, \ldots, \vec{T}'_{m'}\right], \ldots\right]\right) \longrightarrow^* \left(p'_j, C_j\left[\ldots, C'_{m'}\left[\vec{T}'_1, \ldots, \vec{T}'_{m'}\right], \ldots\right]\right)$$

and if $b_j = 0$, we have a run

$$\left(p'_j, C_{j-1}\left[\ldots, C'_{m'}\left[\vec{T}'_1, \ldots, \vec{T}'_{m'}\right], \ldots\right]\right) \longrightarrow^* \left(p'_j, C_j\left[\ldots, C'_{m'}\left[\vec{T}'_1, \ldots, \vec{T}'_{m'}\right], \ldots\right]\right).$$

To check that the above defined contexts have the right number of variables to be in accordance with π_{i+1} one only need observe that we replaced the tree $T^1_{r,1}$ with the new context (matching $\pi_i - r$), then inserted the new contexts (C'_j) corresponding to the addition of $(\vec{\eta}_1, \ldots, \vec{\eta}_{m'})$ new trees, matching

$$(\beta_{i+1}, \pi_{i+1}) = (\beta_i, \pi_i - r) \ +\!+ \ (p'_1, b'_1, \vec{\eta}_1) \ldots (p'_{m'}, b'_{m'}, \vec{\eta}_{m'}) \ .$$

2. If the run uses Δ_{RES} then we have a run

$$(\beta_i, \pi_i) \longrightarrow \left(\lhd_1^{\beta_{i+1}}, \pi_i\right) \longrightarrow^* \left(\lhd_k^{\beta_{i+1}}, \pi_{i+1}\right) \longrightarrow (\beta_{i+1}, \pi_{i+1})$$

where $\beta_i = (p_1, b_1)\,(p_2, b_2)\ldots(p_m, b_m)$ and $\beta_{i+1} = (p_2, b_2)\ldots(p_m, b_m)$ and $b_2 = 1$. Furthermore, for all $r \in \mathcal{R}$ it is the case that for all $1 \leq j < m$ we have $\pi_{i+1}(x_j^r) \leq \pi_i(x_{j+1}^r)$ and for all $m \leq j \leq k$ we have $\pi_{i+1}(x_j^r) = 0$.

We obtain $C'_2, \ldots, C'_m$ to satisfy the induction from the $C_1, \ldots, C_m$ we have by the induction hypothesis. For each rule r, we can fix a tree T_r that is accepted by the RHS of r (we made this benign assumption at the beginning of the section). Thus, intuitively, since C'_j has fewer holes than C_j we can simply plug each C_j with the T_r to obtain C'_j.

Thus we define for all $2 \leq j \leq m$ the context C'_j such that for all $\vec{T}'_2, \ldots, \vec{T}'_m$ with for all $2 \leq j' \leq m$

$$\vec{T}'_{j'} = \vec{T}'_{r_1,j'}, \ldots, \vec{T}'_{r_\ell,j'}$$

and for all r

$$\vec{T}'_{r,j'} = Y^1_{r,j'}, \ldots, Y^{\pi_{i+1}(x^r_{j'-1})}_{r,j'} \in \mathcal{L}(\mathcal{T})^{\pi_{i+1}(x^r_{j'-1})}$$

where $\mathcal{T}$ is the tree automaton on the RHS of r we have

$$C'_j\left[\vec{T}'_2, \ldots, \vec{T}'_m\right] = C_j\left[\vec{T}_1, \ldots, \vec{T}_m\right]$$

where $\vec{T}_1, \ldots, \vec{T}_m$ is given by

$$\vec{T}_1 = \vec{T}_{r_1,1}, \ldots, \vec{T}_{r_\ell,1}$$

and for all r

$$\vec{T}_{r,1} = T_r, \ldots, T_r \in \mathcal{L}(\mathcal{T})^{\pi_i(x^r_1)}$$

and for all $2 \leq j' \leq m$

$$\vec{T}_{j'} = \vec{T}_{r_1,j'}, \ldots, \vec{T}_{r_\ell,j'}$$

and for all r

$$\vec{T}_{r,j'} = Y^1_{r,j'}, \ldots, Y^{\pi_i(x^r_{j'})}_{r,j'}, T_r, \ldots, T_r \in \mathcal{L}(\mathcal{T})^{\pi_i(x^r_{j'})}$$

where $\mathcal{T}$ is the tree automaton on the RHS of r.

To check that $C'_2, \ldots, C'_m$ satisfy the induction hypothesis we first construct ρ_{i+1} combining ρ_i with (since $b_2 = 1$)

$$\left(p_1, C_1\left[\vec{T}_1\right]\right) \longrightarrow^* \left(p_2, C'_2\left[\vec{T}'_2\right]\right)$$

observing that the trees in $\vec{T}_1$ are out of the scope of the quantification and thus fixed. The existence of partial runs for all $1 < j \leq m$ follows directly from the definition of $C'_2, \ldots, C'_m$ and the induction hypothesis (substituting T_r where appropriate as above).

Thus we are able to maintain the induction hypothesis. When we consider (β_h, π_h) we thus get from ρ_h and $\beta_h = (p_{snk}, 1)$ a run

$$(p_{src}, T_{src}) \longrightarrow^* \left(p_{snk}, C_1\left[\vec{T}_1\right]\right)$$

for some C_1 and any appropriate sequence $\vec{T}_1$ (of which one necessarily exists by assumption). This witnesses the reachability property as required. □

7 Conclusion

We have introduced a sub-class of ground tree rewrite systems with state that has a decidable reachability problem. Our sub-class, *senescent ground tree rewrite systems*, takes scope-bounded pushdown systems as inspiration. In this setting, a node of the tree "ages" whenever the control state changes. A node that reaches a fixed age without being rewritten becomes fossilised and thus may no longer be changed. This model generalises weakly extended ground tree rewrite systems by allowing an arbitrary number of control state changes.

We showed that the control state reachability problem is inter-reducible to coverability of reset Petri-nets, and is thus $\mathbf{F}_\omega$-complete. This is a surprising increase in complexity compared to scope-bounded multi-pushdown systems for which the analogous problem is PSPACE-complete.

Thus, we obtain a natural model that captures a rich class of behaviours while maintaining decidability of reachability properties. Moreover, since extending the control state reachability problem to the regular reachability problem results in undecidability, we know we are close to the limits of decidability.

For future work, we would like to investigate the encoding of additional classes of multi-stack pushdown systems (e.g. ordered, phased-bounded, and relaxed notions of scope-bounding, as well as with added features such as dynamic thread creation) into senescent GTRS. This may lead to further generalisations of our model.

It has been shown by tools such as FAST [8] and TREX [2] that high (even undecidable) complexities do not preclude the construction of successful model checkers. Hence, we would like to study practical verification algorithms for our model and their implementation, which may use the aforementioned tools as components.

Acknowledgements We are grateful for helpful and informative discussions with Anthony Lin, Sylvain Schmitz, Christoph Haase, and Arnaud Carayol. This work was supported by the Engineering and Physical Sciences Research Council [EP/K009907/1].

References

[1] P. A. Abdulla, B. Jonsson, P. Mahata, and J. d'Orso. Regular tree model checking. In *CAV*, pages 555–568, 2002.

[2] A. Annichini, A. Bouajjani, and M. Sighireanu. Trex: A tool for reachability analysis of complex systems. In *CAV*, pages 368–372, 2001.

[3] T. Araki and T. Kasami. Some Decision Problems Related to the Reachability Problem for Petri Nets. *Theoretical Computer Science*, 3(1):85–104, 1977.

[4] M. F. Atig, B. Bollig, and P. Habermehl. Emptiness of multi-pushdown automata is 2etime-complete. In *Developments in Language Theory*, pages 121–133, 2008.

[5] M. F. Atig, A. Bouajjani, and S. Qadeer. Context-bounded analysis for concurrent programs with dynamic creation of threads. *Logical Methods in Computer Science*, 7(4), 2011.

[6] T. Ball, V. Levin, and S. K. Rajamani. A decade of software model checking with slam. *Commun. ACM*, 54(7):68–76, 2011.

[7] T. Ball and S. K. Rajamani. Bebop: A symbolic model checker for boolean programs. In *SPIN*, pages 113–130, 2000.

[8] S. Bardin, A. Finkel, J. Leroux, and L. Petrucci. Fast: acceleration from theory to practice. *STTT*, 10(5):401–424, 2008.

[9] A. Bouajjani, J. Esparza, and O. Maler. Reachability analysis of pushdown automata: Application to model-checking. In *CONCUR*, pages 135–150, 1997.

[10] L. Bozzelli, M. Kretínský, V. Rehák, and J. Strejcek. On decidability of ltl model checking for process rewrite systems. *Acta Inf.*, 46(1):1–28, 2009.

[11] W. S. Brainerd. Tree generating regular systems. *Information and Control*, 14(2):217–231, 1969.

[12] 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.

[13] A. Cyriac, P. Gastin, and K. N. Kumar. MSO decidability of multi-pushdown systems via split-width. In *CONCUR*, pages 547–561, 2012.

[14] M. Dauchet, T. Heuillard, P. Lescanne, and S. Tison. Decidability of the confluence of finite ground term rewrite systems and of other related term rewrite systems. *Inf. Comput.*, 88(2):187–201, 1990.

[15] M. Dauchet and S. Tison. The theory of ground rewrite systems is decidable. In *LICS*, pages 242–248, 1990.

[16] J. Esparza, A. Kucera, and S. Schwoon. Model checking ltl with regular valuations for pushdown systems. *Inf. Comput.*, 186(2):355–376, 2003.

[17] C. P. Estes. *The Faithful Gardener: A Wise Tale About That Which Can Never Die*. Tree clause book. HarperCollins, 1995.

[18] A. Finkel, B. Willems, and P. Wolper. A direct symbolic approach to model checking pushdown systems. In *INFINITY*, volume 9, pages 27–37, 1997.

[19] S. Ginsburg and E. H. Spanier. Semigroups, presburger formulas, and languages. *Pacific Journal of Mathematics*, 16:285 – 296, 1966.

[20] S. Göller and A. W. Lin. Refining the process rewrite systems hierarchy via ground tree rewrite systems. In *CONCUR*, pages 543–558, 2011.

[21] C. Haase. Subclasses or presburger arithmetic and the weak exponential-time hierarchy. Under submission.

[22] N. D. Jones and S. S. Muchnick. Even simple programs are hard to analyze. *J. ACM*, 24(2):338–350, 1977.

[23] R. M. Karp and R. E. Miller. Parallel program schemata: A mathematical model for parallel computation. In *SWAT (FOCS)*, pages 55–61. IEEE Computer Society, 1967.

[24] S. La Torre and M. Napoli. Reachability of multistack pushdown systems with scope-bounded matching relations. In *CONCUR*, pages 203–218, 2011.

[25] S. La Torre and M. Napoli. A temporal logic for multi-threaded programs. In *IFIP TCS*, pages 225–239, 2012.

[26] S. La Torre and G. Parlato. Scope-bounded multistack pushdown systems: Fixed-point, sequentialization, and tree-width. In *FSTTCS*, pages 173–184, 2012.

[27] A. W. Lin. Weakly-synchronized ground tree rewriting - (with applications to verifying multi-threaded programs). In *MFCS*, pages 630–642, 2012.

[28] C. Löding. *Infinite Graphs Generated by Tree Rewriting*. PhD thesis, RWTH Aachen, 2003.

[29] P. Madhusudan and G. Parlato. The tree width of auxiliary storage. In *POPL*, pages 283–294, 2011.

[30] M. Maidl. The common fragment of ctl and ltl. In *FOCS*, pages 643–652, 2000.

[31] R. Mayr. *Decidability and Complexity of Model Checking Problems for Infinite-State Systems*. PhD thesis, TU-München, 1998.

[32] K. McAloon. Petri nets and large finite sets. *Theoretical Computer Science*, 32:173–183, 1984.

[33] R. Piskac. *Decision Procedures for Program Synthesis and Verification*. PhD thesis, Laboratoire d'Analyse et de Raisonnement Authomatisés, École Polytechnique Fédérale de Lausanne, 2011.

[34] L. Pottier. Minimal solutions of linear diophantine systems: Bounds and algorithms. In *RTA*, pages 162–173, 1991.

[35] S. Qadeer. The case for context-bounded verification of concurrent programs. In *Proceedings of the 15th international workshop on Model Checking Software*, SPIN '08, pages 3–6, Berlin, Heidelberg,

2008. Springer-Verlag.

[36] R. Büchi. Regular canonical systems. *Archiv fur Math. Logik und Grundlagenforschung 6*, pages 91–111, 1964.

[37] T. W. Reps, S. Schwoon, S. Jha, and D. Melski. Weighted pushdown systems and their application to interprocedural dataflow analysis. *Sci. Comput. Program.*, 58(1-2):206–263, 2005.

[38] S. Qadeer and J. Rehof. Context-bounded model checking of concurrent software. In *TACAS*, pages 93–107, 2005.

[39] S. Schmitz and P. Schnoebelen. Algorithmic Aspects of WQO Theory. http://cel.archives-ouvertes.fr/cel-00727025, August 2012.

[40] P. Schnoebelen. Verifying lossy channel systems has nonprimitive recursive complexity. *Inf. Process. Lett.*, 83(5):251–261, 2002.

[41] P. Schnoebelen. Revisiting ackermann-hardness for lossy counter machines and reset petri nets. In *MFCS*, pages 616–628, 2010.

[42] S. Schwoon. *Model-checking Pushdown Systems*. PhD thesis, Technical University of Munich, 2002.

[43] A. W. To. *Model Checking Infinite-State Systems: Generic and Specific Approaches*. PhD thesis, LFCS, School of Informatics, University of Edinburgh, 2010.

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