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.

Categories and Subject Descriptors Theory of computation [*Formal languages and automata theory*]: Rewrite systems

General Terms Theory

Keywords Ground tree rewrite systems, ground term rewrite systems, automata, concurrency, scope-bounding, bounded-context switches, under-approximation, reachability, petri-nets, reset petri-nets, Ackermann-hard, pushdown systems

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, the stack is 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, 17].

Pushdown systems can accurately model the control-flow of first-order programs [22] and as such they have been well-studied as an automata-theoretic approach to software model checking (E.g. [9, 16, 17, 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 successful 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.

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 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 hard-

CSL-LICS 2014, July 14–18, 2014, Vienna, Austria.

ACM 978-1-4503-2886-9. . . $15.00.
http://dx.doi.org/10.1145/2603088.2603112

[1] Also known as Ground Term Rewrite Systems

ened 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 and also allow additional features such as dynamic thread creation (by creating a new branch of the tree). 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.

An extended version of this work appears on `arXiv.org` [21].

2. Related Work

Abdulla *et al.* [1] define a regular model checking algorithm for tree automatic structures where 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 on the number of times a node is changed during a run. 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 6.1, 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 well studied as generators of graphs (e.g. [28]) and have decidable verification problems for repeated reachability [28], first-order logic [15], confluence [14], &c. However, LTL and CTL model checking are undecidable [10, 19, 28]. They have intimate connections (e.g. [19, 28]) 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 — based on bounded tree- [29] or split-width [13] — 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 denote by $\mathbf{F}_\omega$ both the Ackermann function and the class of problems solvable in $\mathbf{F}_\omega$-time. Following Schmitz [39], we have the class $\mathbf{F}_\omega$ of problems computable in Ackermannian time, which 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 $\mathtt{rank} : \Sigma \mapsto \mathbb{N}$. 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 $\mathtt{rank}(a) = m$ 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 a set of 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 $\mathtt{rank}(x_i) = 0$ 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 $\mathtt{rank}(a) = m$. 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 $\mathtt{rank}(a) = m$ 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 a tree T, let $\mathcal{T}_T$ be an NTA accepting only T. For example, $\mathcal{T}_{a(b)}$ is the automaton accepting only the tree $a(b)$, and $\mathcal{T}_a$ accepts only the tree containing a single node labelled a. Note, we do not use natural numbers as tree labels, hence $\mathcal{T}_1, \mathcal{T}_2, \ldots$ may range over all NTAs.

3.2 Reset Petri-Nets

We give a simplified presentation of reset Petri-nets as counter machines with increment, decrement and reset operations. This is easily 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 will write π_0 for the marking assigning zero to all counters. We write

$$p \underset{\widetilde{o}}{\to} p'$$

to denote a rule $(p, \widetilde{o}, p') \in \Delta$ and omit the set notation when $\widetilde{o}$ 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, operations are applied in the order $decr(x), res(x), incr(x)$, and if x is 0, attempting to apply $decr(x)$ causes the Petri net to become stuck. We write $(q, \pi) \longrightarrow^* (q', \pi')$ for 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 $\pi' \leq \pi''$.*

Coverability for reset Petri nets is decidable via the Karp-Miller algorithm [23] whose complexity is bounded by $\mathbf{F}_\omega$ [32]. In fact, the problem is $\mathbf{F}_\omega$-complete [40, 41]. In contrast, the reachability problem 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 *asks if there is a run $(q, \pi) \longrightarrow^* (q', \pi')$ of $\mathcal{N}$.*

3.3 Ground Tree Rewrite Systems with State

We consider a generalisation of GTRS where regular automata appear in the rewrite rules. Hence, a single rule may correspond to an infinite number of rules containing concrete trees. Such an extension is common (e.g. [14, 27, 28]). Note, 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. See Section 5 for an example of senescent GTRSs, which is a restriction of sGTRSs.

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 (GTRSs 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 $(p_1, \mathcal{T}_1) \to (p_2, \mathcal{T}_2)$ 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* $(p_1, T_1) \to (p_2, T_2)$ whenever there is a rule $(p_1, \mathcal{T}_1) \to (p_2, \mathcal{T}_2) \in \mathcal{R}$ 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

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

such that for all $1 \leq i < h$ we have $(p_i, T_i) \to (p_{i+1}, T_{i+1})$ is a transition of G. We write $(p, T) \to^* (p', T')$ 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 $(p_{src}, T_{src}) \to^* (p_{snk}, T)$ of G for some tree T.*

DEFINITION 3.6 (Regular Reachability Problem). *For 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 $(p_{src}, T_{src}) \to^* (p_{snk}, T)$ for some $T \in \mathcal{L}(\mathcal{T})$.*

3.3.2 Output Symbols

As part of the proofs, we are interested in sGTRSs whose transitions are labelled with output symbols. Hence, runs of an sGTRS produce words over the output alphabet.

DEFINITION 3.7 (GTRSs 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* $\mathcal{E} \notin \mathcal{L}(\mathcal{T}_1) \cup \mathcal{L}(\mathcal{T}_2)$.

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 $\gamma_1 \dots \gamma_h$. 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, 19]. 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 $(p_1, \mathcal{T}_1) \to (p_2, \mathcal{T}_2)$) 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*

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

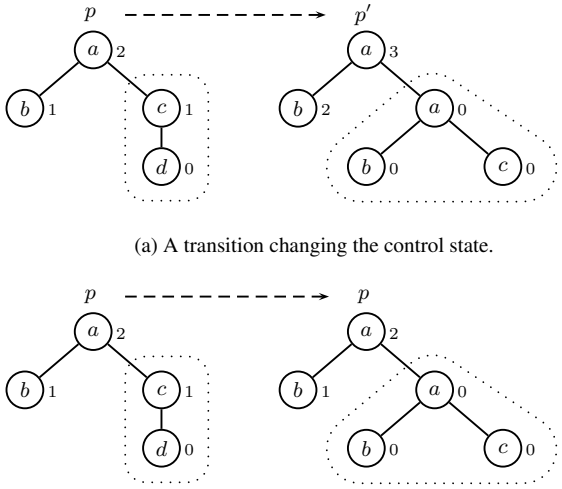
with $p_1 = p_h$ *satisfy* $p_i = p_1$ *for all* $1 \leq i \leq h$.

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 can then build a semilinear set (via Pottier [34] then Haase [20] or Piskac [33]) of a bounded size. From this we can acquire a regular automaton representing the possible outputs of weakly extended sGTRSs, which will be used later in our decidability proofs. More details appear in the arXiv article [21]. 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.



(a) A transition changing the control state.

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

Figure 1: Transitions of a senescent GTRS.

4. Senescent Ground Tree Rewrite Systems with State

We generalise weakly-synchronised sGTRS to define senescent ground tree rewrite systems, incorporating ideas from scope-bounded multi-stack pushdown systems [24]. Intuitively, when the control state changes, the nodes in the tree "age" by one timestep. When the nodes reach a certain (fixed) age they may no longer be changed by any future transitions. We give an informal example of this process below, and give a formal definition in the following sections. A full example is given in section 5

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.

4.1 Model Definition

Given a run

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

of an sGTRS, let $C_1, \dots, C_{h-1}$ 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 $T_i = C_i\left[T_i^{\text{out}}\right]$ and $T_{i+1} = C_i\left[T_{i+1}^{\text{in}}\right]$ where $(p_i, \mathcal{T}_i) \to (p_{i+1}, \mathcal{T}_i')$ 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 $C_j\left[T_j^{\text{in}}\right]$ 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, described above.

A lifespan restricted run with a lifespan of k is a run such that each transition $\left(p_i, C_i\left[T_i^{\text{out}}\right]\right) \to \left(p_{i+1}, C_i\left[T_{i+1}^{\text{in}}\right]\right)$ has the property that all nodes n in T_i^{out} have an age of at most k. That is,

more precisely, that all nodes n in the domain of $C_i\left[T_i^{\text{out}}\right]$ 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). *We define a* senescent ground tree rewrite system *with* lifespan k *to be an sGTRS* $G = (\mathcal{P}, \Sigma, \mathcal{R})$ *where runs are* lifespan restricted *with a lifespan of* k.

We will study the control state reachability problem and the regular reachability problem for senescent GTRS. These problems are defined analogously to the same problems for sGTRS (with the condition that runs are lifespan restricted). For completeness, we include the precise definition in the arXiv article [21]. We will show in Theorem 7.1 that the control state reachability problem is $\mathbf{F}_\omega$-complete, and in Theorem 6.3 that the regular reachability is undecidable.

One might expect that decidability of control state reachability would imply decidability of regular reachability, by, e.g., encoding the tree automaton into the senescent GTRS. However, one cannot 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.

5. Example

We give a brief example of a simplistic network where large tasks may be sent to a (powerful) processing unit. By abuse of notation, let $(p_1, T_1) \to (p_2, T_2)$ for control states p_1, p_2 and trees T_1, T_2 denote a rule $(p_1, \mathcal{T}_1) \to (p_2, \mathcal{T}_2)$ where $\mathcal{L}(\mathcal{T}_1) = \{T_1\}$ and $\mathcal{L}(\mathcal{T}_2) = \{T_2\}$.

The initial control state will be w indicating the processing unit is waiting. The initial tree will be $\bullet(i, p)$ where i represents a process and p the powerful processing unit and $\bullet$ is just an internal node. The following rules allow a process to either fork, or send a job to the processing unit (via the control state) and finish (by changing its label to j).

$$(w, i) \to (w, \bullet(i, i)) \text{ and } (w, i) \to (t, j)$$

The processing unit can pick up the task t, setting the control state to b to indicate the unit is busy, and then finish the task with the rules

$$(t, p) \to (b, t) \text{ and } (b, t) \to (w, p) \ .$$

Finally, a provision is made for processes to fork a local thread to do their processing if the powerful CPU is busy. The second rule joins the local thread after the processing is done.

$$(b, i) \to (b, \bullet(j, t)) \text{ and } (b, \bullet(j, t)) \to (b, j)$$

A simple correctness condition would be that the control state does not indicate that the powerful CPU is waiting while it is still processing. This is then a regular reachability property: can we reach a configuration with control state w and a tree belonging to the regular set of trees of the form $\bullet(*, t)$ (where $*$ is any tree). Unfortunately, the above system fails this criterion via the run shown in Figure 2. In this run, the changed parts of the tree are highlighted. An erroneous state is reachable because the programmer did not differentiate between a local task and a task running on the powerful CPU. Hence the local task at the bottom of the tree is able to set the control state to idle. Note, since the oldest node rewritten by a rule has age 2, a lifespan of 2 would be required for this run.

In this case, it is also possible to use control state reachability to detect if an error has occurred. To do so, we need to alter the initial tree slightly to $\bullet(i, x(p))$ where x is a marker node allowing us to identify the powerful CPU node. Then, we can use a rule

$$(w, x(t)) \to (e, x(t))$$

to allow the final configuration in Figure 2 (with the additional marker node) to reach the "error" control state e.

6. Modelling Power of Senescent GTRSs

We show in this section that senescent GTRSs at least capture scope-bounded multi-stack pushdown systems. We also show, to obtain our lower bounds, that we can encode coverability and reachability of a reset Petri net, via reductions to control state and regular reachability respectively.

6.1 Scope-Bounded Pushdown Systems

Senescent GTRS can naturally model scope-bounded multi-stack pushdown systems, which were first introduced by La Torre and Napoli [24] and shown to have a PSPACE-complete reachability problem. We first describe scope-bounded pushdown systems before comparing them with senescent GTRS. For spaces reasons, the discussion here will be informal with formal definitions appearing in the arXiv article [21].

6.1.1 Model

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. A transition can change the control state and may optionally update one of the stacks, either by pushing a character onto a stack or popping a character from it.

Scope-bounded pushdown systems restrict the behaviour of a multi-stack pushdown system. Runs 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: a character may only be removed if it was pushed within the previous k rounds. The control state reachability problem then asks, whether, from a given initial configuration, a target control state be reached.

6.1.2 Reduction to Senescent GTRS

The reduction to senescent GTRS 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 $\perp$ 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 $w = a_1 \dots a_m$ (with a_1 being the top of the stack) we write $w(T)$ for the tree $a_m(\cdots a_1(T))$.

A configuration of a single-stack pushdown system (p, w) can be 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. The pushdown system's control state acts 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 model the scheduler moving execution to the next stack, the control state of the senescent GTRS will be used to transfer the pushdown system's control state to the leaf of the next active branch. Thus, a k-scope-bounded multi-stack pushdown system will be modelled by a senescent GTRS with a lifespan of

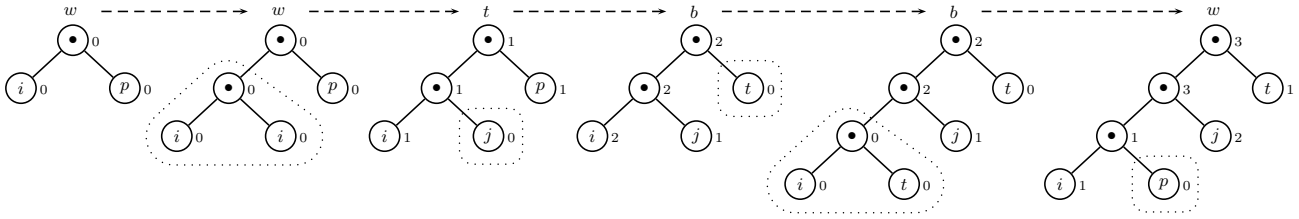


Figure 2: A bad run of the example GTRS.

$k \cdot z$. This is natural since a round of a scope bounded pushdown system contains z communications, and hence k rounds contain $k \cdot z$ communications.

For example, a run where control changes from the first to second stack during the final transition

$$(p_1, aw_1, w_2) \longrightarrow (p_2, w_1, w_2) \longrightarrow (p_3, w_1, bw_2)$$

can be modelled by the following run of a senescent GTRS where $\_$ indicates we are agnostic about the GTRS control state, and $\square_i$ indicates that stack i is not active.

$$\begin{aligned}(\_, w_1(a(p_1)), w_2(\square_2)) &\to (\_, w_1(p_2), w_2(\square_2)) \to \\ ((p_2, 2), w_1(\square_1), w_2(\square_2)) &\to (\_, w_1(\square_1), w_2(p_2)) \\ &\to (\_, w_1(\square_1), w_2(b(p_3)))\end{aligned}$$

The GTRS control state $(p_2, 2)$ indicates that stack 2 should become active with control state p_2. To check control state reachability, we can move the current pushdown control state to the control state of the GTRS at any time.

THEOREM 6.1 (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.*

We could extend scope-bounded pushdown systems to add dynamic thread creation by keeping a leaf node labelled $*$ which will be rewritten to $\bullet(a(p), *)$ when a new thread with stack a and control state p is created. Note, $\bullet$ is just an internal tree node, and we always have one leaf node labelled $*$.

6.2 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. We give informal reductions here and full details in the arXiv article [21].

The idea is that the control state of the reset Petri-net can be directly encoded by the control state of the senescent GTRS and 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 x to an "inactive" label $\wp$, while incrementing the counter requires adding a new leaf labelled x. To avoid leaf nodes becoming fossilised, we allow all leaves to rewrite to themselves in (almost) every control state. We can reset a counter x by forcing the GTRS to change control states k times without allowing any leaves labelled x to refresh; thus, all x nodes become fossilised and the counter is effectively set to zero. Since the control state is encoded directly, is it easy to check control state reachability.

THEOREM 6.2 (Coverability to Control State Reachability). *The coverability problem for reset Petri-nets can be reduced to the control state reachability problem for senescent GTRS.*

Using a slight extension of the coverability reduction, we can show that the reachability problem reduces to the regular reachability problem for GTRS. Naively, since leaf nodes 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. First, when a counter is being reset, we give all leaves labelled by that counter the opportunity to rewrite themselves to $\wp$. 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 leaves labelled by a counter x to rewrite themselves to $\overline{x}$. 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 became fossilised while labelled x (in particular, during a reset, all nodes rewrote themselves to $\wp$). Similarly, after guessing that the target configuration had been reached, all nodes labelled x changed to $\overline{x}$, thus ensuring that the tree accurately represents the true counter values.

THEOREM 6.3 (Reachability to Regular Reachability). *One can reduce the reachability problem for reset Petri-nets to the regular reachability problem for senescent GTRS, and thus regular reachability is undecidable.*

7. Reachability Analysis of Senescent GTRSs

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

THEOREM 7.1 (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 6.2 (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 7.4 a reset Petri-net $\mathcal{N}_G$ triply-exponential in the size of G. From Lemma 7.1 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. □

7.1 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, during the run, the complete tree rooted at n is never matched by the LHS of a rewrite rule, then 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 both. 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, 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 rule application. In the former case, but not the latter, a new independent sub-tree has been generated. 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 7.1 (Independent Sub-Tree Interfaces). *We define an* independent sub-tree interface α *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. The component b_i indicates whether the subtree made 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}. Note, if a rule both changes the control state and generates a new independent sub-tree, we say the sub-tree is generated *after* the control state changed.

7.2 Examples of Independent Sub-Tree Interfaces

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

$$r_1 = (p_1, T_0) \to (p_2, n(T_1, T_2)),$$
$$r_2 = (p_2, T_1) \to \big(p_2, T_1^1\big), \quad r_3 = (p_2, T_2) \to \big(p_3, T_2^1\big),$$
$$r_4 = \big(p_3, T_2^1\big) \to \big(p_4, T_2^2\big), \quad r_5 = \big(p_4, T_1^1\big) \to \big(p_5, T_1^2\big) .$$

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

$$(p_1, T_0) \to (p_2, n(T_1, T_2)) \to \big(p_2, n\big(T_1^1, T_2\big)\big) \to \big(p_3, n\big(T_1^1, T_2^1\big)\big) \to \big(p_4, n\big(T_1^1, T_2^2\big)\big) \to \big(p_5, n\big(T_1^2, T_2^2\big)\big) .$$

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

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

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 $\bigtriangleup$ to denote an independent sub-tree that has been generated, we can decompose the run into two runs

$$(p_1, T_0) \to \Big(p_2, \bigtriangleup\Big)$$

and the run of the generated independent sub-tree

$$(p_2, n(T_1, T_2)) \to \big(p_2, n\big(T_1^1, T_2\big)\big) \to \big(p_3, n\big(T_1^1, T_2^1\big)\big) \to \big(p_4, n\big(T_1^1, T_2^2\big)\big) \to \big(p_5, n\big(T_1^2, T_2^2\big)\big) .$$

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

$$\begin{array}{l}(p_1, 0, \vec{0}), (p_2, 1, \vec{1}) \\ \qquad (p_2, 0, \vec{0}), (p_3, 1, \vec{0}), (p_4, 1, \vec{0}), (p_5, 1, \vec{0})\end{array}$$

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.

$$(p_1, T_0) \to (p_2, \bigtriangleup), \text{ and}$$
$$(p_2, n(T_1, T_2)) \to (p_2, n(\bigtriangleup, T_2)) \to (p_3, n(\bigtriangleup, \bigtriangleup))$$

where the evolution of T_1^1 is given by

$$\big(p_2, T_1^1\big) \to \big(p_3, T_1^1\big) \to \big(p_4, T_1^1\big) \to \big(p_5, T_1^2\big)$$

and the evolution of T_2^1 by

$$\big(p_3, T_2^1\big) \to \big(p_4, T_2^2\big) \to \big(p_5, T_2^2\big) .$$

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 are

$$\begin{array}{l}(p_1, 0, \vec{0}), (p_2, 1, \vec{1}) \\ \qquad (p_2, 0, \vec{2}), (p_3, 1, \vec{4}) \\ \qquad (p_2, 0, \vec{0}), (p_3, 0, \vec{0}), (p_4, 0, \vec{0}), (p_5, 1, \vec{0}) \\ \qquad\qquad (p_3, 0, \vec{0}), (p_4, 1, \vec{0}), (p_5, 0, \vec{0}) .\end{array}$$

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.

7.3 Representing Interfaces

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

$$(p, \mathcal{T}) \to (p_1, \mathcal{T}_1) \in \mathcal{R}$$

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 $w \in \Gamma_I^*$, 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 naturally generalises to $\textsc{Parikh}(\mathcal{A})$. 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

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

of a run beginning with a tree $T \in \mathcal{L}(\mathcal{T}_1)$.

We obtain the above regular automaton as follows. First, from $G, r \in \mathcal{R}$ 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.

In the definition below, we use $\diamond$ to be the starting label of G_I, and the first type of rule is the rule generating a (independent sub-)tree that could have been created by rule r. The second type of rules simply simulate the rules of G that do not change the control state. The next two types 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 types 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 $\mathcal{T}_{\bigtriangleup}$) is created. Note, since $\bigtriangleup$ is a new label, the place-holder sub-tree cannot be rewritten during the remainder a run of G_I.

DEFINITION 7.2 (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,*

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

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

- $((p_1, b_1, 1), \mathcal{T}_\diamond) \xrightarrow{\varepsilon} ((p_1, b_1, 1), \mathcal{T})$, *and*
- $((p_i, b_i, i), \mathcal{T}_1) \xrightarrow{\varepsilon} ((p_i, b_i, i), \mathcal{T}_2)$ *when* $1 \leq i \leq m$ *and* $(p_i, \mathcal{T}_1) \to (p_i, \mathcal{T}_2) \in \mathcal{R}$, *and*
- $((p_i, b_i, i), \mathcal{T}_1) \xrightarrow{\varepsilon} ((p_{i+1}, 1, i+1), \mathcal{T}_2)$ *when* $1 \leq i < m$, $b_{i+1} = 1$ *and* $(p_i, \mathcal{T}_1) \to (p_{i+1}, \mathcal{T}_2) \in \mathcal{R}$, *and*
- $((p_i, b_i, i), \mathcal{T}_a) \xrightarrow{\varepsilon} ((p_{i+1}, 0, i+1), \mathcal{T}_a)$ *when* $1 \leq i < m$, $b_{i+1} = 0$ *and* $a \in \Sigma_I$ *and* a *has arity* 0, *and*
- $((p_i, b_i, i), \mathcal{T}_1) \xrightarrow{(r,i)} \left((p_i, b_i, i), \mathcal{T}_{\bigtriangleup}\right)$ *when* $1 \leq i \leq m$ *and* $r = (p_i, \mathcal{T}_1) \to (p_i, \mathcal{T}_2) \in \mathcal{R}$, *and*
- $((p_i, b_i, i), \mathcal{T}_1) \xrightarrow{(r,i+1)} \left((p_{i+1}, 1, i+1), \mathcal{T}_{\bigtriangleup}\right)$ *when* $1 \leq i < m$, $b_{i+1} = 1$ *and* $r = (p_i, \mathcal{T}_1) \to (p_{i+1}, \mathcal{T}_2) \in \mathcal{R}$,

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

Using G_I we build a regular representation of the independent sub-trees generated during a run with a given interface.

DEFINITION 7.3 ($\mathcal{A}_I$). *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 if and only if $\textsc{Parikh}(w) \in \textsc{Parikh}(\mathcal{A}_I)$.

7.4 Reduction to Reset Petri-Nets

7.4.1 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

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

Let $\pi : X_G \to \mathbb{N}$ be a valuation of the counters. We will refer to a tuple

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

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 an independent sub-tree with interface

$$\left(p_1', b_1', \vec{\eta}_1\right) \ldots \left(p_{m'}', b_{m'}', \vec{\eta}_{m'}\right) .$$

We require $(p_1, b_1) \ldots (p_m, b_m)$ 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, i.e.

$$(p_1, b_1), \ldots, (p_m, b_m) \;{+}{+}\; \left(p_1', b_1'\right) \ldots \left(p_{m'}', b_{m'}'\right)$$

when the two are compatible to be, when $m \leq m'$,

$$\left(p_1, b_1''\right) \ldots \left(p_m, b_m''\right) \left(p_{m+1}', b_{m+1}'\right) \ldots \left(p_{m'}', b_{m'}'\right)$$

and when $m > m'$,

$$\left(p_1, b_1''\right) \ldots \left(p_{m'}, b_{m'}''\right) (p_{m'+1}, b_{m'+1}) \ldots (p_m, b_m)$$

where $b_i'' = 1$ if $b_i = 1$ or $b_i' = 1$, and otherwise $b_i'' = 0$.

Then, the addition,

$$(\beta, \pi) \;{+}{+}\; \left(p_1', b_1', \vec{\eta}_1\right) \ldots \left(p_{m'}', b_{m'}', \vec{\eta}_{m'}\right)$$

when the two are compatible is (β', π') where

$$\beta' = \beta \;{+}{+}\; \left(p_1', b_1'\right) \ldots \left(p_{m'}', b_{m'}\right)$$

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 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. 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

$$\begin{aligned} \textsc{Res}(((p_1, b_1), (p_2, b_2), \ldots, (p_m, b_m), \pi)) \\ = ((p_2, b_2), \ldots, (p_m, b_m), \pi') \end{aligned}$$

when $b_2 = 1$ and where

$$\pi'(x_i^r) = \begin{cases} \pi(x_{i+1}^r) & 1 \leq i < m \\ 0 & i = m \,. \end{cases}$$

7.4.2 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 $r_1 = (p_{src}, \mathcal{T}_1) \to (p_{src}, \mathcal{T}_2)$ 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 T_{src}.

Addition of New Interfaces The Petri net can simulate 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 $(p_1, b_1) \ldots (p_m, b_m)$. 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 7.3 ($\mathcal{A}_I$), 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

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

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

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

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

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

$$\pi'(x_i^r) = \begin{cases} \pi(x_{i+1}^r) & 1 \leq i < m \\ 0 & i = m \,. \end{cases}$$

It does this incrementally from $i = 1$ to $i = m$. For each given i, the first step is to use 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 $\beta = (p_1, b_1) \ldots (p_m, b_m)$ 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 7.3 ($\mathcal{A}_I$) 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 7.4 ($\mathcal{N}_G$). *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*

$$\mathcal{S} = \{(p_1, b_1) \ldots (p_m, b_m) \in (\mathcal{P} \times \{0, 1\})^m \mid 1 \leq m \leq k\}$$

$$\begin{aligned} \mathcal{Q}_G = \mathcal{S} \cup \left\{(\beta, q) \in \mathcal{S} \times \mathcal{Q}_{\beta'}^r \mid \beta' \in \mathcal{S} \wedge r \in \mathcal{R}\right\} \cup \\ \left\{\lhd_i^\beta \mid \beta \in \mathcal{S} \wedge 1 \leq i \leq k\right\} \end{aligned}$$

and $\Delta_G = \Delta_{\textsc{Add}} \cup \Delta_{\textsc{Res}}$ where $\Delta_{\textsc{Add}}$ is the set

$$\begin{aligned} &\left\{\beta \xrightarrow[decr(x_1^r)]{} (\beta_1, q_{\beta_2}^r) \;\middle|\; \begin{array}{c} r \in \mathcal{R} \wedge \beta, \beta_1, \beta_2 \in \mathcal{S} \wedge \\ \beta_1 = \beta \mathbin{+\!\!+} \beta_2 \end{array}\right\} \cup \\ &\left\{(\beta, q) \xrightarrow[incr(x_i^r)]{} (\beta, q') \;\middle|\; \begin{array}{c} \beta \in \mathcal{S} \wedge \exists r', \beta' \text{ s.t.} \\ q \xrightarrow{(r,i)} q' \in \Delta_{\beta'}^{r'} \end{array}\right\} \cup \\ &\left\{(\beta, f_{\beta'}^r) \underset{\emptyset}{\to} \beta \mid r \in \mathcal{R} \wedge \beta, \beta' \in \mathcal{S}\right\} \end{aligned}$$

and $\Delta_{\textsc{Res}}$ is the set

$$\begin{aligned} &\left\{\beta \underset{\widetilde{o}}{\to} \lhd_1^{\beta'} \;\middle|\; \begin{array}{c} \beta, \beta' \in \mathcal{S} \wedge \beta = (p_1, b_1)\, \beta' \wedge \\ \beta' = (p_2, b_2) \ldots (p_m, b_m) \wedge \\ b_2 = 1 \wedge \widetilde{o} = \{res(x_1^r) \mid r \in \mathcal{R}\} \end{array}\right\} \cup \\ &\left\{\lhd_i^\beta \xrightarrow[\{decr(x_{i+1}^r), incr(x_i^r)\}]{} \lhd_i^\beta \mid \beta \in \mathcal{S} \wedge 1 \leq i < k\right\} \cup \\ &\left\{\lhd_i^\beta \underset{\widetilde{o}}{\to} \lhd_{i+1}^\beta \;\middle|\; \begin{array}{c} \beta \in \mathcal{S} \wedge 1 \leq i < k \wedge \\ \widetilde{o} = \{res(x_{i+1}^r) \mid r \in \mathcal{R}\} \end{array}\right\} \cup \\ &\left\{\lhd_k^\beta \underset{\emptyset}{\to} \beta \mid \beta \in \mathcal{S}\right\} \,. \end{aligned}$$

Note that the size of $\mathcal{N}_G$ is dominated by the size of the regular automata $\mathcal{A}_\beta^r$. Thus, the size of $\mathcal{N}_G$ is triply exponential in the size of G. The following (correctness) lemma is proved in the arXiv article [21].

LEMMA 7.1 (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 $(p_{src}, T_{src}) \rightarrow \cdots \rightarrow (p_{snk}, T)$ 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 $\pi_0 \leq \pi$.*

8. Conclusion

We introduced a sub-class of ground tree rewrite systems with state inspired by scope-bounded pushdown systems. We showed that control state reachability is inter-reducible with coverability of reset Petri-nets, and is thus $\mathbf{F}_\omega$-complete. This is a surprising increase in complexity compared to scope-bounded pushdown systems, for which reachability is PSPACE-complete. Thus, we obtain a natural model that captures a rich class of behaviours while maintaining decidability. 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 encode additional classes of multi-stack pushdown systems (e.g. ordered, phased-bounded, relaxed notions of scope-bounding, dynamic thread creation) into senescent GTRS, which may lead to generalisations of our model. Furthermore, tools such as FAST [8] and TREX [2] show that high (even undecidable) complexities do not preclude successful model checkers. We would like to study practical verification algorithms, which may use the aforementioned tools as components.

Acknowledgments

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*, 2002.
[2] A. Annichini, A. Bouajjani, and M. Sighireanu. Trex: A tool for reachability analysis of complex systems. In *CAV*, 2001.
[3] T. Araki and T. Kasami. Some Decision Problems Related to the Reachability Problem for Petri Nets. *TCS*, 3(1), 1977.
[4] M. F. Atig, B. Bollig, and P. Habermehl. Emptiness of multi-pushdown automata is 2etime-complete. In *DLT*, 2008.
[5] M. F. Atig, A. Bouajjani, and S. Qadeer. Context-bounded analysis for concurrent programs with dynamic creation of threads. *LMCS*, 7(4), 2011.
[6] T. Ball, V. Levin, and S. K. Rajamani. A decade of software model checking with slam. *Commun. ACM*, 54(7), 2011.
[7] T. Ball and S. K. Rajamani. Bebop: A symbolic model checker for boolean programs. In *SPIN*, 2000.
[8] S. Bardin, A. Finkel, J. Leroux, and L. Petrucci. Fast: acceleration from theory to practice. *STTT*, 10(5), 2008.
[9] A. Bouajjani, J. Esparza, and O. Maler. Reachability analysis of pushdown automata: Application to model-checking. In *CONCUR*, 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), 2009.
[11] W. S. Brainerd. Tree generating regular systems. *Information and Control*, 14(2), 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), 1996.
[13] A. Cyriac, P. Gastin, and K. N. Kumar. MSO decidability of multi-pushdown systems via split-width. In *CONCUR*, 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), 1990.
[15] M. Dauchet and S. Tison. The theory of ground rewrite systems is decidable. In *LICS*, 1990.
[16] J. Esparza, A. Kucera, and S. Schwoon. Model checking ltl with regular valuations for pushdown systems. *Inf. Comput.*, 186(2), 2003.
[17] A. Finkel, B. Willems, and P. Wolper. A direct symbolic approach to model checking pushdown systems. In *INFINITY*, 1997.
[18] S. Ginsburg and E. H. Spanier. Semigroups, presburger formulas, and languages. *Pacific Journal of Mathematics*, 16, 1966.
[19] S. Göller and A. W. Lin. Refining the process rewrite systems hierarchy via ground tree rewrite systems. In *CONCUR*, 2011.
[20] C. Haase. Subclasses or presburger arithmetic and the weak exponential-time hierarchy. Under submission.
[21] Matthew Hague. Senescent ground tree rewrite systems, 2013. arXiv:1311.4915 [cs.FL].
[22] N. D. Jones and S. S. Muchnick. Even simple programs are hard to analyze. *J. ACM*, 24(2), 1977.
[23] R. M. Karp and R. E. Miller. Parallel program schemata: A mathematical model for parallel computation. In *SWAT (FOCS)*, 1967.
[24] S. La Torre and M. Napoli. Reachability of multistack pushdown systems with scope-bounded matching relations. In *CONCUR*, 2011.
[25] S. La Torre and M. Napoli. A temporal logic for multi-threaded programs. In *IFIP TCS*, 2012.
[26] S. La Torre and G. Parlato. Scope-bounded multistack pushdown systems: Fixed-point, sequentialization, and tree-width. In *FSTTCS*, 2012.
[27] A. W. Lin. Weakly-synchronized ground tree rewriting - (with applications to verifying multithreaded programs). In *MFCS*, 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*, 2011.
[30] M. Maidl. The common fragment of ctl and ltl. In *FOCS*, 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, 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*, 1991.
[35] S. Qadeer. The case for context-bounded verification of concurrent programs. In *SPIN*, 2008.
[36] R. Büchi. Regular canonical systems. *Archiv fur Math. Logik und Grundlagenforschung 6*, 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), 2005.
[38] S. Qadeer and J. Rehof. Context-bounded model checking of concurrent software. In *TACAS*, 2005.
[39] Sylvain Schmitz. Complexity hierarchies beyond elementary, 2013. arXiv:1312.5686 [cs.CC].
[40] P. Schnoebelen. Verifying lossy channel systems has nonprimitive recursive complexity. *Inf. Process. Lett.*, 83(5), 2002.
[41] P. Schnoebelen. Revisiting ackermann-hardness for lossy counter machines and reset petri nets. In *MFCS*, 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*, 2007.