Reachability Analysis of Reversal-bounded Automata on Series-Parallel Graphs

Rayna Dimitrova
Max Planck Institute for Software Systems (MPI-SWS), Germany

Rupak Majumdar
Max Planck Institute for Software Systems (MPI-SWS), Germany

Extensions to finite-state automata on strings, such as multi-head automata or multi-counter automata, have been successfully used to encode many infinite-state non-regular verification problems. In this paper, we consider a generalization of automata-theoretic infinite-state verification from strings to labeled *series-parallel graphs*. We define a model of non-deterministic, 2-way, concurrent automata working on series-parallel graphs and communicating through shared registers on the nodes of the graph. We consider the following verification problem: given a family of series-parallel graphs described by a context-free graph transformation system (GTS), and a concurrent automaton over series-parallel graphs, is some graph generated by the GTS accepted by the automaton? The general problem is undecidable already for (one-way) multi-head automata over strings. We show that a *bounded* version, where the automata make a fixed number of reversals along the graph and use a fixed number of shared registers is decidable, even though there is no bound on the sizes of series-parallel graphs generated by the GTS. Our decidability result is based on establishing that the number of context switches is bounded and on an encoding of the computation of bounded concurrent automata to reduce the emptiness problem to the emptiness problem for pushdown automata.

1 Introduction

The language-theoretic approach to verification models the behaviors of a system as a set —or a *language*— of structures (such as strings or trees), and defines machine models that *generate* or *accept* these languages. The verification problem reduces to the language-emptiness problem for these models. The simplest such models are finite-state machines over finite or infinite words or trees, and this forms the basis of the hugely successful *automata-theoretic approach* to (finite-state) model checking [20]. Finite state machines have been generalized in many ways to extend the set of languages that may be needed to model more complex (non-regular) computational processes. For example, they can be extended with data structures such as stacks or counters, or with multiple heads or tapes and allowing 2-way traversals of the input [19, 17],

Since the emptiness problem can be undecidable for many extensions, research in infinite-state verification has focused on finding suitable underapproximations for which language emptiness is algorithmically decidable. For example, the *reversal boundedness* restriction bounds the number of reversals of the counters or of stacks, or the number of traversals of the input [13, 14, 11, 10] and the *bounded language* restriction considers behaviors describable by a bounded language [9, 8]. Overall, the approach has led to beautiful theoretical results and has also been quite successful in modeling many infinite-state parameterized computational models and reasoning about them algorithmically.

Most previous work in parameterized verification has focused on machine models for string or tree languages. In this paper, we study behaviors encoded as *series-parallel* graphs whose edges are labeled with a finite alphabet. Series-parallel graphs generalize strings or multi-tape machines by allowing multiple parallel "tracks" to fork off and rejoin at any point. They allow modeling various natural modes of computation, e.g., fork-join parallelism in data-parallel programs, while retaining enough structure, e.g., having a natural "forward" direction, that is absent in general graphs. Languages over series-parallel graphs can be naturally described using context-free graph transformation systems (GTSs), which describe the dynamic evolution of families of graphs through local rewrite rules [5, 2, 4].

We define and study a class of concurrent finite-state automata traversing series-parallel graphs and communicating through state-holding registers located at the nodes of the graph. More precisely, in our model of computation, a fixed number of finite-state machines traverse the nodes of a series-parallel graph. At each step, one of the machines makes a transition that depends on the current state of the machine, the label it reads on one of the incoming or outgoing arcs, and the value of the register stored at its node. The machine moves along the selected edge, updating its state as well as the register. Machines are thus 2-way and non-deterministic, and communicate through the shared registers. A series-parallel graph is accepted if some subset of machines reaches some final states being at the same node of the graph.

We study the emptiness problem: given a context-free GTS defining a language of series-parallel graphs, and a concurrent finite-state automaton, check if there is a graph in the language of the GTS accepted by the automaton. This problem is, not surprisingly, undecidable: for example, we can encode linear bounded automata over strings. We study a natural restriction of the emptiness problem by restricting the number of reversals along the computation and by putting a bound on the number of shared registers in the graph. With these two restrictions, we show that the emptiness problem is decidable and can be reduced to the emptiness problem for pushdown automata. Note that even with the restrictions, the problem is infinite-state because there is no a priori bound on the size of the series-parallel graphs generated by the GTS.

The reduction is based on two technical observations. First, when the number of reversals and the number of registers are fixed, there is a bound on the number of parallel tracks in the graph that needs to be tracked. We also establish a bound on the number of different times each machine moves along the run (although the length of the run may be unbounded). Second, using the bounds above, we construct a large alphabet that tracks valid runs of the machines on a valid graph generated by the GTS. We do this in several steps. We construct a pushdown automaton that checks that a word is a valid representation of a subgraph of a graph generated by the context-free GTS. We construct a set of automata, one for each machine, that checks that the word encodes a correct run of that machine along the graph. Finally, we construct another automaton that checks that the run is accepted by the concurrent automaton. Some graph generated by the GTS is accepted if the intersection of all these automata is non-empty.

Other Related Work The automata-theoretic approach is often called *regular* model checking, when applied to parameterized verification [1]. An extensive study of the decidability of several verification problems for classes of GTSs was carried out in [4]. The problems considered there are *reachability* of a given graph, *coverability* (reachability of a graph that contains a given graph as a subgraph) and *existential coverability*, which asks whether there exists an initial graph such that the answer to the coverability problem is positive. The classes of GTSs they investigate are defined by structural restrictions on the set of transformation rules. Classes with decidable coverability problem are context-free graph grammars, well-structured GTSs and the ones that keep the number of nodes constant. *Hyperedge-replacement graph grammars* [6] and *vertex-replacement graph grammars* [7] are well-studied classes of GTSs. It is known that for such graph grammars satisfiability of Monadic Second Order (MSO) formulas is decidable [5]. A logic for expressing properties that involve interleaving of temporal and graph modalities was developed in [3] as a combination of MSO and the μ-calculus. They employ an approximation of GTS [2] that preserves fragments of the logic to obtain a sound but incomplete verification method for these fragments. A method to refine such approximations based on counterexamples was developed in [15]. [18] describes a tool for model checking finite-state graph transition systems against first order temporal logic properties. The work [16] studies the emptiness problem for concurrent automata with auxiliary storage and provides a generalization of the decidability results for a number of classes of such automata for which the emptiness problem can be reduced to emptiness of finite-state graph automata defined MSO definable graphs with bounded tree width. It might be possible to obtain or generalize the results we establish in this paper through arguments similar to theirs.

2 Graph-grammar Transition Systems

Let A be a finite set. As usual, the set A^* consists of all finite sequences of elements of A. Let $\pi = a_0a_1\dots a_{n-1}a_n \in A^*$. We define $\pi^{-1} = a_na_{n-1}\dots a_1a_0$ and $\mathsf{last}(\pi) = a_n$. The length $|\pi| = n+1$ of π is the number of elements of π and given $0 \le i \le j \le n$ we denote $\pi[i] = a_i$ and $\pi[i,j] = a_i\dots a_j$.

With $\mathbb{M}(A) = \{S \mid S : A \to \mathbb{N}\}$ we denote the set of multisets over A. For $S_1, S_2 \in \mathbb{M}(A)$ we define $S_1 \preceq S_2$ iff for every $a \in A$ we have $S_1(a) \le S_2(a)$. We use square brackets to denote multisets, for example, $[a_1,a_2,a_2]$ denotes $S \in \mathbb{M}(A)$, where $S(a_1) = 1$, $S(a_2) = 2$ and $S(a) = 0$ for all $a \in A \setminus \{a_1,a_2\}$.

2.1 Series-parallel graph grammars

Fix an alphabet Σ. We consider graphs labeled with letters from Σ. A *graph* is a tuple $G = (N,E,n_b,n_e)$ where N is a finite set of nodes, $E \in \mathbb{M}(N \times N \times \Sigma)$ is a multiset of edges and $n_b, n_e \in N$ are two distinguished nodes called source and sink, respectively. For an edge $e = (n,n',\sigma) \in E$, we write $src(e)$ for n and $trg(e)$ for n', and $\alpha(e)$ for the label σ of e. We write $\mathscr{H}_\Sigma$ for the set of all Σ-labeled graphs.

Let $G = (N,E,n_b,n_e)$ and $G' = (N',E',n'_b,n'_e)$ be graphs on disjoint sets of nodes. For an edge $\widehat{e} = (\widehat{n}_1,\widehat{n}_2,\widehat{\sigma}) \in E$, the *edge replacement* graph $G[\widehat{e} \mapsto G']$ is the (unique up to isomorphism) graph defined by removing one copy of the edge $\widehat{e}$ from G, and adding the nodes and edges of G' by fusing $\widehat{n}_1$ with n'_b, and $\widehat{n}_2$ with n'_e. Formally, $G[\widehat{e} \mapsto G'] = (N'',E'',n_b,n_e)$, where $N'' = N \,\dot\cup\, (N' \setminus \{n'_b,n'_e\})$, $E'' = (E \setminus \{\widehat{e}\}) \cup \widehat{E}'$, where there is an edge (n_1,n_2,σ) in the multiset $\widehat{E}'$ with some multiplicity iff

- in E', with the same multiplicity, there is an edge (n'_b,n'_e,σ), and $n_1 = src(\widehat{e})$ and $n_2 = trg(\widehat{e})$, or
- in E', with the same multiplicity, there is an edge (n_1,n_2,σ), and $n_1 \neq n'_b$ and $n_2 \neq n'_e$, or
- in E', with the same multiplicity, there is an edge (n'_b,n_2,σ), and $n_1 = src(\widehat{e})$, or
- in E', with the same multiplicity, there is an edge (n_1,n'_e,σ), and $n_2 = trg(\widehat{e})$.

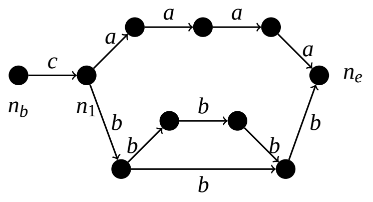
Definition 1 (Series-parallel graph grammar)**.** *A* series parallel graph grammar *(SPGG) is a tuple* $\mathscr{G} = (V,\Sigma,R,G_0)$, *where V is a finite set of* variables, *Σ is a finite alphabet ($\Sigma \cap V = \emptyset$), $R \subseteq V \times \mathscr{H}_{\Sigma\cup V}$ is a finite set of* rules, $G_0 = (\{n_b,n_e\},\{(n_b,n_e,v_0)\},n_b,n_e) \in \mathscr{H}_V$, *with $n_b \neq n_e$ is the* initial graph.

Furthermore, each rule $(v,G') \in R$, where $G' = (N',E',n'_b,n'_e)$, satisfies exactly one of the following:

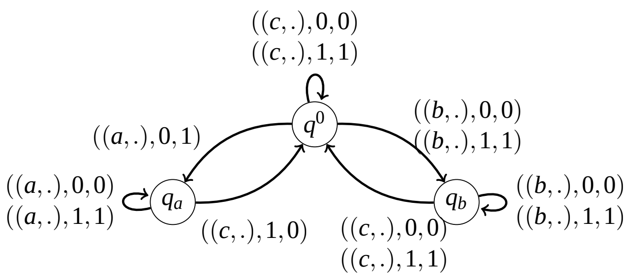
(1) $N' = \{n'_b,n'_e\}$, $E' = \{(n'_b,n'_e,\sigma)\}$ and $\sigma \in \Sigma$, denoted $(v,\sigma) \in R$;

(2) $N' = \{n'_b,n'_e,n'\}$ has three nodes, $E' = \{(n'_b,n',v_1),(n',n'_e,v_2)\}$ and $v_1,v_2 \in V$, denoted by $(v,v_1 \cdot v_2) \in R$ (series composition)*;*

(3) $N' = \{n'_b,n'_e\}$ has two nodes, $E' = \{(n'_b,n'_e,v_1),(n'_b,n'_e,v_2)\}$ and $v_1,v_2 \in V$, denoted by $(v,v_1 \parallel v_2) \in R$ (parallel composition).



(a) A series parallel graph with $\Sigma = \{a,b,c\}$.



(b) A finite-state machine with $\Sigma = \{a,b,c\}$.

Figure 1: A series parallel graph generated by an SPGG and a finite-state machine over the same alphabet.

An SPGG derives a graph in $\mathscr{H}_\Sigma$ as follows. It starts with the graph G_0. In each step, it picks an arbitrary edge e of the current graph G that is labeled with a variable $v \in V$, and applies a rule $(v, G') \in R$ to get a new graph $G'' = G[e \mapsto G']$. In this case, we write $G \Longrightarrow G''$. A graph $G \in \mathscr{H}_\Sigma$ is derived if there is a sequence $G_0 \Longrightarrow G_1 \ldots \Longrightarrow G_n = G$ of steps that results in G. Note that every graph thus derived is a series-parallel graph labeled with Σ, so an SPGG represents a set of series-parallel graphs labeled with Σ. We write $\mathscr{L}(\mathscr{G})$ for the set of graphs in $\mathscr{H}_\Sigma$ derived by $\mathscr{G}$.

Example 1. *As an example of an SPGG consider $\mathscr{G} = (V,\Sigma,R,G_0)$ with variables $V = \{v_0,v_1,v_a,v_b,v_c\}$, set of terminal symbols $\Sigma = \{a,b,c\}$, initial graph $G_0 = (\{n_b,n_e\},\{(n_b,n_e,v_0)\},n_b,n_e)$ and rules*

$$R \;=\; \{(v_0, v_c \cdot v_1),(v_1, v_a \parallel v_b),(v_a,a),(v_b,b),(v_c,c),(v_a, v_a \cdot v_a),(v_b, v_b \cdot v_b),(v_b, v_b \parallel v_b),\}.$$

Figure 1a shows a (series-parallel) graph G derived from the SPGG $\mathscr{G}$. The directions of the edges denote the "natural" direction from source to sink associated with a series parallel graph.

A series-parallel graph has a natural "direction" associated with it from the source to the sink, consistent with the direction $n_1 \to n_2$ of an edge (n_1,n_2,σ). In particular, it has no directed cycles. For convenience, we introduce the "symmetric closure" of series-parallel graphs. For each edge (n_1,n_2,σ) labeled with σ, we augment the label with a direction 1 to obtain $(\sigma,1)$ (1 capturing the "forward" direction), and add an opposite edge $(n_2,n_1,(\sigma,-1))$ labeled with $(\sigma,-1)$ denoting the edge taken in the "backward" direction. Formally, given a series-parallel graph $G = (N,E,n_b,n_e)$, we define its symmetric closure $G' = (N,E',n_b,n_e) \in \mathscr{H}_{\Sigma\times\{1,-1\}}$, where $E' = \{(n,n',(\sigma,1)) \mid (n,n',\sigma) \in E\} \cup \{(n,n',(\sigma,-1)) \mid (n',n,\sigma) \in E\}$. We write $\mathscr{L}^u(\mathscr{G})$ for the set of symmetric closures of all graphs derived by $\mathscr{G}$.

Remark. While for simplicity of the presentation we consider graphs with a single pair of source and sink nodes, our results can in principle be extended to graphs with multiple such nodes. However, the automata construction outlined in Section 4.3 relies on the structure of the rules of an SPGG and does not directly generalize to general context-free GTSs defining sets of directed acyclic graphs.

2.2 Graph-grammar transition systems

We now define communicating finite automata on the symmetric closure of series-parallel graphs. Recall that these are series-parallel graphs whose edges are labeled with an alphabet and a direction. Intuitively, a system of communicating machines has a set of m machines that traverse the edges of a series-parallel graph, some of whose nodes are annotated with Boolean registers. Each automaton traverses the edges of the graph: when the automaton is at a node n of the graph and in state q, it reads the register on the node, chooses an edge with source node n labeled with $(\sigma,d) \in \Sigma\times\{1,-1\}$ based on its current state, the label, and the value read from the register, traverses the edge and moves to the target node of that edge and to a new state q', and writes a value to the register at the source node.

Let Σ be a finite alphabet. A *finite-state machine* $\mathscr{M} = (Q, q^0, \Sigma, \delta)$ consists of finite set of states Q, initial state $q^0 \in Q$, input alphabet Σ, and transition relation $\delta \subseteq Q \times (\Sigma \times \{1,-1\}) \times \mathbb{B} \times Q \times \mathbb{B}$.

The intuitive meaning of a transition $(q,(\sigma,d),b,q',b') \in \delta$ is that when the machine $\mathscr{M}$ is in state q and reads input letter $\sigma \in \Sigma$, direction $\{1,-1\}$, and register value b, then it changes its state to q' and moves along an edge labeled (σ,d) in the graph and writes b' to the register.

Example 2. *Figure 1b shows an example of a finite-state machine $\mathscr{M} = (Q,q^0,\Sigma,\delta)$ with states $Q = \{q^0,q_a,q_b\}$, input alphabet $\Sigma = \{a,b,c\}$ and transition relation δ depicted in Figure 1b, where a label $(((\sigma,d),p,p')$ on an edge from state q to state q' stands for the transition $((q,(\sigma,d),p,q',p')$.*

A system of machines $(\mathscr{M},m)$ is a set of m disjoint copies $\mathscr{M}_1,\dots,\mathscr{M}_m$ of the machine $\mathscr{M}$.

Definition 2 (Graph-grammar transition system)**.** *Let $\mathscr{M} = (Q,q^0,\Sigma,\delta)$ be a finite-state machine. A system of m machines $(\mathscr{M},m)$, together with an SPGG $\mathscr{G} = (V,\Sigma,R,G_0)$ defines a* transition system *$T(\mathscr{M},m,\mathscr{G}) = (\Gamma,\Gamma_0,\rightarrow)$ as follows. The set of* configurations *Γ consists of all tuples $\langle G,\mu,\beta\rangle$ such that $G \in \mathscr{L}^u(\mathscr{G})$ is a graph derived by $\mathscr{G}$ and:*

- *$\mu : N \rightarrow 2^{\{1,\dots,m\}\times Q}$ maps each node in G to the states of the machines at that node; we require that for each $i \in \{1,\dots,m\}$ there exists exactly one $n \in N$ and exactly one $q \in Q$ with $(i,q) \in \mu(n)$;*
- *$\beta : N \rightarrow \mathbb{B}$ maps each node to the value of the Boolean register at that node.*

The set Γ_0 of initial configurations *is such that $\gamma = \langle G,\mu,\beta\rangle \in \Gamma_0$ iff $\gamma \in \Gamma$, $\mu(n_b) = \{(i,q^0) \mid i \in \{1,\dots,m\}\}$, $\mu(n) = \emptyset$ for every $n \in N \setminus \{n_b\}$, and $\beta(n) = 0$ for every $n \in N$. That is, initially all machines are positioned at the source node of the graph and are in their initial state, and all registers are* 0*.*

The successor relation *$\rightarrow \subseteq \Gamma \times \Gamma$ is defined as $\rightarrow = \bigcup_{i=1}^{m} \rightarrow_i$, where for each $i \in \{1,\dots,m\}$ it holds that $(\langle G,\mu,\beta\rangle,\langle G',\mu',\beta'\rangle) \in \rightarrow_i$, (denoted $\langle G,\mu,\beta\rangle \rightarrow_i \langle G',\mu',\beta'\rangle$) iff the following hold:*

- *$G' = G$, where $G = (N,E,n_b,n_e) \in \mathscr{L}^u(\mathscr{G})$ is a graph generated by $\mathscr{G}$.*
- *There exist an edge $e = (n,n',(\sigma,d)) \in E$, states $q,q' \in Q$, and a value $b' \in \mathbb{B}$ such that:*
  - *(i) $(i,q) \in \mu(n)$ (Note: $n \neq n'$, since $\mathscr{G}$ is an SPGG),*
  - *(ii) $(q,\alpha(e),\beta(n),q',b') \in \delta$,*
  - *(iii) $\mu'(n) = \mu(n) \setminus \{(i,q)\}$, $\mu'(n') = \mu(n) \cup \{(i,q')\}$ and $\mu'(n'') = \mu(n'')$ for all $n'' \in N \setminus \{n,n'\}$,*
  - *(iv) $\beta'(n) = b'$ and $\beta'(n'') = \beta(n'')$ for all $n'' \in N \setminus \{n\}$.*

  We say that the edge e is compatible *with the transition $\gamma \rightarrow \gamma'$.*

A *run* ρ of $T(\mathscr{M},m,\mathscr{G}) = (\Gamma,\Gamma_0,\rightarrow)$ is a sequence of configurations $\rho = \gamma_0 \dots \gamma_f \in \Gamma^*$ such that $\gamma_0 \in \Gamma_0$ and $\gamma_{i-1} \rightarrow \gamma_i$ for each $i = 1,\dots,f$.

Intuitively, the infinite-state transition system $T(\mathscr{M},m,\mathscr{G})$ captures the behaviors of m machines, copies of $\mathscr{M}$, on the family of all series-parallel graphs derived by $\mathscr{G}$.

2.3 Configuration properties and verification problem

A *configuration property* describes a set of configurations. Let n be a variable (ranging over nodes), and $S \in \mathbb{M}(Q)$. The set of *configuration properties* consists of the positive Boolean combinations (no negation) of *atomic* properties of the form $\exists n.\ S \preceq \mu(n)$.

A configuration $\gamma = \langle G,\mu,\beta\rangle \in \Gamma$ with $G = (N,E,n_b,n_e)$ satisfies an atomic configuration property $\varphi = \exists n.\ S \preceq \mu(n)$ (written $\gamma \models \varphi$) iff there exists a node $n \in N$ such that $S \preceq [q \in Q \mid (i,q) \in \mu(n)$, where $i \in \{1,\dots,m\}]$, that is, the multiset S is contained in the multiset of machine states in the node n in the configuration γ. The relation $\models$ is naturally extended to positive Boolean combinations.

Let $(\mathscr{M}, m)$ be a system of machines and $\mathscr{G}$ an SPGG. Given a configuration property F describing a set of *final configurations*, the *verification problem* $\mathsf{Reach}(\mathscr{M}, m, \mathscr{G}, F)$ is to decide whether there exists a run $\rho = \gamma_0 \dots \gamma_f$ of $T(\mathscr{M}, m, \mathscr{G})$ such that $\gamma_i \models F$ for some $0 \leq i \leq f$, i.e., a run that reaches F.

Since our model allows machines to do arbitrarily many "reversals" (i.e., following forward and backward edges) and do not fix a bound on the number of shared registers that are read or written, it easily captures linear bounded automata. Thus, the verification problem is in general undecidable.

Proposition 1. *The verification problem is undecidable.*

2.4 Bounded verification problem

Since the general problem is undecidable, we focus on a bounded version. We introduce two restrictions. First, we allow each machine to make only a bounded number of reversals (a reversal occurs when the machine changes direction in the graph). Second, we fix an a priori bound on the number of shared registers. That is, while the SPGG generates a potentially unbounded set of graphs, with unboundedly many nodes, we assume that there is some fixed bound k on the number of Boolean registers located at nodes of a generated graph (these k registers may be situated at arbitrary nodes of the graph though).

Fix a machine $\mathscr{M} = (Q, q^0, \Sigma, \delta)$, the system of machines $(\mathscr{M}, m)$, and an SPGG $\mathscr{G} = (V, \Sigma, R, G_0)$.

Reversal bound. Let us fix a run $\rho = \gamma_0, \dots, \gamma_f$ where $\gamma_i = \langle G, \mu_i, \beta_i \rangle$. Consider the projection of ρ to $\rightarrow_j$ for each machine $j \in \{1, \dots, m\}$. The number of *reversals* made by machine j along the run, intuitively, is the number of times it changes from traversing an edge marked with direction 1 to traversing an edge marked with direction -1, or vice versa.

Formally, let $e_1 e_2 \dots e_n$ be a sequence of edges. A reversal occurs at position i if $\alpha(e_i) = (\cdot, 1)$ and $\alpha(e_{i+1}) = (\cdot, -1)$ or if $\alpha(e_i) = (\cdot, -1)$ and $\alpha(e_{i+1}) = (\cdot, 1)$.

Now, let $\gamma_{i_1} \rightarrow_j \gamma_{i_1+1}$, $\gamma_{i_2} \rightarrow_j \gamma_{i_2+1}$, ... be the transitions of machine j along the run ρ, and let e_{i_1}, e_{i_2}, ... be the compatible edges that were taken by machine j. The number of reversals of machine j along ρ is the number of reversals in the sequence $e_{i_1} e_{i_2} \dots$.

For $r \geq 0$, the set of *r-reversal bounded* runs of $T(\mathscr{M}, m, \mathscr{G})$ is the set of runs in which each machine makes at most r reversals.

Register bound. The register bound fixes a number k of Boolean registers. That is, each graph G derived by $\mathscr{G}$ comes with a mapping $\kappa : N \rightarrow \{0, 1\}$, such that $|\kappa^{-1}(1)| \leq k$, and we allow the machines to read and write register values only when their current node is in $\kappa^{-1}(1)$.

To derive graphs with a mapping κ, we modify an SPGG to "mark" some nodes along the derivation, and ensure that any derived graph has at most k marked nodes. (The formal details are similar to constructing a CFG for a CFL with at most k marked positions from a CFG for the (unmarked) language.) For an SPGG $\mathscr{G}$, we denote by $\mathscr{G}^k$ the SPGG that marks at most k nodes of a derived graph. We write, by abuse of notation, $(G, \kappa) \in \mathscr{L}^u(\mathscr{G}^k)$ for a graph G which is the symmetric closure of a graph derived by $\mathscr{G}^k$ together with the mapping κ.

In addition, we modify the successor relation of the graph-grammar transition systems $T(\mathscr{M}, m, \mathscr{G}^k)$ to require (ii)' $(q, \alpha(e), \beta(n), q', b') \in \delta$ if $\kappa(n) = 1$ and $(q, \alpha(e), 0, q', 0) \in \delta$ otherwise.

Example 3. *The SPGG $\mathscr{G}$ shown in Example 1 can be modified into an SPGG $\mathscr{G}^2$ that derives graphs in which at most 2 nodes are marked. Furthermore, we can consider SPGGs that not only ensure an upper bound on the number of marked nodes, but impose constraints on their location. For example, we can* *consider an SPGG $\mathscr{G}_*^2$ that additionally requires that by applying the rule $(v_0, v_c \cdot v_1)$ the corresponding intermediate node between the edges labeled v_c and v_1 is marked to contain a register.*

If we then consider the graph-grammar transition system $T(\mathscr{M}, 2, \mathscr{G}_*^2)$, where $\mathscr{M}$ is the finite-state machine described in Example 2, and let G be the graph depicted in Figure 1a, then there does not exist a run with underlying graph G that reaches a configuration satisfying $\varphi = \exists n.\ [q_a, q_a] \preceq \mu(n)$, since the register at node n_1 acts as a semaphore that does not allow two copies of the machine $\mathscr{M}$ to enter the part of the graph containing edges labeled with the letter a.

The reversal-bounded and register-bounded verification problem takes as input a system of machines $(\mathscr{M}, m)$, an SPGG $\mathscr{G}$, and parameters r and k, and a configuration property F, and asks if there exists an r-reversal bounded run of the machines on some graph derived by $\mathscr{G}^k$ that reaches F.

Our main result is the following.

Theorem 1. *The reversal- and register-bounded verification problem is decidable.*

Remark. Our decidability results hold for a somewhat more general model, in which the machines can read one of the fixed number of registers not at its current node but can only write to the register at its current node, or vice versa. We work in the simpler setting to keep the notation manageable.

3 Properties of Reversal-Bounded Runs

Fix a machine $\mathscr{M} = (Q, q^0, \Sigma, \delta)$, the system of machines $(\mathscr{M}, m)$, an SPGG $\mathscr{G} = (V, \Sigma, R, G_0)$ and the parameters r and k. In this section we state two properties of r-reversal bounded runs of $T(\mathscr{M}, m, \mathscr{G}^k)$ that allow us to encode such runs as words over a finite alphabet and to reduce the reversal- and register-bounded verification problem to the emptiness test for a context free language.

Given a run $\rho = \gamma_0 \ldots \gamma_f$ and a machine $i \in \{1, \ldots, m\}$, an *i-block* is a segment $\rho[j_1, j_2] = \gamma_{j_1} \ldots \gamma_{j_2}$ of the run ρ such that $\gamma_j \rightarrow_i \gamma_{j+1}$ for each $j_1 \leq j < j_2$. That is, all transitions in the part $\rho[j_1, j_2]$ of the run are made by machine i. The following proposition establishes that for every r-reversal bounded run ρ we can reorder its transitions to obtain an r-reversal bounded run $\widehat{\rho}$ such that the number of maximal blocks in $\widehat{\rho}$ is not greater than a constant depending on m, r and k (and not on the length of the run ρ).

Proposition 2. *For every r-reversal bounded run $\rho = \gamma_0, \ldots, \gamma_f$ of $T(\mathscr{M}, m, \mathscr{G}^k)$ there exist an r-reversal bounded run $\widehat{\rho} = \widehat{\gamma}_0, \ldots, \widehat{\gamma}_f$ of $T(\mathscr{M}, m, \mathscr{G}^k)$ and a sequence of indices $0 = f_0 < f_1 < \ldots < f_u = f$ such that the following conditions are satisfied:*

- $u \leq \big(r \cdot m + k \cdot m \cdot (r+1) + 1\big) \cdot (m+1)$,
- *for each $i \in \{0, \ldots, u-1\}$, there exists $m_i \in \{1, \ldots, m\}$ such that $\widehat{\rho}[f_i, f_{i+1}]$ is a m_i-block,*
- *$\widehat{\gamma}_0 = \gamma_0$ and $\widehat{\mu}_f = \mu_f$, where $\gamma_f = \langle G, \mu_f, \beta_f \rangle$ and $\widehat{\gamma}_f = \langle G, \widehat{\mu}_f, \widehat{\beta}_f \rangle$.*

Remark. In the proof of the above proposition we construct the run $\widehat{\rho}$ by reordering transitions in ρ while keeping in place the transitions that access registers. Thus, the relative order of transitions which modify registers is preserved, which in turn implies that $\widehat{\gamma}_f = \gamma_f$ (that is, we have also $\widehat{\beta}_f = \beta_f$).

The second property uses the bound r on the number of reversals of each machine in an r-reversal bounded run ρ to relate ρ to the set of paths in the underlying graph traversed by the machines in ρ.

A *trace* τ is an element of the set Σ^*. A trace $\tau = \sigma_1 \ldots \sigma_f$ is *compatible* with a run $\rho = \gamma_0, \ldots, \gamma_f$ if there exists a sequence of edges $e_1 e_2 \ldots e_f$ compatible with ρ such that $\alpha(e_i) = (\sigma_i, \cdot)$ for every $0 < i \leq f$.

Given a graph $G = (N, E, n_b, n_e) \in \mathscr{L}^u(\mathscr{G}^k)$ and a trace τ we define $\mathsf{Paths}(G, \tau)$ to be the (possibly empty) set of paths from n_b to n_e whose sequence of edge labels is $\tau = \sigma_1 \ldots \sigma_f$. Formally, for a sequence of nodes $\pi = n_0 n_1 \ldots n_f \in N^*$ we have $\pi \in \mathsf{Paths}(G, \tau)$ iff $n_0 = n_b$, $n_f = n_e$ and $(n_{i-1}, n_i, (\sigma_i, 1)) \in E$.

Below we establish a property of an r-reversal bounded run $\rho = \gamma_0 \dots \gamma_f$ of $T(\mathscr{M}, m, \mathscr{G}^k)$ and a trace τ that is compatible with ρ. Namely, for each machine $i \in \{1, \dots, m\}$ the corresponding subsequence τ_i of τ can be split into at most $r+1$ segments, such that each of those segments can be embedded in a trace labelling a simple path from n_b to n_e or from n_e to n_b. This is formalized in the following proposition, which easily follows from the properties of series-parallel graphs.

Proposition 3. *Let ρ be an r-reversal bounded run of $T(\mathscr{M}, m, \mathscr{G}^k)$ and τ be a trace that is compatible with ρ. Let π_i be the sequence of nodes visited in ρ by machine $i \in \{1, \dots, m\}$, in the order they occur in ρ, let τ_i be the corresponding subsequence of τ, and $r_i \leq r$ be the number of reversals of machine i in ρ.*

Then, for each $i \in \{1, \dots, m\}$ and each $h \in \{1, \dots, r+1\}$ there exist traces $\tau_{i,h}, \tau'_{i,h}, \tau''_{i,h}, \tau'''_{i,h} \in \Sigma^*$ and sequences of nodes $\pi_{i,h}, \pi'_{i,h}, \pi''_{i,h}, \pi'''_{i,h} \in N^*$ such that the following conditions are satisfied:

- *$\pi_{i,h} \in \mathsf{Paths}(G, \tau_{i,h})$, and $\tau_{i,h} = \tau'_{i,h} \cdot \tau''_{i,h} \cdot \tau'''_{i,h}$, and $\pi_{i,h} = \pi'_{i,h} \cdot \pi''_{i,h} \cdot \pi'''_{i,h}$;*
- *For each $i \in \{1, \dots, m\}$ there exist indices $0 = j_0 < j_1 < \dots < j_{r_i+1} = |\pi_i| - 1$ such that:*
  - *if $1 \leq h \leq r_i + 1$ and h is odd, then $\tau_i[j_{h-1}+1, j_h] = \tau''_{i,h}$ and $\pi_i[j_{h-1}, j_h] = \pi''_{i,h}$;*
  - *if $1 \leq h \leq r_i + 1$ and h is even, then $\tau_i[j_{h-1}+1, j_h] = {\tau''_{i,h}}^{-1}$ $\pi_i[j_{h-1}, j_h] = {\pi''_{i,h}}^{-1}$.*

Proposition 2 allows us to restrict our reasoning to r-reversal bounded runs with at most $\big(r \cdot m + k \cdot m \cdot (r+1) + 1\big) \cdot (m+1)$ blocks. Proposition 3 allows us to reduce from reasoning about graphs derived by $\mathscr{G}^k$ to reasoning about $r+1$-tuples of traces in such graphs. Based on these results, we define the two parameters $p = \big(r \cdot m + k \cdot m \cdot (r+1) + 1\big) \cdot (m+1)$ and $t = \widetilde{r} \cdot m$, where $\widetilde{r} = r + 1$.

4 Automata-theoretic Algorithm

In this section we present an automata-theoretic algorithm for solving the reversal- and register-bounded verification problem. Before we give an overview of our algorithm and outline the automata constructions it comprises, we recall some basic definitions from automata theory.

4.1 Preliminaries

A 2-way nondeterministic finite automaton (2NFA) is a tuple $\mathscr{A} = (Q, q^0, \Sigma, \delta, A)$, where Q is a finite set of states, $q^0 \in Q$ is the initial state, Σ is a finite alphabet, $\delta \subseteq Q \times \Sigma \times Q \times \{-1, 1\}$ is the transition relation and $A \subseteq Q$ is a set of accepting states. $\mathscr{A}$ is deterministic iff δ is a function from $Q \times \Sigma$ to $Q \times \{-1, 1\}$. $\mathscr{A}$ is a 1-way NFA (NFA) iff $d = 1$ for each $(q, \sigma, q', d) \in \delta$.

For $q, q' \in Q$, $w', w'', w''', w'''' \in \Sigma^*$, $\sigma \in \Sigma$ and $\sigma' \in \Sigma \cup \{\varepsilon\}$, let $\langle q, w', \sigma, w'' \rangle \Rightarrow_{\mathscr{A}} \langle q', w''', \sigma', w'''' \rangle$ iff $(q, \sigma, q', d) \in \delta$ and (1) if $d = 1$, then $w''' = w'.\sigma$, $w'' = \sigma'.w''''$, either $\sigma' \in \Sigma$ or $\sigma' = \varepsilon$ and $w'''' = \varepsilon$ and (2) if $d = -1$, then $w'''' = \sigma.w''$, $w' = w'''.\sigma'$, either $\sigma' \in \Sigma$ or $\sigma' = \varepsilon$ and $w''' = \varepsilon$.

If $\mathscr{A}$ is an NFA, we define $\delta(q, w)$ for $w \in \Sigma^*$ in the obvious way.

Let $\vdash \in \Sigma$ and $\dashv \in \Sigma$, where $\vdash \neq \dashv$, be designated symbols and $w \in (\Sigma \setminus \{\vdash, \dashv\})^*$.

If $\mathscr{A}$ is a 2NFA, then $w \in \mathscr{L}(\mathscr{A})$ iff $\langle q^0, \varepsilon, \vdash, w \dashv \rangle \Rightarrow^*_{\mathscr{A}} \langle q, \vdash w \dashv, \varepsilon, \varepsilon \rangle$ or $\langle q^0, \varepsilon, \vdash, w \dashv \rangle \Rightarrow^*_{\mathscr{A}} \langle q, \varepsilon, \varepsilon, \vdash w \dashv \rangle$ for some $q \in A$. If $\mathscr{A}$ is an NFA, then $w \in \mathscr{L}(\mathscr{A})$ iff $\delta(q^0, \vdash w \dashv) \cap A \neq \emptyset$.

A push-down automaton (PDA) is a tuple $\mathscr{P} = (Q, q^0, \Sigma, \Delta, \bot, \delta)$, where Q is a finite set of states, $q^0 \in Q$ is the initial state, Σ is a finite input alphabet, Δ is a finite stack alphabet, $\bot$ is the start symbol and $\delta \subseteq Q \times (\Sigma \cup \{\varepsilon\}) \times \Delta \times Q \times \Delta^*$ is the transition relation. For $q, q' \in Q$, $\sigma \in \Sigma \cup \{\varepsilon\}$, $w \in \Sigma^*$, $a \in \Delta$, $\alpha, \beta \in \Delta^*$ we define $\langle q, \sigma.w, a.\alpha \rangle \Rightarrow_{\mathscr{P}} \langle q', w, \beta.\alpha \rangle$ iff $(q, \sigma, a, q', \beta) \in \delta$.

For a PDA $\mathscr{P}$, $w \in \mathscr{L}(\mathscr{P})$ iff $\langle q^0, \vdash w \dashv, \bot \rangle \Rightarrow^*_{\mathscr{P}} \langle q, \varepsilon, \bot \rangle$.

4.2 Overview of the algorithm

We now outline the construction of a PDA $\mathscr{A}$, which we use in order to reduce the reversal- and register-bounded verification problem to checking emptiness of a PDA. We begin by describing the input of the automata involved in the construction and then proceed to give an overview of the construction followed by a formal definition of the input alphabets of these automata.

The automaton $\mathscr{A}$ reads words that consist of traces in Σ^*. In order to reflect sufficient information about the corresponding nodes and registers in the underlying graph, these traces are annotated as follows. First, since graphs derived by $\mathscr{G}^k$ contain at most k registers, we assume these registers to have unique identifiers from the set $\{1,\dots,k\}$. Thus, a triple $(\sigma, j_1, j_2) \in \Sigma \times \{0,\dots,k\} \times \{0,\dots,k\}$ consists of an edge label σ and the identifiers of the registers at the source and target node of the edge, where 0 indicates no register at the respective node. We add additional annotation to reflect which nodes are shared in the corresponding paths, that is, positions where paths in the series-parallel graph branch off or join.

The automaton $\mathscr{A}$ reads such annotated traces and checks the existence of a run by emulating the behaviour of the machines on these traces by guessing an execution for each of them. An *execution* of $\mathscr{M} = (Q, q^0, \Sigma, \delta)$ is a sequence $\xi = q_0, (\sigma_1, b_1, b'_1), q_1, \dots, (\sigma_f, b_f, b'_f), q_f$ such that $(q_{l-1}, \sigma_l, b_l, q_l, d_l, b'_l) \in \delta$ for some $d_l \in \{1, -1\}$. In addition to verifying that each guess is indeed an execution, $\mathscr{A}$ needs to also check that the values written to and read from the shared registers by different machines are consistent.

Formally, an annotated trace and executions of the machines define a *read-write* sequence $\eta = (j_1, b_1, b'_1), \dots, (j_f, b_f, b'_f) \in (\{0,\dots,k\} \times \mathbb{B} \times \mathbb{B})^*$, where, intuitively, j_i is the location that is read and/or written. Such a read-write sequence η is *valid w.r.t. an initial register valuation* $\beta_0 : \{1,\dots,k\} \to \mathbb{B}$ iff each read operation reads the value written by the most recent write operation, or the initial value from β_0 if it is not overwritten, that is, for $i \in \{1,\dots,f\}$ with $j_i > 0$ it holds that if there is $i' < i$ such that $j_{i'} = j_i$, then $b_i = b'_{i'}$ for the largest such i', and otherwise $b_i = \beta_0(j_i)$.

Thus, the automaton $\mathscr{A}$ accepts tuples of traces in some graph derived by $\mathscr{G}^k$, annotated with information about registers and about nodes shared by the corresponding paths in the graph. $\mathscr{A}$ also guesses an execution for each of the m machines. The PDA $\mathscr{A}$ is constructed as the intersection of a PDA $\mathscr{P}_\mathsf{t}$ and an NFA $\mathscr{A}_\mathsf{e}$ (Section 4.3). $\mathscr{P}_\mathsf{t}$ checks that its input word encodes a tuple of traces in some graph derived by $\mathscr{G}^k$ and that these are correctly annotated with information about registers and the nodes that are shared among the paths corresponding to these traces. The NFA $\mathscr{A}_\mathsf{e}$ guesses and verifies the executions of the machines. It is obtained as the intersection of $m+2$ NFAs: m NFAs A_i, one for each $i \in \{1,\dots,m\}$, an NFA $\mathscr{A}_\mathsf{c}$ and an NFA $\mathscr{A}_\mathsf{s}$. The NFA $\mathscr{A}_i$ verifies that the guess of an execution of machine $i \in \{1,\dots,m\}$ is correct. We describe the construction of $\mathscr{A}_i$ as a 2NFA (Section 4.4) which is then converted to an NFA using standard techniques [12]. Automaton $\mathscr{A}_\mathsf{c}$ checks the validity of the read-write sequence corresponding to the annotated traces and the guessed executions (Section 4.5). Automaton $\mathscr{A}_\mathsf{s}$ (Section 4.6) checks that a configuration in F is reached. The reversal- and register-bounded verification problem thus reduces to checking emptiness of the language of the constructed automaton $\mathscr{A}$.

According to Section 3, it suffices to reason about $t = m \cdot (r+1)$ traces in graphs derived by $\mathscr{G}^k$. To this end, we define the *trace alphabet* $\Sigma_\mathsf{t} = \Big(\big(\Sigma \,\dot\cup\, \{\flat\}\big) \times \{0,\dots,k\}^2 \times \{1,\dots,t\}\Big)^t \,\dot\cup\, \{1,\dots,t\}^t$. Words over Σ_t are tuples of t traces in some graph G, annotated with additional information. Each letter in Σ_t contains one row for each of the m machines and each of the $\widetilde{r} = r+1$ paths corresponding to it. There are two types of letters. Each row in a letter of the first type consists of a letter in Σ (or the special symbol $\flat$) together with two *register identifiers* in $\{0,\dots,k\}$ and a *path index* in $\{1,\dots,t\}$. The letters of the second type are t-tuples of path indices in $\{1,\dots,t\}$, where equal indices indicate paths sharing a node.

The *execution alphabet* $\Sigma_\mathsf{e} = \big(\{0,\dots,p\} \times \mathbb{B} \times \mathbb{B} \times Q \times \{1,\dots,t\}\big)^t$ is used to describe tuples of executions, one for each of the m machines. Each letter contains $\widetilde{r} = r+1$ rows for each machine, one for each of its paths. Each row in the letter consists of a *block number* in $\{0,\dots,p\}$, two *register values* (one for the read and one for the write operations), a *successor state* and an *index of a row* in an associated trace word (word in Σ_{t}^*). Let $\widetilde{\Sigma} = \Sigma_{\mathsf{t}} \times \Sigma_{\mathsf{e}}$ be the product of the trace and execution alphabets.

In what follows, if $\widetilde{\tau} = \widetilde{\sigma}_1 \dots \widetilde{\sigma}_f \in \Sigma_{\mathsf{t}}^*$, then $\widetilde{\sigma}_j = (\widetilde{\sigma}_{1,j},\dots,\widetilde{\sigma}_{t,j})$ denotes the elements of the j-th letter of the word $\widetilde{\tau}$ for $j \in \{1,\dots,f\}$, and we use $\widetilde{\tau}_i = \widetilde{\sigma}_{i,1}\dots\widetilde{\sigma}_{i,f}$ to denote the i-th row of $\widetilde{\tau}$ for $i \in \{1,\dots,t\}$. Similarly, if $\widetilde{\tau} = \widetilde{\sigma}_1 \dots \widetilde{\sigma}_f \in \widetilde{\Sigma}^*$, the j-th letter is $\widetilde{\sigma}_j = (\widetilde{\sigma}_{1,j},\dots,\widetilde{\sigma}_{t,j},\widetilde{\eta}_{1,1,j},\dots,\widetilde{\eta}_{1,\widetilde{r},j},\dots,\widetilde{\eta}_{m,1,j},\dots,\widetilde{\eta}_{m,\widetilde{r},j})$, for $j \in \{1,\dots,f\}$, and the i-th row is $\widetilde{\tau}_i = \widetilde{\sigma}_{i,1}\dots\widetilde{\sigma}_{i,f}$, for $i \in \{1,\dots,t\}$. For $n \in \{1,\dots,m\}$, $h \in \{1,\dots,\widetilde{r}\}$ and $j \in \{1,\dots,f\}$, the corresponding letter from Σ_{e} is denoted $\widetilde{\eta}_{n,h,j} = (p_{n,h,j}, b_{n,h,j}, b'_{n,h,j}, q'_{n,h,j}, t_{n,h,j})$.

In the remainder of this section we present the intuition behind the automata constructions and their properties.

4.3 PDA accepting traces in a graph

The PDA $\mathscr{P}_{\mathsf{t}}$ is the intersection of a PDA $\mathscr{P}$ obtained from $\mathscr{G}^k$, where $\mathscr{G} = (V,\Sigma,R,G_0)$ is an SPGG, and a NFA $\mathscr{A}_{\mathsf{r}}$ that checks that register identifiers are correctly placed.

The construction of $\mathscr{P} = (Q_p, q_p^0, \Sigma_{\mathsf{t}} \dot\cup \{\vdash,\dashv\}, \Sigma_{\mathsf{t}} \dot\cup \{\vdash,\dashv\} \dot\cup \widetilde{V} \dot\cup \{\perp\}, \perp, \delta_p)$ resembles the classical construction of a PDA given a CFG. Here, instead of words generated by a CFG the language $\mathscr{L}(\mathscr{P})$ of $\mathscr{P}$ consists of t-tuples of (annotated) traces in some graph generated by the grammar. The automaton has a stack alphabet $\Sigma_{\mathsf{t}} \dot\cup \{\vdash,\dashv\} \dot\cup \widetilde{V} \dot\cup \{\perp\}$, where $\widetilde{V}$ consists of symbols corresponding to the variables in $\mathscr{G}$. The transitions in δ_p can be grouped according to the top symbol on the stack: empty stack, top symbol $\widetilde{\sigma} \in \Sigma_{\mathsf{t}} \cup \{\vdash,\dashv\}$, and top symbol $\widetilde{v} \in \widetilde{V}$. Transitions for $\widetilde{v} \in \widetilde{V}$ correspond to the production rules of the SPGG $\mathscr{G}$. For the series composition δ_p employs the additional symbol $\flat$ to allow for traces that are aligned in a way that letters in $\{1,\dots,t\}^t$ reflect the information about nodes shared by the respective paths. For the parallel composition δ_p guesses symbols, in the graphs generated by which the corresponding traces occur, together with the number of traces in the subgraph generated by each symbol. The number of times a new branch is introduced is bounded by t, the number of parallel traces.

$\mathscr{P}$ does not check that the register identifiers in the annotation are consistent among letters corresponding to edges in the graph that share a node, i.e., that letters corresponding to these edges have the same identifier for this node. This is done by the NFA $\mathscr{A}_{\mathsf{r}} = (Q_{\mathsf{r}}, q_{\mathsf{r}}^0, \Sigma_{\mathsf{t}} \dot\cup \{\vdash,\dashv\}, \delta_{\mathsf{r}}, F_{\mathsf{r}})$, which also verifies that identifiers for different nodes are unique. To this end, each state $\widetilde{q}$ of $\mathscr{A}_{\mathsf{r}}$ contains a path index $l_h \in \{1,\dots,t\}$ and a register identifier $i_h \in \{0,\dots,k\}$ for each row $\widetilde{\tau}_h$ of $\widetilde{\tau}$. δ_{r} checks that the letters in $\widetilde{\tau}$ that correspond to edges incident with the same node agree on the corresponding register identifier. The path indices l_h in $\widetilde{q}$ are used to identify branching or joining paths and the register identifiers i_h to check the required equalities. In the accepting states the equalities for the sink node of the graph must be satisfied. Additionally, δ_{r} verifies that the register identifiers in $\widetilde{\tau}$ corresponding to different nodes are different.

The PDA $\mathscr{P}_{\mathsf{t}}$ has $\mathscr{L}(\mathscr{P}_{\mathsf{t}}) = \mathscr{L}(\mathscr{P}) \cap \mathscr{L}(\mathscr{A}_{\mathsf{r}})$. The construction of $\mathscr{P}$ and $\mathscr{A}_{\mathsf{r}}$ ensures that if $\widetilde{\tau} \in \mathscr{L}(\mathscr{P}_{\mathsf{t}})$, then there exists $(G,\kappa) \in \mathscr{L}^u(\mathscr{G}^k)$ and for each $i \in \{1,\dots,t\}$ there exists a sequence of nodes $\widetilde{\pi}_i$ in G such that for each row $\widetilde{\tau}_i$ of $\widetilde{\tau}$ there exists a subsequence $\pi_i \in \mathsf{Paths}(G,\tau_i)$ of $\widetilde{\pi}_i$ corresponding to the projection $\tau_i = (\widetilde{\tau}_i|_{\Sigma\times\{0,\dots,k\}^2\times\{1,\dots,t\}})|_\Sigma$ of $\widetilde{\tau}_i$ on Σ. Furthermore, these paths can be chosen such that edges corresponding to rows with the same path index connect the same pair of nodes. Additionally, the mapping κ for the nodes on these paths agrees with the corresponding register identifiers in $\widetilde{\tau}$.

Conversely, if $(G,\kappa) \in \mathscr{L}^u(\mathscr{G}^k)$ and for every $n \in \{1,\dots,m\}$ and $h \in \{1,\dots,\widetilde{r}\}$ we are given a path $\widehat{\pi}_{n,h} \in \mathsf{Paths}(G,\widehat{\tau}_{n,h})$ for some trace $\widehat{\tau}_{n,h} \in \Sigma^*$, then there exists a word $\widetilde{\tau} \in \mathscr{L}(\mathscr{P}_{\mathsf{t}})$, which corresponds to these paths and traces. The word $\widetilde{\tau}$ is obtained by ordering, extending and annotating the given traces.

4.4 2NFA accepting executions

We construct a 2NFA $\widetilde{\mathscr{A}_n}$ for each $n \in \{1,\dots,m\}$ that checks that the sequence described by the rows of the word that correspond to n is indeed an execution of $\mathscr{M}$ that reads the corresponding rows of the trace word. Furthermore, $\widetilde{\mathscr{A}_n}$ verifies that the machine switches between traces described by different rows of the trace word only at positions at which the traces share a node in the corresponding paths.

Each state $\widetilde{q}$ of $\widetilde{\mathscr{A}_n} = (\widetilde{Q}_n, \widetilde{Q}^0_n, \widetilde{\Sigma} \mathbin{\dot\cup} \{\vdash,\dashv\}, \widetilde{\delta}_n, \widetilde{Q}_n)$ contains a state $q \in Q$ of $\mathscr{M}$, which is the current state of the simulated machine, and an index $i \in \{1,\dots,t\}$ in the trace-word that is part of the input word. $\widetilde{\delta}_n$ refers to the transition relation δ of $\mathscr{M}$ to check the existence of a transition of $\mathscr{M}$ that performs the read and write operations determined by the read letter of $\widetilde{\tau}$. The state q is updated according to δ and remains unchanged when the machine is inactive in the current part of the trace. The row of the trace word that is read in state $\widetilde{q}$ is determined by i. The index i can be changed by $\widetilde{\delta}_n$ only if for the current letter we have $\widetilde{\sigma}_{t_{n,h}} \in \{1,\dots,t\}$ and $p_{n,h} > 0$, and for the new value i' it must hold that $\sigma_{i'} = \sigma_i$. That is, the machine can switch between traces only at positions where the paths intersect. An additional component of $\widetilde{q}$ is used to check that block number 0 in $\widetilde{\tau}$ is used to correctly encode the reversals of the machine (which do not have to be at the start or sink nodes of the graph). All states are accepting.

Then, $\widetilde{\tau} \in \mathscr{L}(\widetilde{\mathscr{A}_n})$ iff by taking the elements of $\widetilde{\tau}$ corresponding to machine n in the appropriate order we can construct an execution ξ_n, formally defined as follows. For each $h \in \{1,\dots,\widetilde{r}\}$ and $l \in \{1,\dots,f\}$, if $\widetilde{\sigma}_{t_{n,h,l},l} = (\sigma, c, j_1, j_2) \in \Sigma \times \{1,\dots,t\} \times \{0,\dots,k\}^2$ and $p_{n,h,l} > 0$, then, if h is odd, then $\widehat{\xi}_{n,h,l} = (\sigma, b_{n,h,l}, b'_{n,h,l}) \cdot q'_{n,h,l}$, and if h is even, then $\widehat{\xi}_{n,h,l} = q'_{n,h,l} \cdot (\sigma, b_{n,h,l}, b'_{n,h,l})$, Otherwise, $\widehat{\xi}_{n,h,l} = \varepsilon$. Then $\widehat{\xi}_{n,h} = \widehat{\xi}_{n,h,1} \cdot \ldots \cdot \widehat{\xi}_{n,h,f}$ if h is odd, and $\widehat{\xi}_{n,h} = \widehat{\xi}_{n,h,f} \cdot \ldots \cdot \widehat{\xi}_{n,h,1}$ otherwise. Finally, $\xi_n = q^0 \cdot \widehat{\xi}_{n,1} \cdot \ldots \cdot \widehat{\xi}_{n,\widetilde{r}}$.

4.5 2NFA accepting valid read-write sequences

Here we describe a 2NFA $\widetilde{\mathscr{A}_\mathsf{c}}$ that checks that the executions of the different machines described by the input word are compatible with each other. That, is that the read and write operations of different machines match when executed in the order determined by the input word, where each operation is labelled with a block number. $\widetilde{\mathscr{A}_\mathsf{c}}$ verifies that each block number is used in a single execution and that for each execution the sequence of positive block numbers is nondecreasing. To check the validity of the corresponding read-write sequence w.r.t. the initial register values, $\widetilde{\mathscr{A}_\mathsf{c}}$ tracks the register values at the end and at the beginning of each block and compares the values at the beginning of block $i+1$ with those at the end of block i. An *assumption* is a partial function $A : \{1,\dots,p\} \to \mathbb{B}^k$ that maps a block number to a valuation of the registers, representing the obligation to verify that at the beginning of a block the registers have the respective values. Similarly, a *guarantee* is a function $G : \{1,\dots,p\} \to \mathbb{B}^k$ used to propagate the guarantee that at the end of a block the registers have a certain value.

Each state of the automaton $\widetilde{\mathscr{A}_\mathsf{c}} = (\widetilde{Q}_\mathsf{c}, \widetilde{q}^0_\mathsf{c}, \widetilde{\Sigma} \mathbin{\dot\cup} \{\vdash,\dashv\}, \widetilde{\delta}_\mathsf{c}, \widetilde{F}_\mathsf{c})$ contains a block number p_n and a valuation of the registers β_n for machine n, a set P of already seen block numbers, an assumption A and a guarantee G. The transition relation $\widetilde{\delta}_\mathsf{c}$ checks that all read operations of machine n except those at the beginning of a block read the value stored in β_n. At the beginning of a block of machine n, $\widetilde{\delta}_\mathsf{c}$ guesses a valuation of the registers for read operations and stores them in β_n. The new block number and the guess are added to the set A. The values of its write operations are used to update β_n and, at the end of a block the respective guarantee is added to G. $\widetilde{\delta}_\mathsf{c}$ discharges assumptions in A for which the respective guarantees are in G. In an accepting state the set A should be empty and the set P of all block numbers in $\widetilde{\tau}$ should contain all block numbers smaller or equal the maximal one.

By construction, in each word $\widetilde{\tau} \in \mathscr{L}(\widetilde{\mathscr{A}_\mathsf{c}})$ each block number is assigned to at most one machine and for each machine the sequence of positive block numbers is nondecreasing. All such words $\widetilde{\tau}$ are accepted by $\widetilde{\mathscr{A}_{\mathsf{c}}}$ iff the read-write sequence, constructed by ordering elements of $\widetilde{\tau}$ according to block number while preserving the order for each individual machine, is valid w.r.t. the initial register contents.

4.6 NFA checking configuration properties

The NFA $\mathscr{A}_{\mathsf{s}} = (\widetilde{Q}_{\mathsf{s}}, \widetilde{q}^0_{\mathsf{s}}, \widetilde{\Sigma} \,\dot{\cup}\, \{\vdash, \dashv\}, \widetilde{\delta}_{\mathsf{s}}, \widetilde{F}_{\mathsf{s}})$ checks that in some run in $T(\mathscr{M}, m, \mathscr{G}^k)$ corresponding to the input word, a configuration that satisfies the given configuration property $F = \exists n.\ S \preceq \mu(n)$ is reached.

Since the configuration property F asserts the existence of a node in the graph of a configuration, potential such configurations can be detected by inspecting (at most) two consecutive letters in the word. The information relevant for the satisfaction of a configuration property consists of the block number and successor state components of the letters of the execution word and the letters of the trace word. Thus, we define the set $C = \{1,\dots,p\}^t \times (Q \cup \{\bot\})^t \times \Sigma_{\mathsf{t}}$ and consider pairs of elements of C.

Let $c_0 = (p^0_{1,1},\dots,p^0_{m,\widetilde{r}}, q^0_{1,1},\dots,q^0_{m,\widetilde{r}}, \widetilde{\sigma}^0_{1,1},\dots,\widetilde{\sigma}^0_{m,\widetilde{r}})$, $c_\bot = (p^\bot_{1,1},\dots,p^\bot_{m,\widetilde{r}}, q^\bot_{1,1},\dots,q^\bot_{m,\widetilde{r}}, \widetilde{\sigma}^\bot_{1,1},\dots,\widetilde{\sigma}^\bot_{m,\widetilde{r}})$, where for $n \in \{1,\dots,m\}$ and $h \in \{1,\dots,\widetilde{r}\}$, $p^0_{n,h} = p^\bot_{n,h} = 0$, $q^0_{n,h} = q^0$, $q^\bot_{n,h} = \bot$, $\widetilde{\sigma}^0_{n,h} = \widetilde{\sigma}^\bot_{n,h} = (\flat,1,0,0)$.

Let us consider two elements of the set C: $c' = (p'_{1,1},\dots,p'_{m,\widetilde{r}}, q'_{1,1},\dots,q'_{m,\widetilde{r}}, \widetilde{\sigma}'_{1,1},\dots,\widetilde{\sigma}'_{m,\widetilde{r}}) \in C$ and $c'' = (p''_{1,1},\dots,p''_{m,\widetilde{r}}, q''_{1,1},\dots,q''_{m,\widetilde{r}}, \widetilde{\sigma}''_{1,1},\dots,\widetilde{\sigma}''_{m,\widetilde{r}}) \in C$.

We say that the pair (c',c'') *occurs in* $\widetilde{\tau} = \widetilde{\sigma_1}\dots\widetilde{\sigma_f} \in \widetilde{\Sigma}$ iff there exists a sequence of consecutive letters in $\widetilde{\tau}$ such that those of c' and c'' that are not equal to c_0 and $c_\bot$ match these letters of $\widetilde{\tau}$ in the same order. Formally, (c',c'') *occurs in* $\widetilde{\tau}$ iff one of the following conditions is satisfied.

(1) $c' = c_0$, and $p''_{n,h} = p_{n,h,1}$, $q''_{n,h} = q_{n,h,1}$ and $\widetilde{\sigma}''_{n,h} = \widetilde{\sigma}_{t_{n,h,1},1}$ (c'' matches $\widetilde{\sigma}_1$).

(2) $c'' = c_\bot$, and $p'_{n,h} = p_{n,h,1}$, $q'_{n,h} = q_{n,h,1}$ and $\widetilde{\sigma}'_{n,h} = \widetilde{\sigma}_{t_{n,h,1},f}$ (c' matches $\widetilde{\sigma}_f$).

(3) There exists $1 < l \le f$ such that
- $p'_{n,h} = p_{n,h,l-1}$, $q'_{n,h} = q_{n,h,l-1}$ and $\widetilde{\sigma}'_{n,h} = \widetilde{\sigma}_{t_{n,h,l-1},l-1}$ (c' matches $\widetilde{\sigma}_{l-1}$),
- $p''_{n,h} = p_{n,h,l}$, $q''_{n,h} = q_{n,h,l}$ and $\widetilde{\sigma}''_{n,h} = \widetilde{\sigma}_{t_{n,h,l},l}$ (c'' matches $\widetilde{\sigma}_l$).

Consider a configuration $\gamma \in \Gamma$ of a run ρ that satisfies the configuration property F. This means that there exists an edge $e \in E$, such that some of the nodes $src(e)$ and $trg(e)$ makes the property true. Furthermore, there exits a set of machines involved in the satisfaction of the property in γ. Among these machines, we distinguish between the one that executed the last transition in ρ leading to this configuration and the remaining machines. By the definition of runs of $T(\mathscr{M}, m, \mathscr{G}^k)$, the current node and states of these remaining machines should be reached at the end of one of their execution blocks. We define a predicate about pairs of elements of C, sets of machines and corresponding positions in their executions (i.e., rows in the respective letter of the execution word). The automaton $\mathscr{A}_s$ will use this predicate to identify letters of the word that may encode configurations satisfying the configuration property F.

Let $S \in \mathbb{M}(Q)$, $M \subseteq \{1,\dots,m\}$ and $f_M : M \to \{1,\dots,\widetilde{r}\}$. For each $n \in M$, let $f_n = f_M(n)$ and if f_n is odd, then $p_n = p'_{n,f_n}$, $q_n = q'_{n,f_n}$ and $\sigma_n = \widetilde{\sigma}'_{n,f_n}$, and if f_n is even, then $p_n = p''_{n,f_n}$, $q_n = q''_{n,f_n}$ and $\sigma_n = \widetilde{\sigma}''_{n,f_n}$. Let $n_0 \in M$ be such that for each $n \in M$, it holds that $p_n \le p_{n_0}$. We define $p_n(c',c'',M,f_M) = p_n$ for each $n \in M$ and $n_0(c',c'',M,f_M) = n_0$.

The *node predicate* $\mathsf{NodeProperty}(S,c',c'',M,f_M)$ is true iff the conditions listed below hold.

- $S = [q_n \mid n \in M]$ and for each $n \in M \setminus \{n_0\}$, $p_n < p_{n_0}$ and $p'_{n,f_n} \neq p''_{n,f_n}$.
- One of the following requirements is satisfied:
  - $\sigma'_{n_0} \in \{1,\dots,t\}$ and $\sigma'_n = \sigma'_{n_0}$ for each $n \in M$.
  - $\sigma'_{n_0} = (\sigma_0, l_0, j_0, j'_0) \in \Sigma \times \{1,\dots,t\} \times \{0,\dots,k\}^2$ and for each $n \in M$ there exist $\sigma \in \Sigma$, $j, j' \in \{0,\dots,k\}$ such that $\sigma'_n = (\sigma, l_0, j, j')$.

- $\sigma''_{n_0} \in \{1,\dots,t\}$ and $\sigma''_n = \sigma''_{n_0}$ for each $n \in M$.
- $\sigma''_{n_0} = (\sigma_0, l_0, j_0, j'_0) \in \Sigma \times \{1,\dots,t\} \times \{0,\dots,k\}^2$ and for each $n \in M$ there exist $\sigma \in \Sigma, j, j' \in \{0,\dots,k\}$ such that $\sigma''_n = (\sigma, l_0, j, j')$.

In order to evaluate the above predicates on positions of the input word $\widetilde{\tau}$, the NFA $\mathscr{A}_{\mathsf{s}}$ stores in its state an element c' of C. The current letter of the input word determines an element c'' of C, and $\mathscr{A}_{\mathsf{s}}$ evaluates the node predicate on the pair (c', c''). In order to verify that some run corresponding to $\widetilde{\tau}$ contains a configuration satisfying the configuration property F, $\mathscr{A}_{\mathsf{s}}$ must detect all pairs on which the predicate $\mathsf{NodeProperty}$ holds true, that is, they might encode a final configuration. For each such pair $\mathscr{A}_{\mathsf{s}}$ must verify that some of the pairs on which the predicate $\mathsf{NodeProperty}$ holds actually fulfils a global condition on $\widetilde{\tau}$. Namely, that for all the involved machines, the currently executed block number is the last one. (We can, w.l.o.g. restrict to runs where only the last configuration can be final.)

A state $\widetilde{q}$ of $\mathscr{A}_{\mathsf{s}}$ contains an element c' of C, a boolean value $final \in \mathbb{B}$, and for each machine n it contains components $P_n \in 2^{\{1,\dots,p\}}$ and $p_n \in \{0,\dots,p\}$. The set P_n consists of the already seen block numbers for n. If the current letter defines $c \in C$ such that $\mathsf{NodeProperty}(S, c', c, M, f_M)$ holds true for some $M \subseteq \{1,\dots,m\}$ and some function f_M, then $final$ can be set to 1 if it is 0, and the current block number of machine n for each $n \in M$ can be stored in p_n, in order to verify later that a final configuration is indeed reached (by checking that p_n is the maximal block number for n). Based on the currently read letter of $\widetilde{\tau}$, the transition relation $\widetilde{\delta}_{\mathsf{s}}$ updates the component c' of the state. The accepting state $\widetilde{q}^f_{\mathsf{s}}$ can be entered after reading $\dashv$ if $final = 1$ and if p_n is the maximal block number for machine n for each n.

The construction of $\mathscr{A}_{\mathsf{s}}$ ensures that $\widetilde{\tau} \in \mathscr{L}(\mathscr{A}_{\mathsf{s}})$ iff there exists a pair (c', c'') occurring in $\widetilde{\tau}$ encoding a configuration that satisfies the given configuration property F.

4.7 Correctness of the algorithm

Let $\mathscr{A}_1, \dots, \mathscr{A}_m$ be an NFA obtained respectively from $\widetilde{\mathscr{A}_1}, \dots, \widetilde{\mathscr{A}_m}$ such that for each $i \in \{1,\dots,m\}$, $\mathscr{L}(\mathscr{A}_i) = \mathscr{L}(\widetilde{\mathscr{A}_i})$. Let $\mathscr{A}_{\mathsf{c}}$ be an NFA constructed from $\widetilde{\mathscr{A}_{\mathsf{c}}}$ such that $\mathscr{L}(\mathscr{A}_{\mathsf{c}}) = \mathscr{L}(\widetilde{\mathscr{A}_{\mathsf{c}}})$. We then construct the NFA $\mathscr{A}_{\mathsf{e}}$ by intersecting $\mathscr{A}_1, \dots, \mathscr{A}_m$, $\mathscr{A}_{\mathsf{c}}$ and $\mathscr{A}_{\mathsf{s}}$ and projecting the result on Σ_{t}, i.e., $\mathscr{L}(\mathscr{A}_{\mathsf{e}}) = \left(\bigcap_{i=1}^m \mathscr{L}(\mathscr{A}_i) \cap \mathscr{L}(\mathscr{A}_{\mathsf{c}}) \cap L(\mathscr{A}_{\mathsf{s}})\right)|_{\Sigma_{\mathsf{t}}}$. The PDA $\mathscr{A}$ is the intersection of $\mathscr{P}_{\mathsf{t}}$ and $\mathscr{A}_{\mathsf{e}}$.

Theorem 2. *$\mathscr{L}(\mathscr{A}) \neq \emptyset$ iff there exists an r-reversal bounded run $\rho = \gamma_0 \dots \gamma_f$ in $T(\mathscr{M}, m, \mathscr{G}^k)$ such that $\gamma_i \models F$ for some $0 \le i \le f$, i.e., a run that reaches a configuration satisfying F.*

5 Conclusion.

In this paper we define and study a class of concurrent finite-state automata traversing series-parallel graphs and communicating through shared finite registers located at the nodes of the graph. We considered a model in which a *fixed number* of finite-state machines traverse the nodes of a *series-parallel graph*. The series-parallel graphs are generated by a graph grammar, and as we do not impose an a priori bound on the size of the graphs, the resulting system is infinite-state. Since the emptiness problem for this model is in general undecidable, we consider a natural restriction by putting bounds on the number of reversals along the computation and the number of shared registers in the graph. With these two restrictions, we show that the emptiness problem is decidable and can be reduced to PDA emptiness.

As we noted in Section 2.4, our decidability result holds for a more general model of communication between the machines, in which either read or write (but not both) operations on registers can be non-local, that is, access a register that is not at the node where the machine is currently located. Another possible extension that we omitted for simplicity concerns the language of configuration properties.

While here we consider properties that quantify over individual nodes in the graph, we can, in principle, extend the construction described in Section 4.6 to handle configuration properties asserting the existence of edges with certain labels, or a fixed number of adjacent nodes and edges.

Interesting directions for future work include establishing the complexity of the bounded emptiness problem for our model, as well as studying different extensions. One possibility is to allow parametrization in the number of concurrent machines, another is to consider other classes of context-free GTSs. For example, using the techniques from [16] one can try to extend our results to a more general class of graphs of bounded tree width.

Acknowledgements. We thank the anonymous reviewers for their helpful and insightful comments.

References

[1] Parosh Aziz Abdulla, Bengt Jonsson, Marcus Nilsson & Mayank Saksena (2004): *A Survey of Regular Model Checking*. In Philippa Gardner & Nobuko Yoshida, editors: *CONCUR 2004 - Concurrency Theory, 15th International Conference, London, UK, August 31 - September 3, 2004, Proceedings*, *Lecture Notes in Computer Science* 3170, Springer, pp. 35–48, doi:10.1007/978-3-540-28644-8_3.

[2] Paolo Baldan, Andrea Corradini & Barbara König (2001): *A Static Analysis Technique for Graph Transformation Systems*. In: *Proc. CONCUR'01*, *LNCS* 2154, Springer, pp. 381–395, doi:10.1007/3-540-44685-0_26.

[3] Paolo Baldan, Andrea Corradini, Barbara König & Alberto Lluch-Lafuente (2006): *A Temporal Graph Logic for Verification of Graph Transformation Systems*. In: *WADT*, *LNCS* 4409, Springer, pp. 1–20, doi:10.1007/978-3-540-71998-4_1.

[4] Nathalie Bertrand, Giorgio Delzanno, Barbara König, Arnaud Sangnier & Jan Stückrath (2012): *On the Decidability Status of Reachability and Coverability in Graph Transformation Systems*. In: *RTA*, *LIPIcs* 15, doi:10.4230/LIPIcs.RTA.2012.101.

[5] Bruno Courcelle & Joost Engelfriet (2012): *Graph Structure and Monadic Second-Order Logic - A Language-Theoretic Approach*. 138, Cambridge University Press, doi:10.1017/CBO9780511977619.

[6] Frank Drewes, Hans-Jörg Kreowski & Annegret Habel (1997): *Hyperedge Replacement Graph Grammars*. In: *Handbook of Graph Grammars*, World Scientific, pp. 95–162.

[7] J. Engelfriet & G. Rozenberg (1997): In Grzegorz Rozenberg, editor: *Handbook of Graph Grammars and Computing by Graph Transformation*, chapter Node Replacement Graph Grammars, World Scientific Publishing Co., Inc., pp. 1–94, doi:10.1142/9789812384720_0001.

[8] J. Esparza, P. Ganty & R. Majumdar (2012): *A Perfect Model for Bounded Verification*. In: *LICS 2012*, IEEE Computer Society, pp. 285–294, doi:10.1109/LICS.2012.39.

[9] J. Esparza, P. Ganty & T. Poch (2014): *Pattern-Based Verification for Multithreaded Programs*. *ACM Trans. Program. Lang. Syst.* 36(3), pp. 9:1–9:29, doi:10.1145/2629644.

[10] E.M. Gurari & O.H. Ibarra (1981): *The Complexity of Decision Problems for Finite-Turn Multicounter Machines*. *J. Comput. Syst. Sci.* 22(2), pp. 220–229, doi:10.1016/0022-0000(81)90028-3.

[11] E.M. Gurari & O.H. Ibarra (1982): *Two-Way Counter Machines and Diophantine Equations*. *J. ACM* 29(3), pp. 863–873, doi:10.1109/SFCS.1981.52.

[12] John E. Hopcroft & Jeffrey D. Ullman (2000): *Introduction to Automata Theory, Languages and Computation, Second Edition*. Addison-Wesley.

[13] Oscar H. Ibarra (1978): *Reversal-Bounded Multicounter Machines and Their Decision Problems*. *J. ACM* 25(1), pp. 116–133, doi:10.1145/322047.322058.

[14] Oscar H. Ibarra (2014): *Automata with Reversal-Bounded Counters: A Survey*. In: *DCFS 2014*, Springer, pp. 5–22, doi:10.1007/978-3-319-09704-6_2.

[15] Barbara König & Vitali Kozioura (2006): *Counterexample-Guided Abstraction Refinement for the Analysis of Graph Transformation Systems*. In: *TACAS*, *LNCS* 3920, Springer, pp. 197–211, doi:10.1007/11691372_13.

[16] P. Madhusudan & Gennaro Parlato (2011): *The tree width of auxiliary storage*. In Thomas Ball & Mooly Sagiv, editors: *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26-28, 2011*, ACM, pp. 283–294, doi:10.1145/1926385.1926419.

[17] M. O. Rabin & D. Scott (1959): *Finite Automata and Their Decision Problems*. *IBM Journal of Research and Development* 3(2), pp. 114–125, doi:10.1147/rd.32.0114.

[18] Arend Rensink (2008): *Explicit State Model Checking for Graph Grammars*. In: *Concurrency, Graphs and Models*, *LNCS* 5065, Springer, pp. 114–132, doi:10.1007/978-3-540-68679-8_8.

[19] Arnold L. Rosenberg (1965): *On multi-head finite automata*. In: *6th Annual Symposium on Switching Circuit Theory and Logical Design*, IEEE Computer Society, pp. 221–228, doi:10.1109/FOCS.1965.19.

[20] M.Y. Vardi (2014): *From Löwenheim to PSL and SVA*. In: *Language, Culture, Computation. Computing - Theory and Technology - Essays Dedicated to Yaacov Choueka on the Occasion of His 75th Birthday, Part I*, *Lecture Notes in Computer Science* 8001, Springer, pp. 78–102, doi:10.1007/978-3-642-45321-2_5.