A Robust Theory of Series Parallel Graphs

RAJEEV ALUR, University of Pennsylvania, USA
CALEB STANFORD*, University of California, San Diego, USA and University of California, Davis, USA
CHRISTOPHER WATSON, University of Pennsylvania, USA

Motivated by distributed data processing applications, we introduce a class of labeled directed acyclic graphs constructed using sequential and parallel composition operations, and study automata and logics over them. We show that deterministic and non-deterministic acceptors over such graphs have the same expressive power, which can be equivalently characterized by Monadic Second-Order logic and the graded μ-calculus. We establish closure under composition operations and decision procedures for membership, emptiness, and inclusion. A key feature of our graphs, called *synchronized series-parallel graphs* (SSPG), is that parallel composition introduces a synchronization edge from the newly introduced source vertex to the sink. The transfer of information enabled by such edges is crucial to the determinization construction, which would not be possible for the traditional definition of series-parallel graphs.

SSPGs allow both ordered ranked parallelism and unordered unranked parallelism. The latter feature means that in the corresponding automata, the transition function needs to account for an arbitrary number of predecessors by counting each type of state only up to a specified constant, thus leading to a notion of *counting complexity* that is distinct from the classical notion of state complexity. The determinization construction translates a nondeterministic automaton with n states and k counting complexity to a deterministic automaton with 2^{n^2} states and kn counting complexity, and both these bounds are shown to be tight. Furthermore, for nondeterministic automata a bound of 2 on counting complexity suffices without loss of expressiveness.

CCS Concepts: • **Information systems** → **Data streaming**; • **Theory of computation** → **Formal languages and automata theory**.

Additional Key Words and Phrases: series-parallel graphs, distributed stream processing, regular languages, logic in computer science

ACM Reference Format:
Rajeev Alur, Caleb Stanford, and Christopher Watson. 2023. A Robust Theory of Series Parallel Graphs. *Proc. ACM Program. Lang.* 7, POPL, Article 37 (January 2023), 31 pages. https://doi.org/10.1145/3571230

1 INTRODUCTION

Regular languages and automata over words play a fundamental role in software development practice today; they are used for tasks including parsing and input validation [Chapman and Stolee 2016], program synthesis [Gulwani 2011], and program verification [Amadini 2021; Hojjat et al. 2019]. The theory of regular languages is especially appealing because it is *robust*: for example, (1) regular languages are closed under operations such as union, concatenation, and complement; (2) regular languages have multiple equivalent characterizations in terms of automata, regular expressions, and monadic second-order logic (MSO); and (3) relevant decision problems,

*Work done while at the University of Pennsylvania.

Authors' addresses: Rajeev Alur, Computer and Information Science, University of Pennsylvania, USA, alur@cis.upenn.edu; Caleb Stanford, University of California, San Diego, USA and University of California, Davis, USA, cstanford@ucsd.edu; Christopher Watson, Computer and Information Science, University of Pennsylvania, USA, ccwatson@seas.upenn.edu.

2475-1421/2023/1-ART37
https://doi.org/10.1145/3571230

including membership, equivalence, and emptiness, are decidable. This has prompted a longstanding investigation in logic in computer science to generalize the theory to more complex structures, for example to infinite words and trees [Thomas 1990] and nested words [Alur and Madhusudan 2004, 2009]. We continue this trend and study automata and logics over series-parallel graphs. We aim to identify a *robust theory* that enjoys the properties (1)-(3) that have contributed to the widespread success of word languages. We are in particular interested in series-parallel graphs because they can be used as an abstract model of partially ordered data; a robust theory of such graphs could enable new techniques for parsing, validation, synthesis, and verification for distributed data processing software.

The core object of our study is a class of labeled graphs which we call *synchronized series-parallel graphs* (SSPGs) (Section 2). An SSPG is a planar directed acyclic graph with a unique source and unique sink vertex constructed using three inductive cases: *series composition* of two graphs, *ordered parallel composition* of two graphs, and *unordered parallel composition* of any positive integer number of graphs. Each of the two parallel composition operations introduces a new source and sink vertex as well as new edges, including a *synchronization edge* that connects the source to the sink. We use a fixed finite set of edge labels to make the graph's structure visible; in particular, edge labels distinguish ordered and unordered parallelism. Each vertex of an SSPG has a label drawn from some finite alphabet Σ.

Practical Motivation. Series-parallel structures arise naturally in distributed data stream processing applications. Prior language proposals consider the items in a stream to be tuples (in the sense of relational databases) [Arasu et al. 2006; Babcock et al. 2002], arriving in parallel over multiple distributed devices, with additional constraints on ordering between stream events, particularly system events known as "punctuation marks" [Tucker et al. 2003]. Prior work (including a subset of the current authors) identifies series-parallel partial orders as a useful abstraction at the type system and language level for concisely describing these complex orderings [Alur et al. 2021; Kallas et al. 2022; Mamouras et al. 2019].

However, existing type systems for streams have no way to describe *temporal integrity constraints*; that is, constraints on the pattern of data that occurs over time. For example: "on each parallel stream, every begin-transaction event B is matched by a closing end-transaction event E." Regular languages of SSPGs could close this gap by providing a formalism for describing temporal integrity constraints – while ensuring that the constraints remain computationally tractable. We elaborate on this connection in Section 4.

The connection to data streams also motivates some of the design choices of the SSPG model. *Unordered parallelism* corresponds to a common form of parallelism in data streams known as key-based parallelism [Gedik 2014; Shatdal and Naughton 1995]. For example, a stream of bids in an online auction would be parallelized by item identifiers, yielding a parallel stream for each item being sold. This parallelism is inherently *unranked* (the number of parallel graphs is not bounded by a fixed constant), because the number of different items is not known statically. Unordered, unranked parallelism can express other standard streaming data models, including the sequence of relations model [Arasu et al. 2003, 2006] (Section 3.5). *Ordered parallelism* models parallel streams that should be treated asymmetrically, for example, a pair comprising a stream of auction bids and a stream of direct purchases. Multiple levels of nested parallelism (of both types) also occur, for example in Timely Dataflow [Murray et al. 2013]. We discuss design choices (including the asymmetry between binary ordered and n-ary unordered parallelism) in Section 3.4.

Results. To study regular properties of SSPGs, we introduce *synchronized series-parallel graph automata* (SSPGAs) as language acceptors over SSPGs (Section 3). An SSPGA is an acylic graph automaton [Arbib and Give'on 1968; Kamimura and Slutzki 1981a; Thomas 1997]. Similar to other

models, an SSPGA labels the vertices of a graph with states in a topologically sorted order based on each vertex's immediate predecessors, as allowed by the SSPGA's transition relation. At a join vertex following either an ordered or unordered parallel construct, the synchronization edge provides access to the state at the matching split vertex. A key difference from standard graph automata lies in the case of unordered parallelism, since a join node in an SSPG may have an unbounded number of predecessors. To ensure each SSPGA has a finite description, we introduce a transition relation based on *counting* the number of predecessors of each state up to a maximum finite threshold k, then applying a function to the vector of counts. As a brief preview, the structure of the unordered join transition function will be a function of type $Q \times [Q \to \{0, 1, 2, \ldots, k\}] \to Q$. For example, this allows transitioning to a state q' if exactly two predecessor vertices are in state q_1 and at most three predecessor vertices are in state q_2. This means that an automaton has two notions of succinctness: its *state complexity* and its *counting complexity* k.

Our main results are that the nondeterminstic and deterministic SSPGA models are expressively equivalent to each other, are compositional (Section 5), and are expressively equivalent to both MSO and graded μ-calculus over SSPGs (Section 7). We also show that membership, emptiness, and inclusion are decidable (Section 6). The determinization construction of a machine with state complexity n and counting complexity k yields a machine with state complexity 2^{n^2} and counting complexity kn; we show each blowup in complexity to be tight. Interestingly, we show more generally that a counting complexity of 2 suffices for full expressiveness of nondeterministic automata. We also show that a counting complexity of 1 leads to strictly weaker expressiveness. To our knowledge, these results on counting complexity have not been made in the context of automata on graphs or trees.

The addition of a *synchronization edge* between the new source and sink during each parallel composition of graphs plays a key role in our results. Inspired by nested words [Alur and Madhusudan 2009], the synchronization edge is necessary for a robust theory, in particular for determinization and equivalence with MSO (see Section 3.4). Importantly, synchronization edges do not fundamentally change the class of graphs considered: an SSPG G can be uniquely recovered from the graph G' formed by removing all synchronization edges from G. However, without synchronization edges, a pushdown model would be needed to recover the matching between split and join edges; with them, a finite-state model suffices.

While automata over series-parallel graphs have been defined previously [Lodaya and Weil 1998, 2000], as they lack the synchronization edge, they do not admit determinization. We discuss related work on tree, graph, trace, and word automata in Section 8.

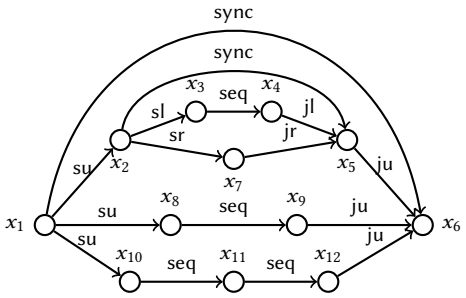
2 SYNCHRONIZED SERIES PARALLEL GRAPHS

Before introducing SSPGs, which have vertex labels drawn arbitrarily from some finite alphabet, we present their underlying structure in the form of *vertex-unlabeled SSPGs*.

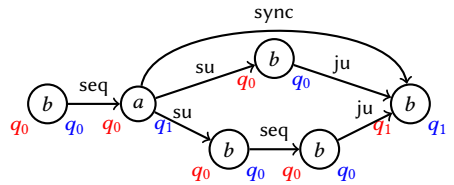
Vertex-Unlabeled SSPG. A vertex-unlabeled SSPG comprises a nonempty set of vertices, a source vertex, a sink vertex, and a set of labeled directed edges. The base case will be a single-vertex graph, where the source and sink are the same; in all other cases, the source and sink will be different. Edge labels are drawn from the set

$$\Gamma = \{\mathsf{seq}, \mathsf{sl}, \mathsf{sr}, \mathsf{su}, \mathsf{jl}, \mathsf{jr}, \mathsf{ju}, \mathsf{sync}\}$$

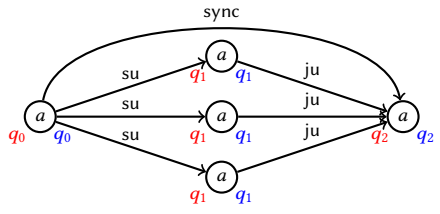
Where seq, sl, sr, su, jl, jr, ju, and sync are shorthands for (respectively) sequence, left-split, right-split, unordered-split, left-join, right-join, unordered-join, and synchronization. For arbitrary vertices u and v and some label $\gamma \in \Gamma$, the directed edge from u to v with label γ is denoted $u \xrightarrow{\gamma} v$. We say that v is the γ-successor of u and that u is the γ-predecessor of v.



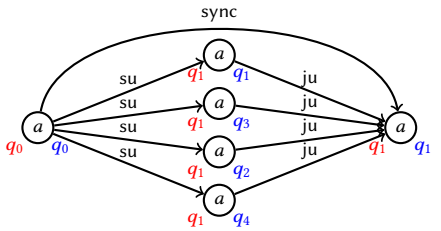
(a) A vertex-unlabeled SSPG.



(b) An accepting run of A_{SING} on an SSPG (see Example 3.7). Left- and right- states are marked in red and blue, respectively.



(c) A rejecting run of A_2 on a G_3 (see Example 3.8). At the join, the machine counts 3 incoming copies of the state q_1, and so transitions to the fail state.



(d) An accepting run of B_4 on G_4 (see Example 3.9). At the join, the machine counts one each of $q_1, \ldots, q_4$, and so the run can continue toward acceptance.

Fig. 1. Examples of SSPG structure and SSPGA runs.

Definition 2.1 (Vertex-unlabeled Synchronized Series Parallel Graph)**.** A vertex-unlabeled SSPG is a tuple (V, E, s, t) constructed using the rules below. Here, V is the nonempty vertex set, $E \subseteq V \times \Gamma \times V$ is the labeled edge set, $s \in V$ is the source, and $t \in V$ is the sink.

(1) **Singleton:** For an arbitrary vertex u, the graph
$$G = (\{u\}, \{\}, u, u)$$
is a vertex-unlabeled SSPG.

(2) **Series composition:** For arbitrary disjoint vertex-unlabeled SSPGs $G_1 = (V_1, E_1, s_1, t_1)$ and $G_2 = (V_2, E_2, s_2, t_2)$, the graph
$$G = (V_1 \cup V_2, E_1 \cup E_2 \cup \{t_1 \xrightarrow{\text{seq}} s_2\}, s_1, t_2)$$
is a vertex-unlabeled SSPG.

(3) **Ordered parallel composition:** For arbitrary disjoint vertex-unlabeled SSPGs $G_1 = (V_1, E_1, s_1, t_1)$ and $G_2 = (V_2, E_2, s_2, t_2)$ and fresh source and sink s and t we define:
- $V = V_1 \cup V_2 \cup \{s, t\}$
- $E = E_1 \cup E_2 \cup \{s \xrightarrow{\text{sl}} s_1, s \xrightarrow{\text{sr}} s_2, t_1 \xrightarrow{\text{jl}} t, t_2 \xrightarrow{\text{jr}} t, s \xrightarrow{\text{sync}} t\}$

and say $G = (V, E, s, t)$ is a vertex-unlabeled SSPG.

(4) **Unordered parallel composition:** For $n \geq 1$ arbitrary disjoint vertex-unlabeled SSPGs $G_1, ..., G_n$ such that for $1 \leq i \leq n$, $G_i = (V_i, E_i, s_i, t_i)$ and fresh source and sink s and t we define:
- $V = \bigcup_{i=1}^{n} [V_i] \cup \{s, t\}$

- $E = \bigcup_{i=1}^{n} [E_i \cup \{s \xrightarrow{\text{su}} s_i, t_i \xrightarrow{\text{ju}} t\}] \cup \{s \xrightarrow{\text{sync}} t\}$

and say $G = (V, E, s, t)$ is a vertex-unlabeled SSPG.

Each vertex in a vertex-unlabeled SSPG has exactly one *type*, uniquely determined by its incoming and outgoing edges. A vertex with outgoing left- and right-split edges is an *ordered-split* vertex. A vertex with at least one outgoing unordered-split edge is an *unordered-split* vertex. A vertex with incoming left- and right-join edges is an *ordered-join* vertex. A vertex with at least one incoming unordered-join edge is an *unordered-join* vertex. All other vertices are *internal* vertices. In Figure 1a, x_2 is an ordered-split vertex, x_1 is an unordered-split vertex, x_5 is an ordered-join vertex, x_6 is an unordered-join vertex, and $x_3, x_4, x_7, x_8, x_9, x_{10}, x_{11}$, and x_{12} are all internal vertices.

SSPG. An SSPG has the same underlying structure as a vertex-unlabeled SSPG, with the addition of vertex-labels drawn from a finite alphabet.

Definition 2.2 (SSPG)**.** A synchronized series-parallel graph (SSPG) is a tuple $(V, E, s, t, \Sigma, \sigma)$ such that

- (V, E, s, t) is a vertex-unlabeled SSPG
- Σ is a finite set of vertex labels
- $\sigma : V \to \Sigma$ is a vertex-labeling function

For an SSPG $G = (V, E, s, t, \Sigma, \sigma)$ we say that V is the vertex set, E is the edge set, s is the source, t is the sink, Σ is the set of vertex labels, and σ is the vertex labeling function of G. For a finite alphabet Σ, let $SSPG(\Sigma)$ denote the set of all labeled SSPGs with vertex labels drawn from Σ.

3 SSPG AUTOMATA

An *SSPG Automaton (SSPGA)* is a finite-state acceptor over SSPGs. In Section 3.1, we introduce the SSPGA model, which characterizes the notion of regularity for SSPG languages. In Section 3.2, we define the deterministic subclass of SSPGAs. In Section 3.3, we provide examples of SSPGAs and the runs they may take on certain SSPGs.

3.1 Syntax and Semantics

We fix the notation $[i..j]$ for the set of natural numbers $\{i, i+1, \ldots, j\}$ and define SSPGA as follows.

Definition 3.1 (SSPGA)**.** An SSPGA is defined by a tuple $(Q, \Sigma, Q_0, F, k, \Delta_{\text{up}}, \Delta_{\text{seq}}, \Delta_{\text{so}}, \Delta_{\text{su}}, \Delta_{\text{jo}}, \Delta_{\text{ju}})$ where

- Q is a finite set of states
- Σ is a set of vertex labels
- $Q_0 \subseteq Q$ is the set of initial states
- $F \subseteq Q$ is the set of final (accepting) states
- $k \in \mathbb{N}$ is the *counting complexity* of A
- $\Delta_{\text{up}} \subseteq Q \times \Sigma \times Q$ is the *update* transition relation
- $\Delta_{\text{seq}} \subseteq Q \times Q$ is the *sequence* transition relation
- $\Delta_{\text{so}} \subseteq Q \times Q \times Q$ is the *ordered split* transition relation
- $\Delta_{\text{su}} \subseteq Q \times Q$ is the *unordered split* transition relation
- $\Delta_{\text{jo}} \subseteq Q \times Q \times Q \times Q$ is the *ordered join* transition relation
- $\Delta_{\text{ju}} \subseteq Q \times [Q \to [0..k]] \times Q$ is the *unordered join* transition relation (note that this is a finite set, so there are finitely many possible transitions).

A *run* of an SSPGA on an SSPG is an assignment of a *left-state* and a *right-state* to each vertex of the graph. The update transition relation defines legal triples of left-state, vertex label, and right-state at a single vertex. The remaining transition relations define legal combinations of left-state at

a vertex and right-states at its predecessors, or right-state at a vertex and left-states at its successors. Since an unordered-join vertex may have unboundedly many unordered-join predecessors, the states of which cannot all be maintained by a finite control, the SSPGA will "stop counting" once a per-state threshold is reached. This threshold is the counting complexity of the SSPGA, and motivates the following definition:

Definition 3.2 (k-truncation ($\lfloor\ \rfloor_k$))**.** For any $k \in \mathbb{N}$ and finite set Q, we define a function *k-truncate* that maps an arbitrary multiset M with elements drawn from Q to a vector $\lfloor M \rfloor_k : Q \to [0..k]$ such that for any $q \in Q$,

$$\lfloor M \rfloor_k(q) = \begin{cases} i, & q \text{ appears in } M \text{ with multiplicity } i < k \\ k, & \text{otherwise} \end{cases}$$

and say that $\lfloor M \rfloor_k$ is the k-truncation of M.

Definition 3.3 (Run of an SSPGA)**.** A run of an SSPGA $A = (Q, \Sigma, Q_0, F, k, \Delta_{\mathsf{up}}, \Delta_{\mathsf{seq}}, \Delta_{\mathsf{so}}, \Delta_{\mathsf{su}}, \Delta_{\mathsf{jo}}, \Delta_{\mathsf{ju}})$ on an SSPG $G = (V, E, s, t, \Sigma, \sigma)$ is an assignment of a left-state $L(u)$ and right-state $R(u)$ to each vertex in V such that the following conditions hold:

(1) At each vertex u the states satisfy $(L(u), \sigma(u), R(u)) \in \Delta_{\mathsf{up}}$
(2) At any vertex u with a sequence successor v, the states satisfy $(R(u), L(v)) \in \Delta_{\mathsf{seq}}$
(3) At any ordered-split vertex u with left-split successor v_ℓ and right-split successor v_r, the states satisfy $(R(u), L(v_\ell), L(v_r)) \in \Delta_{\mathsf{so}}$
(4) At any unordered-split vertex u and any unordered-split successor v of u, the states satisfy $(R(u), L(v)) \in \Delta_{\mathsf{su}}$
(5) At any ordered-join vertex v with synchronization, left-join, and right-join predecessors u_s, u_ℓ, and u_r, the states satisfy $(R(u_s), R(u_\ell), R(u_r), L(v)) \in \Delta_{\mathsf{jo}}$
(6) At any unordered-join vertex v with synchronization predecessor u_s and unordered-join predecessors $u_1, \ldots, u_m$, the states satisfy $(R(u_s), \lfloor \{R(u_1), \ldots, R(u_m)\} \rfloor_k, L(v)) \in \Delta_{\mathsf{ju}}$

A run is *accepting* if it satisfies the above conditions, assigns an element of Q_0 as the left-state of the source s, and assigns an element of F as the right-state of the sink t.

Definition 3.4 (Language of an SSPGA)**.** A SSPGA A *accepts* an SSPG G iff there exists an accepting run of A on G. The *language* $L(A)$ of an SSPGA A is the set of all SSPGs that A accepts.

A *language* is a (possibly infinite) set of SSPGs. A language L is *regular* iff there exists an SSPGA A such that $L = L(A)$.

3.2 Deterministic SSPGA

A *deterministic SSPGA (DSSPGA)* is an SSPGA such that during a run on an SSPG, for any vertex u, each assignment of right-states to the predecessors of u uniquely determines the left-state of u, and each pair of left-state and label of u uniquely determines the right-state of u. Additionally, a DSSPGA must have exactly one initial state.

Definition 3.5 (Deterministic SSPGA (DSSPGA))**.** A SSPGA $A = (Q, \Sigma, Q_0, F, k, \Delta_{\mathsf{up}}, \Delta_{\mathsf{seq}}, \Delta_{\mathsf{so}}, \Delta_{\mathsf{su}}, \Delta_{\mathsf{jo}}, \Delta_{\mathsf{ju}})$ is a *deterministic SSPGA (DSSPGA)* if

(1) $|Q_0| = 1$
(2) $\forall q \in Q,\ \forall a \in \Sigma$ there is exactly one $q' = \delta_{\mathsf{up}}(q, a)$ such that $(q, a, q') \in \Delta_{\mathsf{up}}$
(3) $\forall q \in Q$ there is exactly one state $q' = \delta_{\mathsf{seq}}(q)$ such that $(q, q') \in \Delta_{\mathsf{seq}}$
(4) $\forall q \in Q$ there is exactly one pair $(q_\ell, q_r) = \delta_{\mathsf{so}}(q)$ such that $(q, q_\ell, q_r) \in \Delta_{\mathsf{so}}$
(5) $\forall q \in Q$ there is exactly one $q' = \delta_{\mathsf{su}}(q)$ such that $(q, q') \in \Delta_{\mathsf{su}}$

(6) $\forall q_s, q_l, q_r \in Q$ there is exactly one $q' = \delta_{\text{jo}}(q_s, q_\ell, q_r)$ such that $(q_s, q_\ell, q_r, q') \in \Delta_{\text{jo}}$
(7) $\forall q \in Q$, $\forall f \in [Q \to [0..k]]$ there is exactly one $q' = \delta_{\text{ju}}(q, f)$ such that $(q, f, q') \in \Delta_{\text{ju}}$

Definition 3.6 (Deterministic run of a DSSPGA)**.** The *deterministic run* of a DSSPGA $A = (Q, \Sigma, Q_0, F, k, \Delta_{\text{up}}, \Delta_{\text{seq}}, \Delta_{\text{so}}, \Delta_{\text{su}}, \Delta_{\text{jo}}, \Delta_{\text{ju}})$ on an SSPG $G = (V, E, s, t, \Sigma, \sigma)$ is the unique run of A on G that assigns the sole element of Q_0 to be the left-state of s.

For any DSSPGA $A = (Q, \Sigma, Q_0, F, k, \Delta_{\text{up}}, \Delta_{\text{seq}}, \Delta_{\text{so}}, \Delta_{\text{su}}, \Delta_{\text{jo}}, \Delta_{\text{ju}})$, Definition 3.5 implicitly defines functions $\delta_{\text{up}}, \delta_{\text{seq}}, \delta_{\text{so}}, \delta_{\text{su}}, \delta_{\text{jo}}, \delta_{\text{ju}}$ that characterize the transitions of A. To highlight the fact that such an A is deterministic, we may equivalently denote A as $(Q, \Sigma, Q_0, F, k, \delta_{\text{up}}, \delta_{\text{seq}}, \delta_{\text{so}}, \delta_{\text{su}}, \delta_{\text{jo}}, \delta_{\text{ju}})$. When an SSPGA is known to be deterministic, we may switch freely between the "transition relation" (using capital Δ_γ) and "transition function" (using lower-case δ_γ) notations.

Expressive equivalence of deterministic and nondeterministic SSPGAs is shown in section 5.1.

3.3 Examples

Example 3.7 (Single occurrence of a)**.** Fix $\Sigma = \{a, b\}$ and let $\text{SING}(a)$ be the set of all SSPGs with labels drawn from Σ containing exactly one vertex labeled a. We define a DSSPGA A_{SING} such that $L(A_{\text{SING}}) = \text{SING}(a)$.

A_{SING} has three states: q_0 with the interpretation "no a has been encountered," q_1 with the interpretation "one a has been encountered," and a failing state q_2. At a vertex labeled a, if the left-state is q_0 then the update transition function sets the right-state to be q_1. Similarly, the left-states q_1 and q_2 would each transition to q_2 on the symbol a.

At a split vertex assigned some right-state q, the machine makes use of the synchronization edge to recover q at the corresponding join. Therefore, all non-synchronization successors can be initialized to q_0. On a join vertex, if there have been at least two as encountered in total across all the parallel branches and prior to the split, the machine assigns the failing left-state q_2. For unordered joins, this requires counting complexity 2. An example run is shown in Figure 1b.

Formally, $A_{\text{SING}} = (\{q_0, q_1, q_2\}, \Sigma, \{q_0\}, \{q_1\}, 2, \Delta_{\text{up}}, \Delta_{\text{seq}}, \Delta_{\text{so}}, \Delta_{\text{su}}, \Delta_{\text{jo}}, \Delta_{\text{ju}})$. where the transition relations are given by:

- $\Delta_{\text{up}} = \{(q_0, a, q_1), (q_0, b, q_0), (q_1, a, q_2), (q_1, b, q_1), (q_2, a, q_2), (q_2, b, q_2)\}$
- $\Delta_{\text{seq}} = \{(q_0, q_0), (q_1, q_1), (q_2, q_2)\}$
- $\Delta_{\text{so}} = \{(q_0, q_0, q_0), (q_1, q_0, q_0), (q_2, q_0, q_0)\}$
- $\Delta_{\text{su}} = \{(q_0, q_0), (q_1, q_0), (q_2, q_0)\}$
- $\Delta_{\text{jo}} = D_1 \cup D_2$ where
  - $D_1 = \{(q_0, q_0, q_0, q_0), (q_1, q_0, q_0, q_1), (q_0, q_1, q_0, q_1), (q_0, q_0, q_1, q_1)\}$
  - $D_2 = (Q \times Q \times Q \times \{q_2\}) \setminus D_1$
- $\Delta_{\text{ju}} \subseteq Q \times [Q \to [0..2]] \times Q = D_1 \cup D_2$ where
  - $D_1 = \{(q_0, f, q_0) \mid f(q_1) = 0 \wedge f(q_2) = 0\} \cup$
    $\{(q_0, f, q_1) \mid f(q_1) = 1 \wedge f(q_2) = 0\} \cup \{(q_1, f, q_1) \mid f(q_1) = 0 \wedge f(q_2) = 0\}$
  - $D_2 = (Q \times [Q \to [0..2]] \times \{q_2\}) \setminus D_1$

Example 3.8 (Count parallel branches)**.** Fix the unary alphabet $\Sigma = \{a\}$. For any natural number s, let G_s be the SSPG comprising the unordered parallel composition of s singleton vertices. Let L_s be the one-element language containing G_s. For any s, we construct a DSSPGA A_s such that $L(A_s) = L_s$.

A_s has three states: a start state q_0, a "normal operation" state q_1, and a fail state q_2. During its deterministic run on the graph $G_s \in L_s$, A_s assigns q_0 to be left- and right-state of the source and assigns q_1 to be the the left- and right-state of every other vertex in G_s.

A_s uses its counting complexity $s+1$ to ensure the right number of parallel branches. Δ_{ju} requires that the count of q_0 in the $(s+1)$-truncation of the states at its unordered join-predecessors is exactly s. It also requires that none of the unordered join-predecessors of the join vertex were assigned the fail state q_2. If these requirements are met, the successor (which must be the sink) is assigned q_1 and A_s will accept the graph. An example run for A_2 is shown in Figure 1c.

Formally, we let $A_s = (Q, \Sigma, \{q_0\}, \{q_1\}, s+1, \Delta_{\text{up}}, \Delta_{\text{seq}}, \Delta_{\text{so}}, \Delta_{\text{su}}, \Delta_{\text{jo}}, \Delta_{\text{ju}})$ where the state-set $Q = \{q_0, q_1, q_2\}$ and the transition functions are defined as follows:

- $\Delta_{\text{up}} = \{(q_0, a, q_0), (q_1, a, q_1), (q_2, a, q_2)\}$
- $\Delta_{\text{seq}} = Q \times \{q_2\}$
- $\Delta_{\text{so}} = Q \times \{q_2\} \times \{q_2\}$
- $\Delta_{\text{su}} = \{(q_0, q_1), (q_1, q_2), (q_2, q_2)\}$
- $\Delta_{\text{jo}} = Q \times Q \times Q \times \{q_2\}$
- $\Delta_{\text{ju}} = \{(q_0, f, q_1)\} \cup ((Q \times [Q \to [0..s+1]] \times \{q_2\}) \setminus \{(q_0, f, q_2)\})$ where f is the count vector such that $f(q_0) = 0$, $f(q_1) = s$, and $f(q_2) = 0$

Observe that $L(A_s) = L_s$.

Perhaps counterintuitively, it is also possible to construct a (nondeterministic) SSPGA that accepts L_s but has counting complexity 2.

Example 3.9 (Nondeterministic counting)**.** Let L_s be defined as in example 3.8. We define such an SSPGA $B_s = (Q, \Sigma, \{q_0\}, \{q_1\}, 2, \Delta_{\text{up}}, \{\}, \{\}, \Delta_{\text{su}}, \{\}, \Delta_{\text{ju}})$ such that $L(B_s) = L_s$.

B_s includes *indexes* in its states to keep counting complexity constant. The states of B_s are $\{q_0, \ldots, q_s\}$. For any $i \in [1..s]$, q_i is the state with index i (q_0 is not part of the indexing scheme).

Immediately before the join, Δ_{up} can nondeterministically set the state at a join-predecessor to any of $q_1, ..., q_s$. Then, Δ_{ju} requires that the multiplicity of each of $q_1, \ldots, q_s$ is exactly 1 (and the multiplicity of q_0 is 0) in the incoming count-vector. By requiring exactly one state of each index, the machine "counts" to s while maintaining counting complexity 2. An example run for B_4 is shown in Figure 1d.

The sequence, ordered-split, and ordered-join transition relations are empty to ensure that the graph consists of exactly one ordered-parallel composition of singleton vertices. Formally,

- $Q = \{q_0, \ldots, q_s\}$
- $\Delta_{up} = \{(q_0, a, q_0)\} \cup (\{q_1\} \times \{a\} \times Q)$
- $\Delta_{\text{su}} = \{(q_0, q_1)\}$
- $\Delta_{\text{ju}} = \{(q_0, f, q_1) \mid f(q_0) = 0 \wedge \forall q \in Q \setminus \{q_0\}. f(q) = 1\}$

3.4 Design Choices

Unranked Parallelism. In the SSPG model, ordered parallelism is ranked and binary, while unordered parallelism is unranked (n-ary for any n). Our rationale is that on the one hand n-ary ordered parallelism can be encoded using repeated binary parallelism (this trick, similar to currying, can be applied in unranked, ordered tree automata [Courcelle 1989; Cristau et al. 2005]). On the other hand, unranked unordered parallelism cannot be encoded this way: any repeated binary operation would implicitly define an order which could then be used by the model. To wit, consider the language of SSPGs EVEN(a), consisting of SSPGs with an even number of as. This would be regular if only binary ordered (or even binary unordered!) parallelism were allowed, but it is not regular in the unordered case (following our definition). Intuitively, computing this property requires picking an order on as, then counting them up (modulo 2).

This highlights a difference between *order-independent* processing (where the input is processed in order, but the final result is order-independent) and *order-unaware* processing (where the processing

does not have access to an order). If we were to allow computations over unordered parallel graphs to be only order-independent, and not order-unaware, it would break the expressive equivalence with MSO shown in Section 7.1, giving up robustness property (2) from the introduction. See Section 8 for a comparison of our model with existing automaton models defined over traditional series-parallel graphs, which do not support unranked unordered parallelism.

Transition Semantics of Unordered Join. Rather than counting up to a set threshold, an alternative treatment of unordered joins would be to define a merge function $merge : Q \times Q \to Q$ and apply it repeatedly (see e.g. [Courcelle 1989; Lodaya and Weil 1998]). For deterministic automata, we would then need to require that *merge* be commutative and associative [Courcelle 1989]. However, just as with encoding n-ary unranked parallelism using ranked parallelism, this choice would give up expressive equivalence with MSO because the property EVEN(a) discussed above is expressible with a commutative, associative merge function but not in MSO.

We choose to describe unordered join transitions rather abstractly, as arbitrary functions $Q \times [Q \to [0..k]] \to Q$. This leaves open the question of how to *symbolically* represent these functions in an implementation, where rather than giving the function as a complete table, we might describe symbolic expressions such as "transition to state q_2 if there are at least 3 copies of state q_3 and no copies of state q_4". This is analogous to in classical automata theory how the state set Q and transition set Δ are abstract functions, but in applications it is more fruitful to represent them symbolically using predicates, Boolean expressions, or binary decision diagrams (BDDs) [D'Antoni and Veanes 2021; D'Antoni and Veanes 2017]. By leaving the function abstract, we are able to study a natural notion of counting complexity that is implementation-independent, analogous to how state complexity is independent of how states are physically represented.

Finally, another alternative treatment of unordered joins would be to disallow counting, and instead express only a Boolean combination of existential and universal properties: e.g., there exists a predecessor labeled q_2 but no predecessor labeled q_3. This is equivalent to our model where counting complexity is restricted to be at most 1. This is natural but less expressive, and would lead to only expressing bisimulation-invariant properties, an expressive limitation shared by the (non-graded) modal μ-calculus [Bradfield and Walukiewicz 2018]. In our model, we show that counting complexity of 2 is necessary and sufficient for full expressiveness in the nondeterministic setting (Section 5.2).

Synchronization Edge. The synchronization edge is inspired by nested words, where it has been shown that removing it results in a loss of expressiveness, where nondeterministic and deterministic automata no longer coincide in expressiveness [Alur and Madhusudan 2009]. Split- and join-vertices appear in matched pairs in an SSPG, analogously to well-matched parentheses in a string. We include the synchronization edge to make this matching structure visible; when an SSPGA processes a join-vertex, the synchronization edge provides information from the matching split-vertex. Without the synchronization edge, a pushdown model with unbounded memory would be needed to recover the matching structure.

The synchronization edge in SSPGs allows the model to naturally express properties that require saving some state before a split. The following result shows that without the synchronization edge, it is impossible to define even simple languages such as SING(a) (Example 3.7), the set of graphs with exactly one vertex labeled a, using a DSSPGA. This lack of expressiveness would preclude expressive equivalence to MSO, because SING(a) is MSO-definable even without the synchronization-edge.

Proposition 3.10. *Fix $\Sigma = \{a, b\}$ and define SING(a) as in Example 3.7. Any DSSPGA A whose unordered-join transition function δ_{jo} does not use the* sync *edge (i.e. for any choice of q, q', and f, $\delta_{\mathrm{jo}}(q, f) = \delta_{\mathrm{jo}}(q', f)$) does not recognize SING(a).*

Proof. Intuitively, without the sync edge, on a nested composition of unordered parallel compositions, the automaton would have to keep track of the exact history of where a prior a was in order to distinguish it from any other joined threads.

Assume for the sake of contradiction that $A = (Q, \Sigma, Q_0, F, k, \delta_{\text{up}}, \delta_{\text{seq}}, \delta_{\text{so}}, \delta_{\text{su}}, \delta_{\text{jo}}, \delta_{\text{ju}})$ recognizes $\text{SING}(a)$. Let m be some integer such that $m > |Q|$. For any choice of i and j s.t. $i < j < m$, define two SSPGs G_i and $G_{i,j}$ that differ only in their vertex labels. Each SSPG starts with a linear path of i vertices connected by nested su edges. The i^{th} vertex of each graph is a binary unordered-split vertex. Each parallel branch is a linear path of $2m-2i-1$ vertices connected by nested su and ju edges. The suffixes after the m^{th} positions of each graph are equal. The only difference between G_i and $G_{i,j}$ is that the i^{th} vertex of G_1 is labeled a while the two j^{th} vertices of $G_{i,j}$ are labeled a. All other vertices of each graph are labeled b. By the pigeonhole principle and the fact that A is finite-state, there must be a choice of i and j such that the deterministic run of A assigns the same right-state to the m^{th} vertex in each graph, hence to sink of each graph (since the suffixes are equal). Thus $G_i \in \text{SING}(a) \leftrightarrow G_{i,j} \in \text{SING}(a)$. But clearly $G_i \in \text{SING}(a)$ while $G_{i,j} \notin \text{SING}(a)$, a contradiction. □

Runs. Our definition of the *run* of an automaton annotates a left- and right-state at each vertex, rather than just a single state. This is convenient, but not essential; we could just as well label only one state at each vertex, as long as we are consistent about whether that state is prior to reading in the input letter at that vertex (left-state) or after (right-state). A second notable alternative would be to label edges, rather than vertices, with alphabet symbols Σ; however, this makes comparison with logical formalisms (MSO and graded μ-calculus) more cumbersome, as logics are usually expressed using vertex labels.

SSPGs as Trees. Finally, an SSPG with vertex labels drawn from some finite Σ can be translated to a directed rooted tree (with edge labels drawn from $\Gamma \setminus \{\text{sl}, \text{sr}, \text{su}\}$ and vertex labels drawn from Σ) obtained by removing all left-split, right-split, and unordered-split edges while leaving the remaining structure unchanged. Like the original SSPG, this resulting tree is over an unranked alphabet and exhibits unordered branching of unbounded arity. The above translation is an alternative to the standard notion of *unwinding* an acyclic graph to obtain a tree with a potentially exponential increase in size. For a comparison with existing tree automaton models, see Section 8. Although the split-edges are uniquely determined by the remaining structure of an SSPG, their explicit inclusion facilitates a natural flow of information during the run of an automaton.

3.5 Special Cases

SSPGs with restricted structure can model familiar objects including strings, multisets of strings, sequences of relations, and *nested words*. When we only consider SSPGs that model a particular structure, the *regular SSPG languages* (to be defined in Section 3.1) captures a natural notion of regularity for the underlying structure.

String. An SSPG constructed using only singleton vertices and series composition represents a string. Regular SSPG languages restricted to this structure coincide with the familiar regular languages of strings.

Multiset of Strings. The unordered parallel composition of 1 or more SSPGs each of which represents a string (as described above) represents a nonempty multiset of strings. The source and sink vertices are not used to represent the multiset, so we label them with a specially reserved placeholder $\# \in \Sigma$.

Relation. A *relation* (in the sense of relational algebra, i.e. a set of tuples) can be modeled as a multiset of strings where each string is a single character. Each character represents a tuple of one of finitely many types, where the label is the type of the tuple. Regular languages of such relations correspond to those definable in first-order logic over the language of equality: essentially expressing the existence or non-existence of a finite pattern of vertex labels.

Sequence of Relations. A *sequence of relations* is the classical model of a stream in traditional query processing literature [Arasu et al. 2003, 2006]. A sequence of relations in our setting is a series composition of some number of relations. It thus consists of multisets of Σ-labeled events separated by # marks in between.

Nested Word. An SSPG constructed using only singleton vertices, sequential composition, and 1-ary unordered parallel composition represents a *well-matched nested word* [Alur and Madhusudan 2009]. The synchronization edges of the SSPG correspond to the nesting edges, and all other edges of the SSPG correspond to the linear edges of the nested word. Unordered-split and unordered-join vertices correspond to *call* and *return* positions in the nested word, respectively. A language of SSPGs restricted to this form is regular iff it corresponds to a regular language of well-matched nested words.

4 APPLICATION TO DATA STREAMS

In this section, we describe a potential application of regular languages of SSPGs to express *temporal integrity constraints* on data streams. Three particular features of this connection are promising: SSPGs model the kinds of parallelism that arise in practice (Section 4.1); SSPGAs can be used to describe useful temporal integrity constraints that aren't expressible in existing type systems (Section 4.2); and decision procedures on SSPGAs can be reinterpreted as type-checking algorithms over temporal integrity constraints (Section 4.3). We also discuss limitations (Section 4.4).

4.1 SSPGs as Distributed Data Dtreams

Distributed data streams can be viewed as series-parallel structures. For example, consider a centralized system to monitor data from a distributed fleet of taxis. The events in the system consist of *begin ride* events `B` and *end ride* events `E`, representing the start time and close time for a ride at each individual taxi; and *punctuation marks* # which are placed at the start and at the end of each day. Punctuation marks act as global synchronization; after all taxi rides for the day complete, the distributed set of events is always closed by a # before the beginning of the next day.

We represent one possible input stream as the SSPG in Figure 2. In the graph, the first day comprises data streams from three taxis in parallel, containing 4, 2, and 1 events, respectively. The second day contains only a single end-ride event `E`, and the third day contains no events. Sequential composition is used to create the stream of events for a specific taxi on a specific day; sequential composition is used to combine data from different days; and unordered parallelism is used to combine streams from all of the different taxis within a single day. The stream can be thought of as a partial order on events: events are ordered within a specific taxi stream and ordered with respect to punctuation events.

Prior Work on Type Systems for Streams. A series-parallel semantics is inherent in any type system for streams that includes key-based parallelism (in this case, over taxi events) and synchronizing punctuation events. Some modern systems have more expressive type systems for describing how events are ordered in *nested parallel* contexts. For example, in Timely Dataflow [Murray et al. 2013, 2016], timestamps may be nested sequences, such as an event occurring in a given day, hour, and second written as (d, h, s); and events may be parallelized within each day, within each hour,

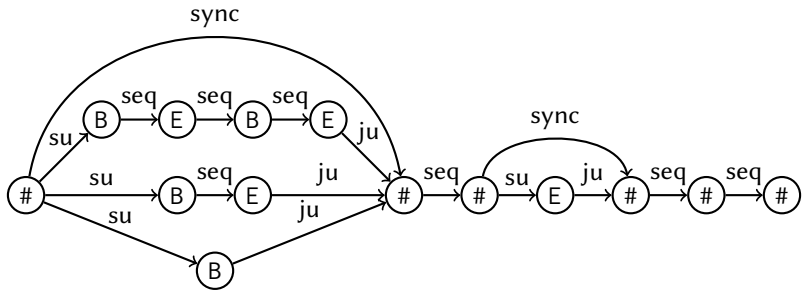


Fig. 2. A series-parallel stream of events from a distributed fleet of taxis: begin-ride events B, end-ride events E, and punctuation marks # marking the start and end of each day.

and within each second. Using synchronization schemas [Alur et al. 2021], which evolved from the earlier data-trace types proposal [Mamouras et al. 2019], the example in Figure 2 could be assigned a type such as `TaxiStream = Synch<#, KeyBy<id, Seq<B, E, G>>`.

4.2 SSPGAs as Temporal Integrity Constraints

With the SSPG view, we can combine a stream type, given using an existing type system, with a temporal integrity constraint on the stream, given as an SSPGA describing a regular language. A *temporal* constraint is one that describes the pattern of data that occurs over time.

Example 4.1. Define the property φ_1 that the begin- and end-ride events correspond for each taxi: the synchronizing events consist only of #, and for each key, the events of key `i` between consecutive # events match the regular expression `(B E)*`. The trace in Figure 2 violates φ_1 because the third parallel taxi stream on the first day consists of a B with no corresponding E.

We can write a nondeterministic SSPGA to monitor φ_1 using 4 states (a deterministic automaton would be similar, but require an additional reject state). The top-level state of the machine is q_0 indicating that no violation has been seen in the trace so far. On a split, we use the remaining two states q_B and q_E (indicating the next character expected) to match the regex `(B E)*`. On a join, we require that all joined vertices match the regex. The counting complexity for this property is 1. The formal transition relation is as follows:

- $\Delta_{\text{up}} = \{(q_0, \#, q_0), (q_\mathsf{B}, \mathsf{B}, q_\mathsf{E}), (q_\mathsf{E}, \mathsf{E}, q_\mathsf{B})\}$
- $\Delta_{\text{seq}} = \{(q_0, q_0), (q_\mathsf{B}, q_\mathsf{B}), (q_\mathsf{E}, q_\mathsf{E})\}$
- $\Delta_{\text{so}} = \Delta_{\text{jo}} = \{\}$
- $\Delta_{\text{su}} = \{(q_0, q_\mathsf{B})\}$
- $\Delta_{\text{ju}} \subseteq Q \times [Q \to [0..1]] \times Q$ contains exactly the triples (q_0, f, q_0) where $f(q_\mathsf{E}) = f(q_0) = 0$.

Example 4.2. Define φ_2 as the property that every event in the stream should be eventually followed by # (as opposed to a non-punctuation event), which is a necessary property for punctuation marks to ensure progress. The stream in Figure 2 satisfies this property because the final event is #. This property can be decided by an SSPGA with two states to remember the most recent event. The automaton has counting complexity 0; that is, it does not need to know which states have occurred or not before a join vertex.

Example 4.3. For an example with nontrivial counting complexity, define a taxi with 10 or more end-ride events in a given day to be *overloaded*, and let φ_3 be the property that there are at most 3 overloaded taxis in a given day. This property can be monitored with an SSPGA with counting complexity 3. The state set is used to record whether the property has been violated so far and to

count the number of end-ride events in a stream (maxing out at 10, requiring 11 states). On a join, the automaton checks whether there are at most 3 states that have reached a count of 10.

Example 4.4. Finally, for an example that uses the synchronization edge, define the property φ_4 that there are at least three overloaded taxis for two consecutive days (perhaps indicating the need to deploy more taxis). It is most convenient to express this using an SSPGA which is a small modification of the SSPGA for φ_3: we simply use the synchronization edge to directly recover whether or not the previous day was also overloaded. Note, however, that because nesting depth is bounded in this example, it is also possible to express φ_4 without the use of the synchronization edge, by passing this information to the children rather than saving it at the parent level (though this would require additional state complexity).

Existing type systems cannot describe temporal constraints like φ_1, φ_2, φ_3, and φ_4. In our experience, temporal constraints are common when working with streams in practice. For example, one might encounter a stream of aggregates such as S, C, #, S, C, #, …, where the S events are running sums and the C events are running counts; each occurs exactly once per punctuation event #. Now suppose we want to divide the sum and count to produce an average. This is difficult programmatically because the stream doesn't have the type information that S and C events actually come in adjacent pairs.

4.3 Decidability as Type Checking

Each property φ in the previous section can be seen as *refinement type* [Rondon et al. 2008]: given a stream type S in an existing type system, the refinement type $\{S \mid \varphi\}$ refines the type with a temporal constraint on the items in that stream. For example, a type such as

$$\{\texttt{Stream<\#, B, E, G>} \mid \varphi_1\}$$

describes the set of SSPGs of events #, B, E, and G satisfying the formula φ_1. From this perspective, decision procedures for regular SSPG languages are useful for type-checking. The *membership* problem is useful for monitoring: given a stream s (represented as an SSPG) of type S and an SSPGA A for φ, we can check whether s has type $\{S \mid \varphi\}$ by running A on input s. And more interestingly, if we are given s of type $\{S \mid \varphi\}$ (e.g., the output of one operator) and we want to check whether this is compatible with the input to another operator of type $\{S \mid \varphi'\}$, this amounts to checking inclusion between two regular SSPG languages: we check whether the language of an automaton for φ is included in the language of an automaton for φ'. This is a form of subtyping. Thanks to the robustness properties of the class of regular languages of SSPGs, these checks can be done efficiently and compositionally (see Section 6).

4.4 Limitations

Finally, there are useful properties that SSPGAs cannot describe. SSPGAs operate on the series-parallel structure with only finitely many labels; this means they cannot describe properties that work across taxis or that require knowing which taxi is which from one day to the next. For example, there is no way to express the property that the *same* taxi is overloaded from one day to the next, as this information is not present in the SSPG input. SSPGAs also cannot describe properties which require saving more than finite-state information from one day to the next, such as the following:

Example 4.5. Let φ_5 be the property that B and E events form matched pairs, when considered across days: each taxi stream within a day should match the regular expression $(\mathsf{E} \cup \epsilon)(\mathsf{BE})^*(\mathsf{B} \cup \epsilon)$, and the number of unclosed B events on each day must equal the number of starting E events on the following day. Although the trace in Figure 2 violates our initial constraint φ_1, it does satisfy φ_5. However, the set of SSPGs satisfying φ_5 is not regular.

5 AUTOMATA CONSTRUCTIONS

5.1 Determinization

Nondeterministic and deterministic SSPGA are equally expressive. The determinization procedure includes a subset construction over pairs of states to keep track of state assignments during all possible runs of a nondeterministic machine. This technique is borrowed from the determinization procedure for nested word automata [Alur and Madhusudan 2009]. Unordered joins pose a new challenge for determinization, and require increasing the counting complexity of the determinized machine.

Theorem 5.1 (Determinization). *Given an SSPGA A, one can effectively construct a DSSPGA B such that $L(A) = L(B)$. If A has n states and counting complexity k, then B has 2^{n^2} states and counting complexity nk.*

Proof. Let L be the language accepted by the nondeterministic SSPGA $A = (Q, \Sigma, Q_0, F, k, \Delta_{\text{up}}, \Delta_{\text{seq}}, \Delta_{\text{so}}, \Delta_{\text{su}}, \Delta_{\text{jo}}, \Delta_{\text{ju}})$.

The components of B are $(2^{Q\times Q}, \Sigma, Q_0', F', k', \delta_{\text{up}}', \delta_{\text{seq}}', \delta_{\text{so}}', \delta_{\text{su}}', \delta_{\text{jo}}', \delta_{\text{ju}}')$ described below.

Definitions and Overview. In an SSPG G, we define the *least linear predecessor* of a vertex v to be the most distant ancestor u of v such that there is a (possibly 0-length) path from u to v in G consisting solely of sequence- and synchronization-edges. Next we define a *summary* to be a pair of states drawn from Q. The states of B are sets of summaries.

We will design B such that *for any SSPG $G = (V, E, s, t, \Sigma, \sigma)$, there exists a run of A on G that assigns some $q \in Q$ as the left-state of s and assigns some $q' \in Q$ as the right-state of t if and only if there exists a run of B on G that assigns some right-state $S \in 2^{Q\times Q}$ to t such that $(q, q') \in S$*. We call the preceding specification of B the *summary invariant*.

The proof of the main theorem proceeds in three parts: first, we will define the state sets of B such that the summary invariant entails $L(A) = L(B)$. Second, we will define the transitions of B to satisfy the summary invariant. The summary invariant can be proved by induction on the SSPG structure; we omit the details of this proof. Finally, we will observe that the constructed B is deterministic.

State Sets and Acceptance. The states of B are sets of summaries. The initial state is the set of summaries that each have equal first and second components:

$$Q_0' = \{(q, q) \mid q \in Q\}$$

A determinized state $S \in 2^{Q\times Q}$ is accepting iff it contains a summary (q, q') such that $q \in Q_0$ and $q' \in F$:

$$F' = \{S \mid \exists (q, q') \in S \text{ s.t. } q \in Q_0 \wedge q' \in F\}$$

Assuming the summary invariant, such a summary will be an element of the right-state assigned to the sink of G by the deterministic run of B exactly when there is an accepting run of A on G. From this we obtain $L(A) = L(B)$.

Update, Sequence, and Split. The transition functions δ_{up}', and δ_{seq}' leave the first elements of their input summaries unchanged while updating the second components according to Δ_{up} or Δ_{seq}, respectively:

$$\delta_{\text{up}}'(S_s, a) = \{(q_s, q') \mid \exists q \text{ s.t. } (q_s, q) \in S,\ (q, a, q') \in \Delta_{\text{up}}\}$$
$$\delta_{\text{seq}}'(S) = \{(q_s, q') \mid \exists q \text{ s.t. } (q_s, q) \in S,\ (q, q') \in \Delta_{\text{seq}}\}$$

The transition functions δ_{so}', and δ_{su}' assign the set of all summaries with equal first and second components to the left position of each successor of an (ordered or unordered) split vertex. Note

that this does not make use of Δ_{so} and Δ_{su}; we defer checking the split relations of A until the join, while continuing to preserve the summary invariant.

$$\delta'_{\text{so}}(S) = \{(q,q) \mid q \in Q\} \times \{(q,q) \mid q \in Q\}$$
$$\delta'_{\text{su}}(S) = \{(q,q) \mid q \in Q\}$$

Ordered Join. The ordered join transition function receives inputs from a synchronization-predecessor u_s, a left-predecessor u_ℓ, and a right-predecessor u_r. The transition function δ'_{jo} inspects combinations of incoming summaries to see if the states assigned to the least-linear predecessors of u_ℓ and u_r could have been achieved by splitting some state that could be assigned to u_r according to Δ_{so}. If so, then those three summaries could correspond to a single run, and appropriate updated summaries are included in the output according to Δ_{jo} (with care to preserve the summary invariant):

$$\delta'_{\text{jo}}(S_s, S_\ell, S_r) = \{(q_s, q') \mid \exists q'_s, q_\ell, q'_\ell, q_r, q'_r \text{ s.t. } (q_s, q'_s) \in S_s, (q_\ell, q'_\ell) \in S_\ell, (q_r, q'_r) \in S_r \\ (q'_s, q_\ell, q_r) \in \Delta_{\text{so}}, (q'_s, q'_\ell, q'_r, q') \in \Delta_{\text{jo}}\}$$

Counting Complexity and Unordered Join. The unordered join transition function of B determines the set of summaries at an unordered join vertex u according to inputs from its synchronization predecessor u_s and a multiset M of inputs coming from its unordered join predecessors $u_1, \ldots, u_m$. B must k'-truncate M into a finite vector while preserving enough information to determine which summaries belong in the output.

Setting the counting complexity of B to $k' = kn$ (where $n = |Q|$) suffices. For B to decide which summaries to include in the left-state of v (according to the transition relations of the original machine A) requires collapsing M into the k-truncation of a multiset M' of elements drawn from Q. A state $S \in M$ is able to contribute multiplicity to one of 0 or more elements of M', depending on the summaries S contains. There can only be n unique elements of M', and each element can appear in $\lfloor M' \rfloor_k$ with multiplicity at most k, so for any S, $k' = kn$ appearances of S in M can contribute the full k multiplicity to each element of $\lfloor M' \rfloor_k$. That is, for any $k'' \geq k'$, k'' appearances of S can contribute exactly the same distributions of multiplicities as could k' appearances of S, so no information is lost by k'-truncating M.

B collapses its k'-truncated representation of M into M' by mapping each $S_i \in M$ to some $q'_i \in Q$. Such "collapses" are only legal if all of the q'_i could be achieved on a single run of A (that assigns some state some right-state $q'_s \in Q$ to u_s). For each q'_s that appears as the second element of a summary in the right-state that B assigned to u_s, B uses the summary invariant to only consider summaries in each element of $\lfloor M \rfloor_{k'}$ that have as their first component some q_i that could be achieved on an unordered split (according to Δ_{su}) from q'_s. B then uses the other elements of these summaries along with Δ_{ju} to decide which summaries to include in the output, taking care to preserve the summary invariant:

$$\delta'_{\text{ju}}(S_s, f) = \{(q_s, q') \mid \exists q'_s \in Q, \exists m \in [1..|2^{Q\times Q}|k'].\exists S_1, \ldots, S_m \in 2^{Q\times Q}. \\ \exists q_1, q'_1, \ldots, q_m, q'_m \in Q, \exists g : Q \to [0..k] \text{ s.t.} \\ (q_s, q'_s) \in S_s \wedge (\forall i \in [1..m], ((q'_s, q_i) \in \Delta_{\text{su}}) \wedge (q_i, q'_i) \in S_i) \wedge \\ f = \lfloor \{S_1, \ldots, S_m\} \rfloor_{k'} \wedge g = \lfloor \{q'_1, \ldots, q'_m\} \rfloor_k \wedge (q'_s, g, q') \in \Delta_{\text{ju}}\}$$

It suffices to only consider $m \leq |2^{Q\times Q}|k'$, because if the above condition were relaxed to allow m to be quantified over unrestricted $\mathbb{N}$, then for any choice of the existentially quantified variables that satisfies the condition, there would also exist a "small" choice with $m' \leq \max(|2^{Q\times Q}|k', m)$

that satisfies the condition. If the original choice used (determinized) states $S_1, \ldots S_m$ then a corresponding "small" choice would limit the multiplicity of each determinized state in $S_1, \ldots, S_m$ to k' (this preserves the ability to saturate the counting number of B). The choice of $q_1, q_1', \ldots, q_m, q_m'$ similarly can be "shrunk." The choices of q_s' and g need not change.

Determinism. Observe that $|Q_0'| = 1$. Each transition of B is deterministic (this follows from the correspondence between the function and relation notations for transitions described in Section 3.2). Thus B is deterministic. □

The above construction requires expanding the state size to maintain sets of pairs of states. The following theorem shows that this increase in complexity is tight.

Theorem 5.2 (State succinctness of nondeterminism). *There exists a family L_s of languages of SSPGs such that each L_s is accepted by a nondeterministic SSPGA with $O(s)$ states but every DSSPGA accepting L_s must have 2^{s^2} states.*

Proof. Under the correspondence between SSPGs and well-matched nested words described in section 3.5, the family of SSPG languages L_s corresponds to the family of nested word languages used to prove the analagous theorem for nested word automata. The proof proceeds similarly; see theorem 3.4 of Alur and Madhusudan [2009] for details. □

The determinization construction also increases counting complexity by a factor equal to the number of states of the original machine. We show this increase to be tight.

Theorem 5.3 (Counting succinctness of nondeterminism). *There exists a family L_s of SSPG languages of such that each L_s is accepted by a nondeterministic SSPGA with $O(s)$ states and constant counting complexity, but every DSSPGA accepting L_s must have counting complexity at least s.*

Proof. The family L_s is the same family L_s defined in example 3.8. Observe that for any s, the SSPGA B_s defined in example 3.9 accepts L_s using $O(s)$ states and $O(1)$ counting complexity.

For the sake of contradiction, assume there exists a DSSPGA C with counting complexity k such that $k \leq s$ that accepts L_s. Consider the graph $G_s \in L_s$ consisting of the unordered parallel composition of s vertices, and the graph $G_{s+1} \notin L_s$ consisting of the unordered parallel composition of $s+1$ vertices. Since C is deterministic, the sources of G_s and G_{s+1} must be assigned the same right-state q. Moreover, the s parallel vertices of G_s and the $s+1$ parallel vertices of G_{s+1} must all be assigned the same right-state q'. When applying the unordered-join transition function to determine the left-state of the sink, C k-truncates the incoming multiset of states into a count-vector (for $k \leq s$). This means that the s copies of q'' in G_s and the $s+1$ copies of q'' in G_{s+1} are indistinguishable, so G assigns the same left-state (and in turn, right-state) to the sink of each graph. Thus $G_s \in L(C) \leftrightarrow G_{s+1} \in L(C)$, a contradiction. □

Interestingly, any regular SSPG language can be recognized by a nondeterministic SSPGA with constant counting complexity. This is the focus of section 5.2.

5.2 Constant Counting Complexity

Theorem 5.4 (Constant counting complexity). *Given a DSSPGA A with state complexity n and counting complexity $k \geq 1$, one can effectively construct a nondeterministic SSPGA B with state complexity kn and counting complexity 2 such that $L(A) = L(B)$.*

Proof. Denote the components of the DSSPGA A as $(Q, \Sigma, Q_0, F, k, \Delta_{up}, \Delta_{\text{seq}}, \Delta_{\text{so}}, \Delta_{\text{su}}, \Delta_{\text{jo}}, \Delta_{\text{ju}})$. We will construct an equivalent nondeterministic SSPGA B with state set $Q \times (\{1\} \cup [1..k])$ and counting complexity 2 that simulates counting at unordered joins by including an *index* as the second component of each of its states.

Immediately preceding an unordered join, B's update relation nondeterministically assigns each of the unordered predecessors an index. If a rule in A's Δ_{ju} requires the incoming count-vector to include some state q with multiplicity $m < k$, then the corresponding rule of B requires the incoming states $(q,1),\ldots,(q,m)$ to each have multiplicity 1 (and requires $(q,m+1),\ldots,(q,k)$ to have multiplicity 0). If A's rule requires q with multiplicity k, then the corresponding rule of B requires the states $(q,1),\ldots(q,k)$ to each have multiplicity at least 1.

In all other situations, B ignores the index of its states and behaves analogously to A. Formally, let $B = (Q \times (\{1\} \cup [1..k]), \Sigma, Q_0 \times (\{1\} \cup [1..k]), F \times (\{1\} \cup [1..k]), 2, \Delta'_{\text{up}}, \Delta'_{\text{seq}}, \Delta'_{\text{so}}, \Delta'_{\text{su}}, \Delta'_{\text{jo}}, \Delta'_{\text{ju}})$ where the transition relations are

- $\Delta'_{up} = \{((q,i),a,(q',i')) \mid (q,a,q') \in \Delta_{up} \wedge i,i' \in (\{1\} \cup [1..k])\}$
- $\Delta'_{\text{seq}} = \{((q,i),(q',i')) \mid (q,q') \in \Delta_{seq} \wedge i,i' \in (\{1\} \cup [1..k])\}$
- $\Delta'_{\text{so}} = \{((q,i),(q_\ell,i_\ell),(q_r,i_r)) \mid (q,q_\ell,q_r) \in \Delta_{\text{so}} \wedge i,i_\ell,i_r \in (\{1\} \cup [1..k])\}$
- $\Delta'_{\text{su}} = \{((q,i),(q',i')) \mid (q,q') \in \Delta_{\text{su}} \wedge i,i' \in (\{1\} \cup [1..k])\}$
- $\Delta'_{\text{jo}} = \{((q_s,i_s),(q_\ell,i_\ell),(q_r,i_r),(q',i')) \mid (q_s,q_\ell,q_r,q') \in \Delta_{\text{jo}} \wedge i_s,i_\ell,i_r,i' \in (\{1\} \cup [1..k])\}$
- $\Delta'_{\text{ju}} = \{(q,g,q') \mid \exists f \in [Q \to [0..k]].(q,f,q') \in \Delta_{\text{ju}}.\forall q'' \in Q.$
  $(\forall m \in [f(q'')+1..k].g((q'',m)) = 0) \wedge$
  $(f(q'') = k \Rightarrow (\forall i \in (\{1\} \cup [1..k]).g((q'',i)) > 0)) \wedge$
  $(f(q'') < k \Rightarrow (\forall i \in [1..f(q'')].g((q'',i)) = 1))\}$

Let $G = (V,E,s,t,\Sigma,\sigma)$ be an arbitrary SSPG over Σ and assume w.l.o.g. that $V = \{u_1,\ldots,u_p\}$. We claim that for any $q^1_\ell, q^1_r, \ldots, q^p_\ell, q^p_r \in Q$, there exists a run of A that assigns the left- and right states q^i_ℓ and q^i_r (respectively) to each $u_i \in V$ iff there exists a choice of $j^1_\ell, j^1_r, \ldots, j^p_\ell, j^p_r \in (\{1\} \cup [1..k])$ such that there exists a run of B that assigns the left- and right-states (q^i_ℓ, j^i_ℓ) and (q^i_r, j^i_r) (respectively) to each $u_i \in V$. This subclaim can be proved by induction on the structure of G.

Combining the preceding subclaim with the fact that the initial and final state sets of B are the same as those of A modulo indexing, we find that there is an accepting run of A on G iff there is an accepting run of B on G. Since G may be an arbitrary SSPG over Σ, $L(A) = L(B)$.

□

The counting complexity 2 achieved by the construction above is the best possible, as shown below.

Theorem 5.5 (Minimum counting complexity). *There exists a regular SSPG language L such that any SSPGA A accepting L has counting complexity at least 2.*

Proof. For natural numbers s, let L_s and G_s be defined as in example 3.8. L_1 is clearly regular.

Assume, for the sake of contradiction, that there exists an SSPGA A with counting complexity 1 such that $L(A) = L_1$. There must be an accepting run of A on G_1 that assigns some right-state q to the source and some right-state q' to the central vertex.

On G_2, A can assign q to be the right-state of the source and q' to be the right-states of each of the two central vertices, as it did for G_1. Note that the 1-truncations of $\{q'\}$ and $\{q',q'\}$ are identical. Since the inputs to the unordered-join are the same during the accepting run on G_1 as for this run on G_2, $G_1 \in L(A) \leftrightarrow G_2 \in L(A)$, a contradiction. □

5.3 Closure Properties

The regular SSPG languages are closed under intersection, union, complement, language homomorphism, and reversal.

Theorem 5.6 (Boolean closure). *If L_1 and L_2 are regular SSPG languages over Σ, then $L_1 \cup L_2$, $L_1 \cap L_2$, and $SSPG(\Sigma) \setminus L_1$ are also regular SSPG languages.*

Proof. Let $A_i = (Q^i, \Sigma, Q^i_0, F^i, k^i, \Delta^i_{\text{up}}, \Delta^i_{\text{seq}}, \Delta^i_{\text{so}}, \Delta^i_{\text{su}}, \Delta^i_{\text{jo}}, \Delta^i_{\text{ju}})$ for $i = 1, 2$ be an SSPGA such that $L(A_i) = L_i$. Let $n_1 = |Q_i|$ and assume without loss of generality that $k^1 \geq k^2$.

Union and Intersection. Define the product of these machines as follows: the set of states is $Q^1 \times Q^2$ and the set of initial states is $\{(q_1, q_2) | q_1 \in Q^1_0 \wedge q_2 \in Q^2_0\}$. The counting complexity of the product machine is k^1. The $\Delta_{up}, \Delta_{seq}, \Delta_{\text{so}}, \Delta_{\text{su}}$, and Δ_{jo} transition relations are defined in the natural way. For example, the product machine's update transition relation is $\{((q_1, q_2), a, (q'_1, q'_2)) | (q_1, a, q'_1) \in \Delta^1_{\text{up}} \wedge (q_1, a, q'_1) \in \Delta^1_{\text{up}}\}$. The unordered-join transition relation must account for the fact that the counting complexity of A_2 may be less than that of the product machine, which motivates an additional step of further truncating the count-vector before checking membership in Δ^2_{ju}. The product-machine's unordered-join transition relation is thus

$$\{((q_1, q_2), f, (q'_1, q'_2)) | (q_1, f, q'_1) \in \Delta^1_{\text{ju}} \wedge (q_2, \lambda x. \min(k^2, f(x)), q'_2) \in \Delta^2_{\text{ju}}\}$$

Setting the set of accepting states to $F^1 \times F^2$ yields a product machine whose language is the intersection $L_1 \cap L_2$. Setting the set of accepting states to $(F^1 \times Q^2) \cup (Q^1 \times F^2)$ yields a machine whose language is the union $L_1 \cup L_2$. Each of these product machines has $n_1 n_2$ states, counting complexity k_1, and is deterministic if both A_1 and A_2 are deterministic.

Complement. To construct a machine whose language is the complement $SSPG(\Sigma) \setminus L_1$ of L_1, we first construct a DSSPGA that accepts L_1. If A_1 is deterministic, this is immediate, otherwise it requires the determinization procedure of theorem 5.1 and introduces the associated increases in state size and counting complexity. Complementing the the set of accepting states of this DSSPGA yields an automaton whose whose language is $SSPG(\Sigma) \setminus L_1$. If A_1 is deterministic, then this machine has n_1 states and counting complexity k_1. Otherwise, this machine has $2^{n_1^2}$ states and counting complexity $k_1 n_1$. □

Homomorphism. A language homomorphism h maps an arbitrary symbol $a \in \Sigma$ to an SSPG language $h(a)$ (possibly over a different finite alphabet). For an SSPG language L, we define $h(L)$ to be the set of SSPGs obtained by replacing each vertex u of some $G \in L$ with some SSPG G_u such that $G_u \in h(\sigma(u))$.

Theorem 5.7 (Homomorphism closure). *If L is a regular SSPG language over Σ, and h is a language homomorphism such that for each $a \in \Sigma$, $h(a)$ is regular, then $h(L)$ is also regular.*

Proof. This proof is similar that of the analogous theorem for nested word automata, with the addition of a technical detail to ensure that only subgraphs with valid SSPG structure (i.e., have matching splits and joins) can be considered as candidates for membership in $h(a)$ for some $a \in \Sigma$. See theorem 3.8 of Alur and Madhusudan [2009] for details. □

Reverse. The *reverse* operation inverts the direction of all edges in an SSPG, replacing split edges with join edges and vice-versa.

Definition 5.8 (Reverse)**.** For an edge label γ, define $reverse(\gamma)$ by $reverse(\text{so}) = \text{jo}$ and vice versa, $reverse(\text{su}) = \text{ju}$ and vice versa, and $reverse(\gamma) = \gamma$ otherwise. For an edge (u, γ, v), define $reverse((u, \gamma, v)) = (v, reverse(\gamma), u)$. Lift the reverse operation to set E of edges by $reverse(E) = \{reverse(e) \mid e \in E\}$. Given an SSPG $G = (V, E, s, t, \Sigma, \sigma)$ define the *reverse* of G as:

$$reverse(G) = (V, reverse(E), t, s, \Sigma, \sigma)$$

Finally, define the *reverse* of an SSPG language L as:

$$reverse(L) = \{reverse(G) \mid G \in L\}$$

Theorem 5.9 (Reverse closure). *Given an SSPGA A, one can effectively construct an SSPG B such that $L(B) = reverse(L(A))$.*

Proof idea. Recall the definition of *least linear predecessor*, introduced in the proof of Theorem 5.1. Given an automaton A, the idea is to define B such that for any $G = (V, E, s, t, \Sigma, \sigma)$ over Σ (assume w.l.o.g. that $V = \{u_1, \ldots, u_p\}$), there exists a run of A on G that assigns the left- and right-states q_ℓ^j and q_r^j (respectively) to $u_i \in V$ for each $i \in [1..p]$ iff there is a run of B on $reverse(G)$ that assigns the left- and right-states (q_ℓ^j, q_ℓ^i) and (q_r^j, q_r^i) to each $u_i \in V$ with least linear predecessor (in G) u_j (n.b. that u_j might not be the least linear predecessor of u_i in $reverse(G)$). We define B such that this subclaim can be proved by induction on the SSPG structure of G. □

Similarly to how regular languages of words are closed under concatenation, regular SSPG languages are closed under sequential, ordered-parallel, and unordered-parallel composition, as well as iterated sequential composition of an SSPG language with itself (Kleene-plus). We omit the associated constructions.

6 DECISION PROBLEMS

Membership, emptiness, and inclusion are decidable for regular SSPG languages. Table 1 summarizes complexity results for these problems. Throughout the section, we assume that membership and function application can be performed in constant time.

Theorem 6.1 (Membership). *For any SSPGA A with n states and SSPG G with m vertices and o edges, the membership problem $G \in L(A)$ is decidable $O(m + o)$ time when A is deterministic. If A is nondeterministic and has $|\Delta_{\mathsf{ju}}|$ unordered-join transitions, then membership is decidable in $O(m^{3.5} n^3 |\Delta_{\mathsf{ju}}|)$ time.*

Proof. Denote $A = (Q, \Sigma, Q_0, F, k, \Delta_{\mathsf{up}}, \Delta_{\mathsf{seq}}, \Delta_{\mathsf{so}}, \Delta_{\mathsf{su}}, \Delta_{\mathsf{jo}}, \Delta_{\mathsf{ju}})$ and let $n = |Q|$. Let G be an arbitrary SSPG G over Σ with m vertices and o edges.

*A **is deterministic.*** It suffices to construct the deterministic run of A on G, then check the acceptance condition. We process G in the same order as Kahn's topological sort algorithm, [Kahn 1962] performing constant-time additional work at each step to annotate each vertex with the left-state, right-state, and right-states of each of the vertex's predecessors (along with the type of predecessor) that would be assigned by the deterministic run of A on G. Whenever we add a vertex to the topologically-sorted list of vertices, we consult its "predecessor annotations" to determine which transition rule (or Q_0 if there are no predecessor annotations) should be used to determine the vertex's left-state q_ℓ and annotate the vertex's left-state accordingly. Then we annotate the right state $\delta_{up}(q_\ell, a)$ where $a \in \Sigma$ is the vertex's label. Since each edge of the graph adds exactly one such annotation, this matches the running time of classical topological sorting, $O(m + o)$ time.

*A **is nondeterministic.*** We will simulate the determinization procedure of theorem 5.1 on the fly. Correctness follows from the earlier proof; it suffices to show that the sets of summaries that would be the left- and right-states assigned to a vertex by a determinized machine can be constructed in polynomial time, assuming that the left- and right-sets of the vertex's predecessors have already been constructed.

We construct the right-set S' at a vertex labeled $a \in \Sigma$ that has been assigned the left-set S as follows. For each q' that appears as the second element of a summary in S, lookup the set of all q'' such that $(q', a, q'') \in \Delta_{\mathsf{up}}$, and for each such q'' and each q such that $(q, q') \in S$ insert (q, q'') in S'.

This requires $O(n)$ lookups and $O(n^2)$ insertions. The left-sets at internal, ordered split, unordered split, and ordered join vertices can similarly be constructed in polynomial time.

It remains to show that the left-set S' of an unordered-join vertex can be constructed in polynomial time given the right-sets of its predecessors. Let S_s denote the right-set of the synchronization-predecessor and $S_1, \ldots, S_p$ denote the right-sets of the p unordered join-predecessors. Deciding whether a transition rule $(q, f, q') \in \Delta_{\text{ju}}$ justifies adding a summary to S' requires considering the possible "collapses" of $S_1, \ldots, S_p$, as discussed in the proof of Theorem 5.1. This can be reduced to finding a complete matching in a bipartite graph with $2m$ vertices. The left node set of the bipartite graph will comprise p nodes representing each of $S_1, \ldots, S_p$. The left-node set will comprise nodes representing states of Q, with multiplicity determined by f. Edges exist between a node on the left and each state it could "collapse to," in such a way that there exists a perfect matching iff $S_1, \ldots, S_p$ could collapse to a $q_1, \ldots, q_p$ whose k-truncation is f. The construction proceeds as follows:

We will repeat the following procedure for each summary $(q_s, q_s') \in S_s$:

(1) For $i \in [1..p]$, define $T_i = \{q_i' \mid \exists q_i \text{ s.t. } (q_s', q_i) \in \Delta_{\text{su}} \wedge (q_i, q_i') \in S_i\}$. This filtering step ensures that only combinations of states achievable on the same run are considered.
(2) For each rule of the form $(q_s', f, q') \in \Delta_{\text{ju}}$, construct a bipartite graph with left node set X and right node set Y as follows:
   (a) Let X comprise $x_1, \ldots, x_p$.
   (b) For each $q'' \in Q$, add $f(q'')$ nodes to Y and add edges connecting each of them to all $x_i \in X$ such that $q'' \in T_i$.
   (c) If $|X| > |Y|$ then add $|X| - |Y|$ nodes to Y and add edges connecting each of them to all x_i such that T_i contains any q'' for which $f(q'') = k$.
   (d) If there exists a complete matching in the bipartite graph, then add (q_s, q') to S'.

At each iteration, the bipartite graph has $O(m)$ vertices and $O(m^2)$ edges, so the complete matching problem can be solved in $O(m^{2.5})$ time [Hopcroft and Karp 1973]. The entire above procedure to construct the left-set of an unordered-join vertex can thus be performed in $O(m^{2.5} n^3 |\Delta_{\text{ju}}|)$ time. This is the most computationally expensive kind of set to construct, so the entire on the fly simulation takes $O(m^{3.5} n^3 |\Delta_{\text{ju}}|)$ time. □

Theorem 6.2 (Emptiness). *Given an SSPGA A, checking emptiness of $L(A)$ is decidable in polynomial time.*

Proof. Denote $A = (Q, \Sigma, Q_0, F, k, \Delta_{\text{up}}, \Delta_{\text{seq}}, \Delta_{\text{so}}, \Delta_{\text{su}}, \Delta_{\text{jo}}, \Delta_{\text{ju}})$. We define a reachability relation $R \subseteq Q \times Q$ where $q \, R \, q'$ represents the statement "There exists an SSPG G such that there exists a run of A on G that assigns q as the left-state of the source vertex and q' as the right-state of the sink-vertex" or equivalently "q reaches q'." We initialize:

$$R = \{(q, q') \mid \exists a \in \Sigma. (q, a, q') \in \Delta_{up}\}$$

and proceed to apply the following rules (in any order) until a fixed-point is reached: *(Sequence)* For any $q, q', q'', q''' \in Q$ such that $q R q' \wedge (q', q'') \in \Delta_{\text{seq}} \wedge q'' R q'''$, add (q, q''') to R. *(Ordered parallelism)* For any $q, q', q'', q''', q_\ell, q_\ell', q_r, q_r' \in Q$ such that $\exists a \in \Sigma. (q, a, q') \in \Delta_{\text{up}} \wedge (q', q_\ell, q_r) \in \Delta_{\text{so}} \wedge q_\ell \, R \, q_\ell' \wedge q_r R q_r' \wedge (q', q_\ell', q_r', q'') \in \Delta_{\text{jo}} \wedge \exists a \in \Sigma. (q'', a, q''') \in \Delta_{\text{up}}$, add (q, q''') to R. *(Unordered parallelism)* For any $m \in [1..kn]$ and $q, q', q'', q_1, q_1', \ldots, q_m, q_m'$ such that $\exists a \in \Sigma. (q, a, q') \in \Delta_{\text{up}} \wedge \forall i \in [1..m]. ((q', q_i) \in \Delta_{\text{su}} \wedge q_i \, R \, q_i') \wedge (q', \lfloor \{q_1', \ldots, q_m'\} \rfloor_k, q'') \in \Delta_{\text{ju}} \wedge \exists a \in \Sigma. (q'', a, q''') \in \Delta_{\text{up}}$, add (q, q''') to R. The fixed-point will be reached after a polynomial number of iterations. L is empty iff no initial state $q_0 \in Q_0$ reaches any final state $q_f \in F$. □

Table 1. Complexity of SSPGA decision problems (under polytime reductions). Complexity is parameterized on SSPGA state complexity, counting complexity, as well as $|\Delta_{ju}|$ (and the size of the input SSPG in the case of membership).

	Membership	Emptiness	Inclusion
DSSPGA	Linear	PTime	PTime
SSPGA	PTime	PTime	ExpTime-complete

Theorem 6.3 (Inclusion and universality). *Given two SSPGAs A and B over an arbitrary finite alphabet Σ, the inclusion problem $L_A \subseteq L_B$ and the universality problem $L(B) = SSPG(\Sigma)$ are in PTime when B is deterministic and are ExpTime-complete in general.*

Proof. ***B is deterministic.*** Let B' be the DSSPGA obtained by complementing the set of final states of B. Observe that $L(B') = \overline{B}$. Construct the product of A and B' as described in the proof of theorem 5.6 to obtain a (potentially nondeterministic) machine accepting the language $L(A) \cap \overline{L(B)}$. The inclusion problem reduces to checking emptiness of the language of the product machine. As shown in Theorem 6.2, emptiness is decidable in polynomial time.

The universality problem reduces to checking inclusion of the language accepted by the 1-state SSPGA that accepts every SSPG in the language of B.

B is nondeterministic. Inclusion and universality are in ExpTime since we can first determinize B as in the proof of theorem 5.1 and proceed with the procedures described above for deterministic B.

To show ExpTime-hardness, we reduce the membership problem for alternating linear-space Turing machines to the universality problem for SSPGA. The proof proceeds similarly to the analogous theorem for nested word automata; see Theorem 6.2 of Alur and Madhusudan [2009]. □

7 LOGICAL CHARACTERIZATIONS

We now show that monadic second-order logic (MSO) and graded μ-calculus of SSPGs each characterize the regular SSPG languages.

7.1 MSO Characterization

In the MSO of SSPGs, first-order variables are interpreted over vertices and second-order variables are interpreted over sets of vertices. MSO over SSPGs includes an atomic formula for each $\gamma \in \Gamma$ to capture the *γ-labeled edge relation*, which holds between a vertex and its γ-successor.

Definition 7.1 (MSO over SSPGs)**.** Let Σ be a finite alphabet, let FV denote a countable set of first-order variables, and let SV denote a countable set of monadic second-order (set) variables. We use x, y, z, x', etc. to refer to elements of FV and X, Y, Z, X', etc. to refer to elements of SV.

The *monadic second-order logic of SSPGs* (over Σ) is given by

$$\phi := a(x) \mid X(x) \mid x \overset{\gamma}{\rightsquigarrow} y \mid \phi \vee \phi \mid \neg\phi \mid x = y \mid \exists x.\phi \mid \exists X.\phi$$

where $a \in \Sigma$, $x, y \in FV$, $X \in SV$, and $\gamma \in \Gamma$.

$a(x)$ holds when the label at the vertex interpreted for x is a. $x \overset{\gamma}{\rightsquigarrow} y$ holds when the vertex interpreted for y is a γ-successor of the vertex interpreted for y. $x = y$ holds when the same vertex is interpreted for each of x and y. An occurrence of a variable x or X in a MSO formula ϕ is *bound* if it occurs in a subformula $\exists x.\psi$ or $\exists X.\psi$ (respectively), and *free* otherwise. A *sentence* is a formula with no free occurrences of variables.

Simple examples of an MSO-definable properties include the *unlabeled edge* relation and its transitive closure, the *ancestor-of* relation, defined as follows:

$$x \rightsquigarrow y \coloneqq \vee_{\gamma \in \Gamma} (x \overset{\gamma}{\rightsquigarrow} y)$$
$$x \prec y \coloneqq \exists Z.Z(y).\forall z.(Z(z) \Rightarrow (x \rightsquigarrow z \vee \exists z'.Z(z').z' \rightsquigarrow z))$$

Definition 7.2 (Language of an MSO sentence)**.** For an MSO sentence ϕ over Σ, let $L(\phi)$ denote the set of all SSPGs over Σ that satisfy ϕ. Say that ϕ *defines* $L(\phi)$

For example, language SING(a) introduced in Example 3.7 is $L(\phi_{\text{SING}})$ where

$$\phi_{\text{SING}} = \exists x.a(x) \wedge \neg(\exists y.x \neq y \wedge a(y))$$

An SSPG language L is *MSO-definable* iff there exists an MSO sentence ϕ such that $L = L(\phi)$. We show that the class of MSO-definable languages is the regular SSPG languages.

Theorem 7.3 (MSO characterization). *A language L of SSPGs over Σ is regular iff there is an MSO sentence ϕ over Σ that defines L.*

Proof. The proof is similar in structure to the proof that MSO over nested words characterizes the regular nested word languages [Alur and Madhusudan 2009].

For any sentence ϕ, the set $L(\phi)$ of satisfying models is regular. We show the stronger claim that for any MSO formula ψ, the set $L(\psi)$ of satisfying models is regular.

Assume that in all formulas, each variable is quantified at most once. Fix the formula $\psi(x_1, ..., x_m, X_1, ..., X_o)$ with free variables $Z = \{x_1, ..., x_m, X_1, ..., X_o\}$. We construct an alphabet $\Sigma^Z \subseteq \Sigma \times [Z \to \{0, 1\}]$. An SSPG G' over Σ^Z encodes an SSPG G over Σ and a valuation for the free variables of ψ. Let $L(\psi)$ denote the SSPG language such that for any $G' \in L(\psi)$, the underlying SSPG G satisfies ψ under the valuations encoded by G'. We will show that it is possible to construct an SSPGA that accepts $L(\psi)$.

First, we describe how an SSPGA can check the property that each FV is assigned exactly once. Recall example 3.7, in which we construct the SSPGA A_{SING} that accepts the graphs in $SSPG(\{a, b\})$ that have exactly one vertex labeled a. For any FV x, we can adapt this construction to check that x is assigned exactly once. The new machine treats a symbol $(b, f) \in \Sigma^Z$ the way A_{SING} would treat a iff $f(x) = 1$. Regular SSPG languages are closed under product (theorem Theorem 5.6), so there exists an SSPGA that checks that all FVs are assigned once.

The proof now proceeds by induction on the structure of the MSO formula ψ. The atomic formulas $X(x)$, $a(x)$, and $x \overset{\gamma}{\rightsquigarrow} y$ can be checked using the finite control of an SSPGA.

For example, the atomic formula $x \overset{\text{sync}}{\rightsquigarrow} y$ can be checked by designing Δ_{up} such that the right state assigned of any vertex u includes an encoding of the vertex-label at u. The Δ_{jo} and Δ_{ju} relations enforce that at any join, if the synchronization predecessor is interpreted for x, then the left-state assigned to the join vertex encodes the requirement "this vertex is interpreted for y". Δ_{up} then checks that this requirement is met.

The cases for disjunction and negation follow from the fact that regular SSPG languages are closed under boolean operations (Theorem 5.6). The case for existential quantification follows from closure under language homomorphism (Theorem 5.7) because existential quantification corresponds to renaming the alphabet such that the valuation functions exclude a variable, which is a kind of language homomorphism.

For any SSPGA A, there is an equivalent MSO sentence. Let $A = (Q, \Sigma, Q_0, F, k, \Delta_{\text{up}}, \Delta_{\text{seq}}, \Delta_{\text{so}}, \Delta_{\text{su}}, \Delta_{\text{jo}}, \Delta_{\text{ju}})$ where $Q = \{q_1, ..., q_n\}$. Assume w.l.o.g. that A is a DSSPGA (Theorem 5.1). The corresponding MSO sentence ϕ holds on an SSPG G over Σ exactly when $G \in L(A)$. This sentence

is of the form $\exists X_\ell^1.\exists X_r^1.....\exists X_\ell^n.\exists X_r^n.\phi$ where X_ℓ^i represents the set of vertices assigned the left-state q_i and X_r^i represents the sets of vertices assigned the right-state q_i during the run of A on G.

The formula ϕ is a conjunction of conditions that must hold on an accepting run. For example, the condition that Δ_{up} holds at each vertex is be expressed as

$$\forall x. \bigvee_{(q_i,a,q_j)\in\Delta_{\text{up}}} (X_\ell^i(x) \wedge a(x) \wedge X_r^j(x))$$

One must also include conditions to enforce the other transition relations, assign an initial state to the source, and assign a final state to the sink. The most interesting condition is the one for Δ_{ju} given by

$$\forall x.\forall y.\forall z. x \overset{\text{sync}}{\rightsquigarrow} z \wedge y \overset{\text{ju}}{\rightsquigarrow} z \Rightarrow \bigvee_{(q_a,f,q_b)\in\Delta_{\text{ju}}} (X_r^a(x) \wedge \phi_1 \wedge \phi_2 \wedge X_\ell^b(z))$$

where the left side of the implication holds whenever x and z are a matching pair of unordered split and join vertices, and the right side holds whenever some rule in Δ_{ju} holds at y and its predecessors. The formula ϕ_1 ensures that for all $q_c \in Q$ such that $f(q_c) = k$, at least k distinct unordered join predecessors of z are assigned the right-state q_c:

$$\phi_1 = \bigwedge_{c:f(q_c)=k} \exists y_1 \cdots \exists y_k. \bigwedge_{i=1}^{k} \left(y_i \overset{\text{ju}}{\rightsquigarrow} z \wedge X_r^c(y_i) \wedge \bigwedge_{j\neq i} y_i \neq y_j \right)$$

The formula ϕ_2 ensures that for all $q_c \in Q$ such that $f(q_c) < k$, exactly $f(q_c)$ distinct q_c unordered join predecessors of z are assigned the right-state q_c:

$$\phi_2 = \bigwedge_{\{c|f(q_c)<k\}} \left(\exists y_1 \cdots \exists y_{f(q_c)}. \bigwedge_{i=1}^{f(q_c)} \left(y_i \overset{\text{ju}}{\rightsquigarrow} z \wedge X_r^c(y_i) \wedge \bigwedge_{j\neq i} y_i \neq y_j \right) \wedge \nexists y'.y' \overset{\text{ju}}{\rightsquigarrow} z \wedge \bigwedge_{i=1}^{f(q_c)} y' \neq y_i \right)$$

□

7.2 Graded μ-calculus Characterization

The *μ-calculus* [Kozen 1983] extends modal logic with least- and greatest-fixpoint operators, and is an important logic in model checking (see, e.g. Bradfield and Walukiewicz [2018]). A μ-calculus formula is interpreted over a *labeled transition systems (LTS)*, which is a potentially infinite directed graph in which vertices (*states*) are labeled with *propositions* and edges (*transitions*) are labeled with *actions*. We denote by

$$\mathcal{M} = (S, r, \{R_a\}_{a\in\text{Act}}, \{P_p\}_{p\in Prop})$$

the LTS $\mathcal{M}$ with state set S, distinguished root $r \in S$, action set Act, transition relations $R_a \subseteq S \times S$ for each action a, propositions set $Prop$, and set of satisfying states $P_p \subseteq S$ for each proposition p.

An SSPG $G = (V, E, s, t, \Sigma, \sigma)$ can be viewed as an LTS where V is the set of states, s is the root, Σ is the set of propositions, and Γ is the set of actions. Formally, we denote the elements of Σ as $\{p_1, p_2, \ldots, p_\ell\}$ and let the LTS representation of G be

$$\mathcal{M}_G = (V, s, \{R_\gamma\}_{\gamma\in\Gamma}, \{P_p\}_{p\in\Sigma})$$

where for each $\gamma \in \Gamma$, $R_\gamma := \{(u, v) \mid (u, \gamma, v) \in E\}$, and for each $a \in \Sigma$, $P_a := \{u \mid \sigma(u) = a\}$.

The μ-calculus can only express *bisimulation-invariant* properties in LTSs [Bradfield and Walukiewicz 2018]. Since the regular SSPG languages can express properties of SSPGs that are not bisimulation-invariant (e.g. "every unordered-split vertex has k successors"), μ-calculus cannot express all SSPGA-definable properties.

We will show that the extension of μ-calculus with *graded (counting) modalities* is expressively equivalent to the regular SSPG languages.

Definition 7.4 (Graded μ-calculus [Janin and Lenzi 2001; Kupferman et al. 2002])**.** Given a set *Prop* of atomic propositions, a set *Act* of actions, and a set *Var* of (propositional set) variables, the syntax of graded μ-calculus formulas is given by:

$$\phi := \top \mid \bot \mid p \mid \neg p \mid X \mid \phi_1 \wedge \phi_2 \mid \phi_1 \vee \phi_2 \mid \langle n, a\rangle \phi_1 \mid [n, a]\phi_1 \mid \mu X.\phi_1(X) \mid \nu X.\phi_1(X)$$

where $p \in Prop, X \in Var, a \in Act, \phi_1$ and ϕ_2 are graded μ-calculus formulas, and n is a non-negative integer. Note that the boolean constants $\top$ and $\bot$ can be derived from the remaining constructs.

A graded μ-calculus formula ϕ *counts up to* b if the largest integer occurring in graded modalities is $b-1$. We refer to b as the *counting bound* of ϕ. The counting bound of a graded μ-calculus formula is analogous to the counting complexity of an SSPGA.

A graded μ-calculus formula ϕ with variable set *Var* and a valuation $\mathcal{V} : Var \to 2^S$ is interpreted over a LTS $\mathcal{M} = (S, r, \{R_a\}_{a\in \text{Act}}, \{P_p\}_{p\in Prop})$ as a subset $[\![\phi]\!]^{\mathcal{M}}_{\mathcal{V}}$ of S defined inductively as follows [Kupferman et al. 2002]:

$$[\![X]\!]^{\mathcal{M}}_{\mathcal{V}} = \mathcal{V}(X)$$

$$[\![p]\!]^{\mathcal{M}}_{\mathcal{V}} = P_p \qquad [\![\neg p]\!]^{\mathcal{M}}_{\mathcal{V}} = V \setminus P_p$$

$$[\![\phi_1 \wedge \phi_2]\!]^{\mathcal{M}}_{\mathcal{V}} = [\![\phi_1]\!]^{\mathcal{M}}_{\mathcal{V}} \cap [\![\phi_2]\!]^{\mathcal{M}}_{\mathcal{V}} \qquad [\![\phi_1 \vee \phi_2]\!]^{\mathcal{M}}_{\mathcal{V}} = [\![\phi_1]\!]^{\mathcal{M}}_{\mathcal{V}} \cup [\![\phi_2]\!]^{\mathcal{M}}_{\mathcal{V}}$$

$$[\![\langle n, a\rangle \phi_1]\!]^{\mathcal{M}}_{\mathcal{V}} = \{u \in S \mid |\{v \mid (u, v) \in R_a \wedge v \in [\![\phi_1]\!]^{\mathcal{M}}_{\mathcal{V}}\}| > n\}$$

$$[\![[n, a]\phi_1]\!]^{\mathcal{M}}_{\mathcal{V}} = \{u \in S \mid |\{v \mid (u, v) \in R_a \wedge v \notin [\![\phi_1]\!]^{\mathcal{M}}_{\mathcal{V}}\}| \leq n\}$$

$$[\![\mu X.\phi_1]\!]^{\mathcal{M}}_{\mathcal{V}} = \bigcap \{S' \subseteq S \mid [\![\phi_1]\!]^{\mathcal{M}}_{\mathcal{V}[S'/X]} \subseteq S'\}$$

$$[\![\nu X.\phi_1]\!]^{\mathcal{M}}_{\mathcal{V}} = \bigcup \{S' \subseteq S \mid S' \subseteq [\![\phi_1]\!]^{\mathcal{M}}_{\mathcal{V}[S'/X]}\}$$

where $p \in Prop, X \in Var, a \in Act, \phi_1$ and ϕ_2 are graded μ-calculus formulas, and n is a non-negative integer. An occurrence of a variable $X \in Var$ is *bound* if it occurs in a subformula of the form $\mu X.\phi_1$ or $\nu X.\phi_1$ and *bound* otherwise. A *sentence* is a formula without any free occurrences of variables.

For a state $s \in S$, we write $\mathcal{M}, s, \mathcal{V} \vDash \phi$ when $s \in [\![\phi]\!]^{\mathcal{M}}_{\mathcal{V}}$. If ϕ is a sentence then we may omit the valuation and write $\mathcal{M}, s \vDash \phi$. We follow Janin and Lenzi [2001] in saying that LTS $\mathcal{M}$ with root r *satisfies* a sentence ϕ iff $\mathcal{M}, r \vDash \phi$. We equivalently write $\mathcal{M} \vDash \phi$.

Definition 7.5 (Language of graded μ-calculus sentence)**.** For a graded μ-calculus sentence ϕ, we define $L(\phi)$ to be the set of all SSPGs G such that $\mathcal{M}_G \vDash \phi$ for the LTS representation $\mathcal{M}_G$ of G. Say that ϕ *defines* $L(\phi)$.

An SSPG language L is *graded μ-calculus definable* iff there exists a graded μ-calculus sentence ϕ that defines L. We show that the class of graded μ-calculus-definable languages is the regular SSPG languages.

Lemma 7.6. *For any graded μ-calculus sentence ϕ, $L(\phi)$ is a regular SSPG language.*

Proof. Exploiting the MSO characterization of the regular SSPG languages (Theorem 7.3), it suffices to show that for any graded μ-calculus sentence ϕ there is an equivalent MSO sentence ψ such that $L(\phi) = L(\psi)$.

This follows from the fact that any graded μ-calculus formula ϕ in an LTS has the same meaning as some MSO formula [Janin and Lenzi 2001]. □

For the other direction, we show how to take an arbitrary SSPGA A and construct a graded μ-calculus formula ϕ such that $L(A) = L(\phi)$. The main challenge in the construction is that during a run of an SSPGA A on an SSPG G, determining a legal left- and right-state assignment at a vertex

(assuming knowledge of state assignments at its neighbors) may require knowledge of the state assignments at the vertex's predecessors and successors. Since graded μ-calculus has only forward modalities, we wish ϕ to be able to determine the set of legal state assignments with reference only to the assignments at the successors. Conveniently, determining state assignments at a vertex v if A is deterministic only requires reference to the predecessors of v, and the regular SSPG languages are closed under the *reverse* operation (Theorem 5.9). We will first construct a DSSPGA B such that $L(B) = \mathit{reverse}(L(A))$, then use B as a guide to define a graded μ-calculus formula whose language is $L(A)$.

It will be convenient to use the *vectorial syntax* for graded μ-calculus; thus far we have used the *scalar* syntax. We follow the presentation of Bradfield and Walukiewicz [2018] for vectorial μ-calculus, adjusting to include graded-modalities:

Definition 7.7 (Vectorial syntax)**.** A graded μ-calculus formula without fixpoint operators is a *modal formula*. If $\alpha_1, \ldots, \alpha_n$ are modal formulas, then $\boldsymbol{\alpha} = (\alpha_1, \ldots, \alpha_n)$ is a *vectorial graded μ-calculus formula of height n*. For a sequence of variables $\boldsymbol{X} = (X_1, \ldots, X_n)$ and a vectorial formula $\boldsymbol{\alpha}$ of height n, the formulas $\mu\boldsymbol{X}.\boldsymbol{\alpha}$ and $\nu\boldsymbol{X}.\boldsymbol{\alpha}$ are vectorial formulas of height n.

The meaning of a vectorial formula $\boldsymbol{\alpha} = (\alpha_1, \ldots, \alpha_n)$ in an LTS $\mathcal{M}$ under valuation $\mathcal{V}$ is $([\![\alpha_1]\!]^{\mathcal{M}}_{\mathcal{V}}, \ldots, [\![\alpha_n]\!]^{\mathcal{M}}_{\mathcal{V}})$. With the variables $\boldsymbol{X}$ distinguished, the meaning of $\boldsymbol{\alpha}$ is a function from $(2^S)^n$ to $(2^S)^n$ (where S is the state set of $\mathcal{M}$). Then the meanings of $\mu\boldsymbol{X}.\boldsymbol{\alpha}$ and $\nu\boldsymbol{X}.\boldsymbol{\alpha}$ are the least and greatest fixed-points of this function, respectively.

Proposition 7.8 (Scalar representation). *Any vectorial syntax graded μ-calculus formula of height n can be translated into n scalar graded μ-calculus formulas such that the i^{th} formula has the same meaning as the i^{th} coordinate of the original vectorial formula via iterated application of the Bekić principle [Bekić 1984].*

Lemma 7.9. *Given an SSPGA A, one can effectively construct a graded μ-calculus sentence ϕ such that $L(A) = L(\phi)$.*

Proof. Let $B = (Q, \Sigma, Q_0, F, k, \delta_{\mathsf{up}}, \delta_{\mathsf{seq}}, \delta_{\mathsf{so}}, \delta_{\mathsf{su}}, \delta_{\mathsf{jo}}, \delta_{\mathsf{ju}})$ be a DSSPGA such that $L(B) = \mathit{reverse}(L(A))$ (such an SSPGA B can be constructed from A, see Theorem 5.9 and Theorem 5.1). Let n denote $|Q|$ and assume w.l.o.g. that the elements of Q are $\{q_1, q_2, \ldots, q_n\}$.

We will first define a vectorial-syntax graded μ-calculus sentence ψ that encodes the initialization, acceptance, and transition conditions of B, then describe how to obtain a scalar-syntax graded μ-calculus sentence ϕ such that $L(\phi) = \mathit{reverse}(L(B)) = L(A)$.

The formula ψ will have variable set $\mathit{Var} = X_F \cup \bigcup_{i\in[1..n]}\{X^i_\ell, X^i_r\}$, proposition set $\mathit{Prop} = \Sigma$, action set $\mathit{Act} = \Gamma$, and a counting bound of k+1.

We will design ψ such that for any $G \in \mathrm{SSPG}(\Sigma)$ with LTS representation $\mathcal{M}_G$, X_F represents all vertices of G that are assigned an accepting right-state during the run of B on $\mathit{reverse}(G)$, and for any $i \in [1..n]$, X^i_ℓ and X^i_r represent the set of vertices of G that are assigned a the state q_i as a left- or right-state (respectively) during the run of B on $\mathit{reverse}(G)$.

Define the following modal formula to capture the set of vertices that can be assigned an accepting right-state during the run of B on $\mathit{reverse}(G)$:

$$\alpha_F = \bigvee_{i:q_i\in F} X^i_r$$

For each $i \in [1..n]$ define the following modal formula to allow vertices to be assigned the right-state q_i according to δ_{up} during the run of B on $\mathit{reverse}(G)$:

$$\alpha^i_r = \bigvee_{j:(q_j,q_i)\in\Delta_{\mathsf{up}}} (X^j_\ell \wedge a)$$

For each $i \in [1..n]$, define the modal formula β_0^i that allows any $q_i \in Q_0$ to be the left-state of the source and define formulas $\beta_{\mathsf{seq}}^i, \beta_{\mathsf{so}}^i, \beta_{\mathsf{su}}^i, \beta_{\mathsf{jo}}^i$, and β_{ju}^i to allow vertices to be assigned the left-state q_i according to the transitions of B during the run of B on $\mathit{reverse}(G)$:

$$\beta_0^i = \bigwedge_{\gamma \in \Gamma} [0,\gamma]\bot \qquad \beta_{\mathsf{seq}}^i = \bigvee_{j:\delta_{\mathsf{seq}}(q_j)=q_i} \langle 0,\mathsf{seq}\rangle X_r^j$$

$$\beta_{\mathsf{sl}}^i = \bigvee_{j:\mathit{fst}(\delta_{\mathsf{so}}(q_j))=q_i} \langle 0,\mathsf{jl}\rangle X_r^j \qquad \beta_{\mathsf{sr}}^i = \bigvee_{j:\mathit{snd}(\delta_{\mathsf{so}}(q_j))=q_i} \langle 0,\mathsf{jr}\rangle X_r^j \qquad \beta_{\mathsf{su}}^i = \bigvee_{j:\delta_{\mathsf{su}}(q_j)=q_i} \langle 0,\mathsf{ju}\rangle X_r^j$$

$$\beta_{\mathsf{jo}}^i = \bigvee_{\substack{x,y,z:\\ \delta_{\mathsf{jo}}(q_x,q_y,q_z)=q_i}} \left(\langle 0,\mathsf{sync}\rangle X_r^x \wedge \langle 0,\mathsf{sl}\rangle X_r^y \wedge \langle 0,\mathsf{sr}\rangle X_r^z\right)$$

$$\beta_{\mathsf{ju}}^i = \bigvee_{\substack{j,f:\\ \delta_{\mathsf{ju}}(q_j,f)=q_i}} \left(\langle 0,\mathsf{sync}\rangle X_r^j \wedge \left(\bigwedge_{x\in[1..n]} \langle f(q_x)-1,\mathsf{su}\rangle X_r^x\right) \wedge \left(\bigwedge_{\substack{x\in[1..n]:\\ f(q_x)<k}} [f(q_x),\mathsf{su}]\left(\bigvee_{\substack{y\in[1..n]:\\ y\neq x}} X_\ell^y\right)\right)\right)$$

Note that in the last formula we and may abuse notation and include subformulas of the form $\langle -1,\mathsf{su}\rangle\psi'$. Each such subformula should be read as $\top$.

Now, for the (sole) $i \in [1..n]$ such that $q_i \in Q_0$, define:

$$\alpha_r^i = \beta_0^i \vee \beta_{\mathsf{seq}}^i \vee \beta_{\mathsf{sl}}^i \vee \beta_{\mathsf{sr}}^i \vee \beta_{\mathsf{su}}^i \vee \beta_{\mathsf{jo}}^i \vee \beta_{\mathsf{ju}}^i$$

And for each $i \in [1..n]$ such that $q_i \notin Q_0$, define:

$$\alpha_r^i = \beta_{\mathsf{seq}}^i \vee \beta_{\mathsf{sl}}^i \vee \beta_{\mathsf{sr}}^i \vee \beta_{\mathsf{su}}^i \vee \beta_{\mathsf{jo}}^i \vee \beta_{\mathsf{ju}}^i$$

Finally, let $\boldsymbol{X} = (X_F, X_\ell^1, X_\ell^2, \ldots, X_\ell^n, X_r^1, X_r^2, \ldots, X_r^n)$, let $\boldsymbol{\alpha} = (\alpha_F, \alpha_\ell^1, \alpha_\ell^2, \ldots, \alpha_\ell^n, \alpha_r^1, \alpha_r^2, \ldots, \alpha_r^n)$, and let $\psi = \mu\boldsymbol{X}.\boldsymbol{\alpha}$.

For any $G \in \mathrm{SSPG}(\Sigma)$ with LTS representation $\mathcal{M}_G$ and any vertex u of G, let $B_\ell(u)$ and $B_r(u)$ denote the left- and right-state (respectively) assigned to u during the run of B on G. We claim that for any $G = (V, E, s, t, \Sigma, \sigma)$ with LTS representation $\mathcal{M}_G$ and any valuation $\mathcal{V}$,

$$\left([\![\psi]\!]_{\mathcal{V}}^{\mathcal{M}_G}\right)_i = \begin{cases} \{u \in V \mid B_r(u) \in F\} & i = 1 \\ \{u \in V \mid B_\ell(u) = q_i\} & i \in [2..n{+}1] \\ \{u \in V \mid B_r(u) = q_i\} & \mathit{otherwise} \end{cases}$$

The preceding claim can be proved by induction on the structure of the SSPG G. As a consequence of this claim, we obtain that for any $G \in \mathrm{SSPG}(\Sigma)$ and any valuation $\mathcal{V}$, the root of the LTS representation $\mathcal{M}_G$ is in the first component of $[\![\psi]\!]_{\mathcal{V}}^{\mathcal{M}}$ iff $G \in L(A)$. Following the procedure mentioned in Proposition 7.8, one can construct a scalar graded μ-calculus sentence ϕ equivalent to the first component of ψ. For this ϕ, we have that $G \in L(\phi) \leftrightarrow G \in L(A)$ hence $L(\phi) = L(A)$. □

The construction described above relies on the fact that B is deterministic. The formula $\boldsymbol{\alpha}$ merely encodes the conditions that must be true of any run of B on G, and exploits the fact that B is deterministic to ensure that the least fixed-point of $\boldsymbol{\alpha}$ represents an accepting run of B on G iff such a run exists. If B were not deterministic, an analogously defined $\boldsymbol{\alpha}$ would not suffice, and new techniques would be needed to aggregate the potentially many (or zero) runs of B on G. As a consequence of Lemma 7.6 and Lemma 7.9 we get the theorem:

Theorem 7.10 (graded μ-calculus characterization). *A language L of SSPGs over Σ is regular iff there is a graded μ-calculus sentence that defines L.*

8 RELATED WORK

Existing models of automata and logics can be classified based on the input structure. The studied structures include graphs, nested models and trees, Mazurkiewicz traces, and data words.

Graphs. Graph automata are most easily defined over directed acyclic graphs, or equivalently, over labeled partially ordered sets (generalizing beyond this is difficult, see [Bozapalidis and Kalampakas 2008; Reiter 2015]). Like our automata, acyclic graph automata proceed by visiting states in a topologically sorted order and labeling them based on the local neighborhood of predecessors [Kamimura and Slutzki 1981b; Thomas 1997]. Models in this setting typically assume that the incoming edges at each vertex are *ranked*, meaning that the set of predecessors is bounded and ordered [Arbib and Give'on 1968; Kamimura and Slutzki 1981a]. Series-parallel graphs with ordered parallelism (but no unordered parallelism) can be considered a special case for these models.

Lodaya and Weil [1998, 2000] define a model over series-parallel graphs (generalized to infinite graphs in [Kuske 2000]) with unranked unordered parallelism. Their nondeterministic automaton does not have counting, but instead may apply joins multiple times in a nondeterministic order at each join vertex. Each join is of finite arity (e.g., consume q_2 and two copies of q_3 to produce q_5). This notably allows for properties not definable in MSO, such as parity (e.g., number of states labeled q_2 is even). In other respects the model is similar to a standard graph automaton. This results in theory of regular languages with a number of characterized subclasses (notably, bounded-width languages), but no class is shown to have a robust characterization admitting determinization and equivalence to MSO.

Dimitrova and Majumdar [2018] study series parallel graphs using graph grammar transition systems, in which a context-free graph transformation system generating a family of series parallel graphs (with labeled directed edges) is paired with a concurrent finite automaton comprising finitely many two-way finite automata that communicate via boolean-valued registers at the graph's vertices. The graph transformation systems considered are edge replacement graph grammars that support only binary parallelism. More generally, no graph grammar that proceeds by edge replacement and has a finite number of production rules (see [Drewes et al. 1997]) can generate SSPGs with unbounded unordered parallelism.

Finally, note that the decidability of MSO over SSPGs follows from Courcelle's theorem, [Courcelle 1990] since any SSPG has bounded-tree width. The procedure described in Section 7.1 to convert an MSO formula into an equivalent SSPGA, which may then be used to check membership, is an alternative proof of this decidability result.

Nested Models and Trees. Hierarchical nesting is present in many existing theories of automata. A simple model of hierarchical nesting is given by nested words and nested word automata [Alur and Madhusudan 2009], which recognize the class of visibly pushdown languages [Alur and Madhusudan 2004]. Nested words are, up to syntactic differences, essentially equivalent to SSPGs where there is no ordered parallelism and unordered parallelism is only of arity $n = 1$.

Tree automata are one of the classical models in automata theory, originally introduced by Doner [1970] and more abstractly by Thatcher and Wright [1968] (for further reading, see [Baker 1978; Engelfriet 2015; Gécseg and Steinby 1997]). Tree automata have bottom-up and top-down variants. Tree automata typically work over ranked, ordered trees (that is, each node has a fixed number of ordered children). In this setting, traditional tree automata characterize the *regular tree languages* [Comon et al. 2008].

Since our SSPG model supports unranked, unordered branching, the generalization of tree automata to unranked, unordered trees [Carme et al. 2004; Cristau et al. 2005] is relevant to the design of our unordered join. Courcelle [Courcelle 1989] employs a general algebraic approach that works for various structures, including unordered trees, which requires the update function

to satisfy a commutativity property (see Definition 8, p. 117-118). A different approach is to use expressive logics for transition formulas based on the number of occurrences of states, often a variant of Presburger arithmetic [Dal Zilio and Lugiez 2003; Niehren and Podelski 1993]. Multitree automata [Lugiez 2005] are a representative model in this family; however, the expressive logical formulas go beyond MSO-definable properties, for example being able to express whether a node has an equal number of as and bs as children. To our knowledge, the study of counting complexity and its relation to determinism is new to our work.

The semantics of graded μ-calculus is often defined over trees [Bárcenas et al. 2015; Kupferman et al. 2002]. As shown in Section 7.2, when interpreted over SSPGs, graded μ-calculus is expressively equivalent to the regular SSPG languages. Over general labeled transition systems, non-graded μ-calculus is expressively equivalent to the bisimulation-invariant fragment of MSO [Janin and Walukiewicz 1996], and graded μ-calculus is expressively equivalent to the *unwinding-invariant* fragment of MSO [Janin and Lenzi 2001] (Theorem 3.1, attributed to [Walukiewicz 1996]).

Traces. Mazurkiewicz traces [Mazurkiewicz 1986] are a related model of a partially ordered set of events, and automata and logics have been studied over traces [Diekert and Métivier 1997; Diekert and Rozenberg 1995], giving rise to regular languages of traces. In trace theory, the ordering (edges) on elements is derived from the vertex labels via a *dependence relation* $D \subseteq \Sigma \times \Sigma$, whereas we study automata over graphs where the vertex labels are independent of the edges. Unlike traces, series-parallel graphs enforce a hierarchical structure. For example, the graph $\{1, 2, 3, 4\}$ with labels a, a, b, b and edges $1 \to 3, 1 \to 4, 2 \to 3, 2 \to 4$ is definable as a trace but cannot occur in a series-parallel graph.

Data Words. Finally, a *data word* is an element of $(\Sigma \times D)^*$, where Σ is a finite alphabet of labels and D is a potentially infinite data domain. There exist many formalisms to describe properties of strings over infinite alphabets (some of which treat the simpler case where words are drawn from D^*) [Alur and Černý 2011; Björklund and Schwentick 2010; Bojańczyk et al. 2006; Demri and Lazić 2009; Kaminski and Francez 1994; Neven et al. 2004; Shemesh and Francez 1994] (see [Segoufin 2006] for a survey). Data words are relevant for processing key-based parallel data: to a first approximation, the data word automata introduced by Bojańczyk et al. [2006] work in two passes: first, by relabeling the data word using a finite-state transducer, and second, checking whether the projection onto each key $d \in D$ present in the data word matches a regular language. One way to encode a data word as an SSPG is to combine different keys using unordered parallelism, as suggested by the example in Section 4 (where the keys are taxi IDs). Rigorously comparing and unifying these two approaches would be an interesting direction for future work.

ACKNOWLEDGMENTS

We would like to thank Mikołaj Bojańczyk and Alexander Rabinovich for helpful discussions, and the anonymous reviewers for their feedback. This material is based upon work supported by the National Science Foundation under award CCF 1763514.

REFERENCES

Rajeev Alur, Phillip Hilliard, Zachary G Ives, Konstantinos Kallas, Konstantinos Mamouras, Filip Niksic, Caleb Stanford, Val Tannen, and Anton Xue. 2021. Synchronization schemas. In *Proceedings of the 40th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*. 1–18. https://doi.org/10.1145/3452021.3458317

Rajeev Alur and Parthasarathy Madhusudan. 2004. Visibly pushdown languages. In *Proceedings of the thirty-sixth annual ACM symposium on Theory of computing*. 202–211. https://doi.org/10.1145/1007352.1007390

Rajeev Alur and P. Madhusudan. 2009. Adding Nesting Structure to Words. *J. ACM* 56, 3, Article 16 (may 2009), 43 pages. https://doi.org/10.1145/1516512.1516518

Rajeev Alur and Pavol Černý. 2011. Streaming Transducers for Algorithmic Verification of Single-Pass List-Processing Programs. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Austin, Texas, USA) *(POPL '11)*. Association for Computing Machinery, New York, NY, USA, 599–610. https://doi.org/10.1145/1926385.1926454

Roberto Amadini. 2021. A survey on string constraint solving. *ACM Computing Surveys (CSUR)* 55, 1 (2021), 1–38. https://doi.org/10.1145/3484198

Arvind Arasu, Shivnath Babu, and Jennifer Widom. 2003. CQL: A language for continuous queries over streams and relations. In *International Workshop on Database Programming Languages*. Springer, 1–19. https://doi.org/10.1007/978-3-540-24607-7_1

Arvind Arasu, Shivnath Babu, and Jennifer Widom. 2006. The CQL continuous query language: semantic foundations and query execution. *The VLDB Journal* 15, 2 (2006), 121–142. https://doi.org/10.1007/s00778-004-0147-z

Michael A Arbib and Yehoshafat Give'on. 1968. Algebra automata I: Parallel programming as a prolegomena to the categorical approach. *Information and Control* 12, 4 (1968), 331–345. https://doi.org/10.1016/S0019-9958(68)90374-4

Brian Babcock, Shivnath Babu, Mayur Datar, Rajeev Motwani, and Jennifer Widom. 2002. Models and issues in data stream systems. In *Proceedings of the twenty-first ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*. 1–16. https://doi.org/10.1145/543613.543615

Brenda S Baker. 1978. Tree transducers and tree languages. *Information and Control* 37, 3 (1978), 241–266. https://doi.org/10.1016/S0019-9958(78)90538-7

Everardo Bárcenas, Edgard Benítez-Guerrero, and Jesús Lavalle. 2015. On the Model Checking of the Graded μ-calculus on Trees. In *Mexican International Conference on Artificial Intelligence*. Springer, 178–189. https://doi.org/10.1007/978-3-319-27060-9_14

Hans Bekić. 1984. *Definable operations in general algebras, and the theory of automata and flowcharts*. Springer Berlin Heidelberg, Berlin, Heidelberg, 30–55. https://doi.org/10.1007/BFb0048939

Henrik Björklund and Thomas Schwentick. 2010. On notions of regularity for data languages. *Theoretical Computer Science* 411, 4-5 (2010), 702–715. https://doi.org/10.1007/978-3-540-74240-1_9

Mikołaj Bojańczyk, Anca Muscholl, Thomas Schwentick, Luc Segoufin, and Claire David. 2006. Two-variable logic on words with data. In *21st Annual IEEE Symposium on Logic in Computer Science (LICS'06)*. IEEE, 7–16. https://doi.org/10.1109/LICS.2006.51

Symeon Bozapalidis and Antonios Kalampakas. 2008. Graph automata. *Theoretical Computer Science* 393, 1-3 (2008), 147–165. https://doi.org/j.tcs.2007.11.022

Julian C. Bradfield and Igor Walukiewicz. 2018. The mu-calculus and Model Checking. In *Handbook of Model Checking*, Edmund M. Clarke, Thomas A. Henzinger, Helmut Veith, and Roderick Bloem (Eds.). Springer, 871–919. https://doi.org/10.1007/978-3-319-10575-8_26

Julien Carme, Joachim Niehren, and Marc Tommasi. 2004. Querying unranked trees with stepwise tree automata. In *International Conference on Rewriting Techniques and Applications*. Springer, 105–118. https://doi.org/10.1007/978-3-540-25979-4_8

Carl Chapman and Kathryn T Stolee. 2016. Exploring regular expression usage and context in Python. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*. 282–293. https://doi.org/10.1145/2931037.2931073

Hubert Comon, Max Dauchet, Rémi Gilleron, Florent Jacquemard, Denis Lugiez, Christof Löding, Sophie Tison, and Marc Tommasi. 2008. *Tree Automata Techniques and Applications*. 262 pages. https://hal.inria.fr/hal-03367725

Bruno Courcelle. 1989. On recognizable sets and tree automata. In *Algebraic Techniques*. Elsevier, 93–126. https://doi.org/10.1016/B978-0-12-046370-1.50009-7

Bruno Courcelle. 1990. The monadic second-order logic of graphs. I. Recognizable sets of finite graphs. *Information and Computation* 85, 1 (1990), 12–75. https://doi.org/10.1016/0890-5401(90)90043-H

Julien Cristau, Christof Löding, and Wolfgang Thomas. 2005. Deterministic automata on unranked trees. In *International Symposium on Fundamentals of Computation Theory*. Springer, 68–79. https://doi.org/10.1007/11537311_7

Silvano Dal Zilio and Denis Lugiez. 2003. XML schema, tree logic and sheaves automata. In *International Conference on Rewriting Techniques and Applications*. Springer, 246–263. https://doi.org/10.1007/s00200-006-0016-7

Loris D'Antoni and Margus Veanes. 2021. Automata modulo theories. *Commun. ACM* 64, 5 (2021), 86–95. https://doi.org/10.1145/3419404

Stéphane Demri and Ranko Lazić. 2009. LTL with the freeze quantifier and register automata. *ACM Transactions on Computational Logic (TOCL)* 10, 3 (2009), 1–30. https://doi.org/10.1145/1507244.1507246

Volker Diekert and Yves Métivier. 1997. Partial commutation and traces. In *Handbook of formal languages*. Springer, 457–533. https://doi.org/10.1007/978-3-642-59126-6_8

Volker Diekert and Grzegorz Rozenberg. 1995. *The book of traces*. World scientific. https://doi.org/10.1142/2563

Rayna Dimitrova and Rupak Majumdar. 2018. Reachability Analysis of Reversal-Bounded Automata on Series—Parallel Graphs. *Acta Inf.* 55, 2 (mar 2018), 153–189. https://doi.org/10.1007/s00236-016-0290-1

John Doner. 1970. Tree acceptors and some of their applications. *J. Comput. System Sci.* 4, 5 (1970), 406–451. https://doi.org/10.1016/S0022-0000(70)80041-1

Frank Drewes, Hans-Joerg Kreowski, and Annegret Habel. 1997. *Hyperedge replacement graph grammars*. 95–162. https://doi.org/10.1142/9789812384720_0002

Loris D'Antoni and Margus Veanes. 2017. The power of symbolic automata and transducers. In *International Conference on Computer Aided Verification*. Springer, 47–67. https://doi.org/10.1007/978-3-319-63387-9_3

Joost Engelfriet. 2015. Tree Automata and Tree Grammars. *CoRR* abs/1510.02036 (2015), 80 pages. arXiv:1510.02036 http://arxiv.org/abs/1510.02036

Ferenc Gécseg and Magnus Steinby. 1997. Tree languages. In *Handbook of formal languages*. Springer, 1–68. https://doi.org/10.1007/978-3-642-59126-6_1

Buğra Gedik. 2014. Partitioning functions for stateful data parallelism in stream processing. *The VLDB Journal* 23, 4 (2014), 517–539. https://doi.org/10.1007/s00778-013-0335-9

Sumit Gulwani. 2011. Automating string processing in spreadsheets using input-output examples. *ACM Sigplan Notices* 46, 1 (2011), 317–330. https://doi.org/10.1145/1926385.1926423

Hossein Hojjat, Philipp Rümmer, and Ali Shamakhi. 2019. On strings in software model checking. In *Asian Symposium on Programming Languages and Systems*. Springer, 19–30. https://doi.org/10.1007/978-3-030-34175-6_2

John E. Hopcroft and Richard M. Karp. 1973. An $n^{5/2}$ Algorithm for Maximum Matchings in Bipartite Graphs. *SIAM J. Comput.* 2, 4 (1973), 225–231. https://doi.org/10.1137/0202019 arXiv:https://doi.org/10.1137/0202019

D. Janin and G. Lenzi. 2001. Relating levels of the mu-calculus hierarchy and levels of the monadic hierarchy. In *Proceedings 16th Annual IEEE Symposium on Logic in Computer Science*. 347–356. https://doi.org/10.1109/LICS.2001.932510

David Janin and Igor Walukiewicz. 1996. On the expressive completeness of the propositional mu-calculus with respect to monadic second order logic. In *CONCUR '96: Concurrency Theory*, Ugo Montanari and Vladimiro Sassone (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 263–277. https://doi.org/10.1007/3-540-61604-7_60

A. B. Kahn. 1962. Topological Sorting of Large Networks. *Commun. ACM* 5, 11 (nov 1962), 558–562. https://doi.org/10.1145/368996.369025

Konstantinos Kallas, Filip Niksic, Caleb Stanford, and Rajeev Alur. 2022. Stream processing with dependency-guided synchronization. In *Proceedings of the 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 1–16. https://doi.org/10.1145/3503221.3508413

Tsutomu Kamimura and Giora Slutzki. 1981a. Parallel and two-way automata on directed ordered acyclic graphs. *Information and Control* 49, 1 (1981), 10–51. https://doi.org/10.1016/S0019-9958(81)90438-1

Tsutomu Kamimura and Giora Slutzki. 1981b. Transductions of dags and trees. *Mathematical systems theory* 15, 1 (1981), 225–249. https://doi.org/10.1007/BF01786981

Michael Kaminski and Nissim Francez. 1994. Finite-memory automata. *Theoretical Computer Science* 134, 2 (1994), 329–363. https://doi.org/10.1016/0304-3975(94)90242-9

Dexter Kozen. 1983. Results on the propositional μ-calculus. *Theoretical Computer Science* 27, 3 (1983), 333–354. https://doi.org/10.1016/0304-3975(82)90125-6 Special Issue Ninth International Colloquium on Automata, Languages and Programming (ICALP) Aarhus, Summer 1982.

Orna Kupferman, Ulrike Sattler, and Moshe Y Vardi. 2002. The complexity of the graded μ-calculus. In *International Conference on Automated Deduction*. Springer, 423–437. https://doi.org/10.1007/3-540-45620-1_34

Dietrich Kuske. 2000. Infinite series-parallel posets: logic and languages. In *International Colloquium on Automata, Languages, and Programming*. Springer, 648–662. https://doi.org/10.1007/3-540-45022-X_55

Kamal Lodaya and Pascal Weil. 1998. Series-parallel posets: algebra, automata and languages. In *Annual Symposium on Theoretical Aspects of Computer Science*. Springer, 555–565. https://doi.org/10.1007/BFb0028590

Kamal Lodaya and Pascal Weil. 2000. Series–parallel languages and the bounded-width property. *Theoretical Computer Science* 237, 1-2 (2000), 347–380. https://doi.org/10.1016/S0304-3975(00)00031-1

Denis Lugiez. 2005. Multitree automata that count. *Theoretical Computer Science* 333, 1-2 (2005), 225–263. https://doi.org/j.tcs.2004.10.023

Konstantinos Mamouras, Caleb Stanford, Rajeev Alur, Zachary G Ives, and Val Tannen. 2019. Data-trace types for distributed stream processing systems. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 670–685. https://doi.org/10.1145/3314221.3314580

Antoni Mazurkiewicz. 1986. Trace theory. In *Advanced course on Petri nets*. Springer, 278–324. https://doi.org/10.1007/3-540-17906-2_30

Derek G Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martín Abadi. 2013. Naiad: a timely dataflow system. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. 439–455. https://doi.org/10.1145/2517349.2522738

Derek G Murray, Frank McSherry, Michael Isard, Rebecca Isaacs, Paul Barham, and Martin Abadi. 2016. Incremental, iterative data processing with timely dataflow. *Commun. ACM* 59, 10 (2016), 75–83. https://doi.org/10.1145/2983551

Frank Neven, Thomas Schwentick, and Victor Vianu. 2004. Finite state machines for strings over infinite alphabets. *ACM Transactions on Computational Logic (TOCL)* 5, 3 (2004), 403–435. https://doi.org/10.1145/1013560.1013562

Joachim Niehren and Andreas Podelski. 1993. Feature automata and recognizable sets of feature trees. In *Colloquium on Trees in Algebra and Programming*. Springer, 356–375. https://doi.org/10.1007/3-540-56610-4_76

Fabian Reiter. 2015. Distributed graph automata. In *2015 30th Annual ACM/IEEE Symposium on Logic in Computer Science*. IEEE, 192–201. https://doi.org/10.1109/LICS.2015.27

Patrick M Rondon, Ming Kawaguci, and Ranjit Jhala. 2008. Liquid types. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 159–169. https://doi.org/10.1145/1375581.1375602

Luc Segoufin. 2006. Automata and logics for words and trees over an infinite alphabet. In *International Workshop on Computer Science Logic*. Springer, 41–57. https://doi.org/10.1007/11874683_3

Ambuj Shatdal and Jeffrey F Naughton. 1995. Adaptive parallel aggregation algorithms. *Acm Sigmod Record* 24, 2 (1995), 104–114. https://doi.org/10.1145/223784.223801

Yael Shemesh and Nissim Francez. 1994. Finite-state unification automata and relational languages. *Information and Computation* 114, 2 (1994), 192–213. https://doi.org/10.1006/inco.1994.1085

James W. Thatcher and Jesse B. Wright. 1968. Generalized finite automata theory with an application to a decision problem of second-order logic. *Mathematical systems theory* 2, 1 (1968), 57–81. https://doi.org/10.1007/BF01691346

Wolfgang Thomas. 1990. Infinite trees and automaton definable relations over ω-words. In *Annual Symposium on Theoretical Aspects of Computer Science*. Springer, 263–277. https://doi.org/10.1016/0304-3975(92)90090-3

Wolfgang Thomas. 1997. Elements of an automata theory over partial orders. In *Partial order methods in verification*. Vol. 29. American Mathematical Society, 25–40. https://doi.org/10.1090/dimacs/029/02

Peter A. Tucker, David Maier, Tim Sheard, and Leonidas Fegaras. 2003. Exploiting punctuation semantics in continuous data streams. *IEEE Transactions on Knowledge and Data Engineering* 15, 3 (2003), 555–568. https://doi.org/10.1109/TKDE.2003.1198390

Igor Walukiewicz. 1996. Monadic second order logic on tree-like structures. In *STACS 96*, Claude Puech and Rüdiger Reischuk (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 399–413.

Received 2022-07-07; accepted 2022-11-07