A General Framework for Sequential Grammars with Control Mechanisms

Rudolf Freund

A General Framework for Sequential Grammars with Control Mechanisms

Rudolf Freund

Faculty of Informatics, TU Wien
Favoritenstraße 9–11, 1040 Vienna, Austria
rudi@emcc.at

Abstract. Since more than five decades, many control mechanisms have been introduced for sequential string grammars, including control graphs, matrices, permitting and forbidden contexts, and order relations. These control mechanisms then have been extended to sequential grammars working on objects different from strings, for example, to array, graph, and multiset grammars. Many relations between the languages generated by sequential grammars working on these objects with different control mechanisms were shown to be similar to the relations already proved for the string case. Within a general framework for regulated rewriting based on the applicability of rules in sequential grammars, many relations between various control mechanisms can be established in a very general setting without any reference to the underlying objects the rules are working on. Besides the well-known control mechanisms as control graphs, matrices, permitting and forbidden rules, partial order on rules, and priority relations on rules, the new variants of activation of rules as well as activation and blocking of rules are considered. Special results for strings and multisets as well as for arrays in the general variant defined on Cayley grids of finitely presented groups are exhibited based on the general results. Finally, some general results for cooperating distributed grammar systems are established.

Keywords: general framework, regulating rewriting, sequential grammars

1 Introduction

Already thirty years ago, a first comprehensive overview on many concepts of regulated rewriting, especially for the string case, was given the monograph on regulated rewriting by Jürgen Dassow and Gheorghe Păun [7]. Yet as it turned out later, many of the mechanisms considered there for guiding the application of productions/rules can also be applied to other objects than strings, e.g., to n-dimensional arrays [9]. Even in the field of P systems [20] where mostly multisets are considered, such regulating mechanisms were used [4]. Using a general model for graph-controlled, programmed, random-context, and ordered grammars of arbitrary type based on the applicability of rules, many relations between various regulating mechanisms for sequential grammars can be established in a very

general setting without any reference to the underlying objects the rules are working on, as first exhibited in [12] in a comprehensive way. In this overview paper, the results elaborated in [12] are combined with the results obtained in the general framework for sequential grammars using activation and blocking of rules as introduced in [10], [3], and [2]. We recall special results for strings and multisets from [3] as well as results obtained in [10] for array grammars defined on Cayley grids of finitely presented groups. Finally, we establish some even new general results for cooperating distributed grammar systems.

In the following section, we recall some notions from formal language and group theory, especially for Cayley grids of finitely presented groups. In Section 3 we recall the main definitions of the general framework for sequential grammars of arbitrary type and the control mechanisms based on the applicability of rules as initiated in [12] and then continued in [10] and [3], i.e., for graph-controlled, programmed, random-context, and ordered grammars, for grammars with a priority relation on the rules, as well as for sequential grammars with activation and blocking of rules.

In Section 5 we summarize all the general results obtained within the framework for sequential grammars using the control mechanisms considered in this paper.

Specific results on computational completeness as well as some interesting complexity results for strings and multisets as underlying objects then are shown in Section 6.

In Section 7 we first define arrays and array grammars on Cayley grids of finitely presented groups. By proving that ordered array grammars using #-context-free array productions can generate the same language class as array grammars using arbitrary array productions, we then show that such a result not only holds for ordered array grammars but also for array grammars on Cayley grids of finitely presented groups equipped with many other control mechanisms, these results directly following from the general results summarized in Section 5 without needing any further proofs.

Finally, some general even new results for cooperating distributed grammar systems are elaborated in Section 8.

A summary of the results described in this paper and some future research topics conclude this overview paper.

2 Preliminaries

The set of integers is denoted by $\mathbb{Z}$, the set of positive integers by $\mathbb{N}$, the set of non-negative integers by $\mathbb{N}_0$. An *alphabet* V is a non-empty set of abstract *symbols*. Given V, the free monoid generated by V under the operation of concatenation is denoted by V^*; the elements of V^* are called strings, and the *empty string* is denoted by λ; $V^* \setminus \{\lambda\}$ is denoted by V^+. The cardinality of a set M is denoted by $|M|$.

Let $\{a_1, ..., a_n\}$ be an arbitrary alphabet; the number of occurrences of a symbol a_i in x is denoted by $|x|_{a_i}$; the *Parikh vector* associated with x with

respect to $a_1, ..., a_n$ is $\left(|x|_{a_1}, ..., |x|_{a_n}\right)$. The *Parikh image* of a language L over $\{a_1, ..., a_n\}$ is the set of all Parikh vectors of strings in L, and we denote it by $Ps(L)$. For a family of languages FL, the family of Parikh images of languages in FL is denoted by $PsFL$.

A finite multiset over the finite alphabet V, $V = \{a_1, ..., a_n\}$, is a mapping $f : V \longrightarrow \mathbb{N}_0$ and represented by $\langle f(a_1), a_1\rangle ... \langle f(a_n), a_n\rangle$ or by any string x the Parikh vector of which with respect to $a_1, ..., a_n$ is $(f(a_1), ..., f(a_n))$. In the following we will not distinguish between a vector $(m_1, ..., m_n)$, its representation by a multiset $\langle m_1, a_1\rangle ... \langle m_n, a_n\rangle$ or its representation by a string x having the Parikh vector $\left(|x|_{a_1}, ..., |x|_{a_n}\right) = (m_1, ..., m_n)$. Fixing the sequence of symbols $a_1, ..., a_n$ in the alphabet V in advance, the representation of the multiset $\langle m_1, a_1\rangle ... \langle m_n, a_n\rangle$ by the string $a_1^{m_1}...a_n^{m_n}$ is unique. The set of all finite multisets over an alphabet V is denoted by V°.

For the basic notions and results of formal language theory the reader is referred to the monographs and handbooks in this area as [7], [23], and [24], and for the basics of group theory and group presentations to [15]. The definitions and examples given in the following subsection are the basis for developing the theory of array grammars defined on Cayley grids of finitely presented groups in Section 7 (see [10]).

2.1 Groups and Group Presentations

Let $G = (G', \circ)$ be a group with group operation $\circ$. As is well-known, the group axioms are

- *closure*: for any $a, b \in G'$, $a \circ b \in G'$,
- *associativity*: for any $a, b, c \in G'$, $(a \circ b) \circ c = a \circ (b \circ c)$,
- *identity*: there exists a (unique) element $e \in G'$, called the *identity*, such that $e \circ a = a \circ e$ for all $a \in G'$, and
- *invertibility:* for any $a \in G'$, there exists a (unique) element a^{-1}, called the *inverse* of a, such that $a \circ a^{-1} = a^{-1} \circ a = e$.

In the following, we will not distinguish between G' and G if the group operation is obvious from the context. A group is called *commutative* (*Abelian*), if for any $a, b \in G'$, $a \circ b = b \circ a$. For any element $b \in G'$, the order of b is the smallest number $n \in \mathbb{N}$ such that $b^n = e$ provided such an n exists, and then we write $ord(b) = n$; if no such n exists, $\{b^n \mid n \geq 1\}$ is an infinite subset of G' and we write $ord(b) = \infty$.

For any set B, B^{-1} is defined as the set of symbols representing the inverses of the elements of B, i.e., $B^{-1} = \left\{b^{-1} \mid b \in B\right\}$. We now consider the strings in $\left(B \cup B^{-1}\right)^*$ and two strings as different unless their equality follows from the group axioms, i.e., for any $a, b, c \in \left(B \cup B^{-1}\right)^*$, $abb^{-1}c = ac$; using these reductions, we obtain a set of irreducible strings from those in $\left(B \cup B^{-1}\right)^*$, the set of which we denote by $I(B)$. Then the *free group* generated by B is $F(B) = (I(B), \circ)$ with the elements being the irreducible strings over $B \cup B^{-1}$ and the group operation to be interpreted as the usual string concatenation,

yet, obviously, if we concatenate two elements from $I(B)$, the resulting string eventually has to be reduced again. The identity in $F(B)$ is the empty string.

In general, B (not containing the identity) is called a *generator* of the group G if every element a from G can be written as a finite product/sum of elements from B and its inverses from B^{-1}, i.e., $a = b_1 \circ \cdots \circ b_m$ for $b_1, \ldots, b_m \in B \cup B^{-1}$. In this paper, we restrict ourselves to finitely presented groups, i.e., having a finite presentation $\langle B \mid R \rangle$ with B being a finite generator set and moreover, R being a finite set of relations among the elements of $B \cup B^{-1}$. In a similar way as in the definition of the free group generated by B, we here consider the strings in $(B \cup B^{-1})^*$ reduced according to the group axioms and the relations given in R. Informally, the group $G = \langle B \mid R \rangle$ is the largest one generated by B subject only to the group axioms and the relations in R. Formally, we will restrict ourselves to relations of the form $b_1 \circ \cdots \circ b_m = c^{-1}$ with $b_1, \ldots, b_m, c \in B \cup B^{-1}$, which equivalently may be written as $b_1 \circ \cdots \circ b_m \circ c = e$; hence, instead of such relations we may specify R by strings over $B \cup B^{-1}$ yielding the group identity, i.e., instead of $b_1 \circ \cdots \circ b_m = c^{-1}$ we take $b_1 \circ \cdots \circ b_m \circ c$ (these strings then are called *relators*).

Example 1. The free group $F(B) = (I(B), \circ)$ can be written as $\langle B \mid \emptyset \rangle$ (or even simpler as $\langle B \rangle$) because it has no restricting relations.

Example 2. The *cyclic group* of order n has the presentation $\langle \{a\} \mid \{a^n\} \rangle$ (or, omitting the set brackets, written as $\langle a \mid a^n \rangle$); it is also known as $\mathbb{Z}_n$ or as the quotient group $\mathbb{Z}/n\mathbb{Z}$.

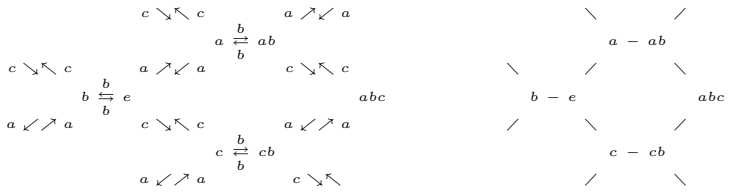
Example 3. $\mathbb{Z}$ is a special case of an Abelian group generated by (1) and its inverse (-1), i.e., $\mathbb{Z}$ is the free group generated by (1). $\mathbb{Z}^d$ is an Abelian group generated by the unit vectors $(0, ..., 1, ..., 0)$ and their inverses $(0, ..., -1, ..., 0)$. It is well known that every finitely generated Abelian group is a direct sum of a torsion group and a free Abelian group where the torsion group may be written as a direct sum of finitely many groups of the form $\mathbb{Z}/p^k\mathbb{Z}$ for p being a prime, and the free Abelian group is a direct sum of finitely many copies of $\mathbb{Z}$.

Remark 1. Given a finite presentation of a group $\langle B \mid R \rangle$, in general it is not even decidable whether the group presented in that way is finite or infinite. If we consider (infinite) groups where the word equivalence problem $u = v$ is decidable, or equivalently, there is a decision procedure telling us whether, given two strings u and v, $uv^{-1} = e$, then we call $\langle B \mid R \rangle$ a *recursive* or *computable* finite group presentation.

2.2 Cayley Graphs

Let $G = \langle B \mid R \rangle$ be a finitely presented group with G' denoting the set of group elements. Then we define the corresponding *Cayley graph* (*Cayley grid*) of G with respect to the generating set B as the directed graph $C(G, B) = (G', E)$ with the set of nodes G' and the set E of directed edges labeled by elements of B by $E = \{(x, a, y) \mid x, y \in G', a \in B, xa = y\}$, i.e., from an element x an edge labeled by the generator a leads to y if and only if $xa = y$.

Example 4. The hexagonal grid is the Cayley graph assigned to the presentation of the group $\langle a, b, c \mid a^2, b^2, c^2, (abc)^2 \rangle$. As all three generators a, b, c are self-inverse and the direction of these elements indicates which generator is meant, we obtain a simpler picture for the hexagonal grid by replacing $a \nearrow\!\!\swarrow a$, $\overset{b}{\rightleftarrows}$, and $c \searrow\!\!\nwarrow c$ by $/$, $-$, and $\backslash$, respectively. Both representations are depicted in the following:



2.3 Register Machines

As a computationally complete model able to generate/accept all sets in $PsRE = Ps(\mathcal{L}(ARB))$ we use register machines/deterministic register machines:

A *register machine* is a construct $M = (n, L_M, R_M, p_0, h)$ where n, $n \geq 1$, is the number of registers, L_M is the set of instruction labels, p_0 is the start label, h is the halting label (only used for the `HALT` instruction), and R_M is a set of (labeled) instructions being of one of the following forms:

- $p : (\texttt{ADD}(r), q, s)$ increments the value in register r and continues with the instruction labeled by q or s,
- $p : (\texttt{SUB}(r), q, s)$ decrements the value in register r and continues the computation with the instruction labeled by q if the register was non-empty, otherwise it continues with the instruction labeled by s;
- $h : \texttt{HALT}$ halts the machine.

M is called deterministic if in all ADD-instructions $p : (\texttt{ADD}(r), q, s)$ $q = s$; in this case we write $p : (\texttt{ADD}(r), q)$. Deterministic register machines can accept all recursively enumerable sets of vectors of natural numbers with k components using exactly $k + 2$ registers, for instance, see [17].

3 A General Model for Sequential Grammars and Regulated Rewriting Based on the Applicability of Rules

In this section we recall the notions for the general model of sequential grammars equipped with specific control mechanisms based on the applicability of rules as elaborated in [12] and in [3].

We first recall the main definitions of the general model for sequential grammars as established in [12], grammars generating a set of terminal objects by

derivations where in each derivation step exactly one rule is applied to exactly one object.

A *(sequential) grammar* G_s is a construct $(O, O_T, w, P, \Longrightarrow_{G_s})$ where

- O is a set of *objects*;
- $O_T \subseteq O$ is a set of *terminal objects*;
- $w \in O$ is the *axiom (start object)*;
- P is a finite set of *rules*;
- $\Longrightarrow_{G_s} \subseteq O \times O$ is the *derivation relation* of G_s.
  Each of the rules $p \in P$ induces a relation $\Longrightarrow_p \subseteq O \times O$ with respect to $\Longrightarrow_{G_s}$. A rule $p \in P$ is called *applicable* to an object $x \in O$ if and only if there exists at least one object $y \in O$ such that $(x, y) \in \Longrightarrow_p$; we also write $x \Longrightarrow_p y$. The derivation relation $\Longrightarrow_{G_s}$ is the union of all $\Longrightarrow_p$, i.e., $\Longrightarrow_{G_s} = \cup_{p \in P} \Longrightarrow_p$. The reflexive and transitive closure of $\Longrightarrow_{G_s}$ is denoted by $\overset{*}{\Longrightarrow}_{G_s}$.

Specific conditions on the rules in P define a special type X of grammars which then will be called *grammars of type* X.

The *language generated by* G is the set of all terminal objects that can be derived from the axiom, i.e.,

$$L(G_s) = \left\{ v \in O_T \mid w \overset{*}{\Longrightarrow}_{G_s} v \right\}.$$

The family of languages generated by grammars of type X is denoted by $\mathcal{L}(X)$.

Let $G_s = (O, O_T, w, P, \Longrightarrow_{G_s})$ be a (sequential) grammar of type X. If for every G_s of type X we have $O_T = O$, then X is called a *pure* type, otherwise it is called *extended*; X is called *strictly extended* if for any grammar G_s of type X, $w \notin O_T$ and for all $x \in O_T$, no rule from P can be applied to x.

In many cases, the type X of the grammar allows for one or even both of the following features:

A type X of grammars is called a *type with unit rules* if for every grammar $G_s = (O, O_T, w, P, \Longrightarrow_G)$ of type X there exists a grammar $G'_s = \left(O, O_T, w, P \cup P^{(+)}, \Longrightarrow_{G'_s}\right)$ of type X such that $\Longrightarrow_{G_s} \subseteq \Longrightarrow_{G'_s}$ and

- $P^{(+)} = \left\{p^{(+)} \mid p \in P\right\}$,
- for all $x \in O$, $p^{(+)}$ is applicable to x if and only if p is applicable to x, and
- for all $x \in O$, if $p^{(+)}$ is applicable to x, the application of $p^{(+)}$ to x yields x back again.

A type X of grammars is called a *type with trap rules* if for every grammar $G_s = (O, O_T, w, P, \Longrightarrow_G)$ of type X there exists a grammar $G'_s = \left(O, O_T, w, P \cup P^{(-)}, \Longrightarrow_{G'_s}\right)$ of type X such that $\Longrightarrow_{G_s} \subseteq \Longrightarrow_{G'_s}$ and

- $P^{(-)} = \left\{p^{(-)} \mid p \in P\right\}$, $P^{(-)} \cap P = \emptyset$;
- for all $x \in O$, $p^{(-)}$ is applicable to x if and only if p is applicable to x, and
- for all $x \in O$, if $p^{(-)}$ is applicable to x, the application of $p^{(-)}$ to x yields an object y from which no terminal object can be derived anymore.

3.1 Graph-controlled and Programmed Grammars

A *graph-controlled grammar* (with applicability checking) of type X is a construct

$$G_{GC} = (G_s, g, H_i, H_f, \Longrightarrow_{GC})$$

where $G_s = (O, O_T, w, P, \Longrightarrow_G)$ is a grammar of type X; $g = (H, E, K)$ is a labeled graph where H is the set of node labels identifying the nodes of the graph in a one-to-one manner, $E \subseteq H \times \{Y, N\} \times H$ is the set of edges labeled by Y or N, $K : H \to 2^P$ is a function assigning a subset of P to each node of g; $H_i \subseteq H$ is the set of initial labels, and $H_f \subseteq H$ is the set of final labels. The derivation relation $\Longrightarrow_{GC}$ is defined based on $\Longrightarrow_{G_s}$ and the control graph g as follows: For any $i, j \in H$ and any $u, v \in O$, $(u, i) \Longrightarrow_{GC} (v, j)$ if and only if

- $u \Longrightarrow_p v$ by some rule $p \in K(i)$ and $(i, Y, j) \in E$ *(success case)*, **or**
- $u = v$, no $p \in K(i)$ is applicable to u, and $(i, N, j) \in E$ *(failure case)*.

The language generated by G_{GC} is defined by

$$L(G_{GC}) = \left\{ v \in O_T \mid (w, i) \Longrightarrow^*_{G_{GC}} (v, j),\ i \in H_i, j \in H_f \right\}.$$

If $H_i = H_f = H$, then G_{GC} is called a *programmed grammar*. The families of languages generated by graph-controlled and programmed grammars of type X are denoted by $\mathcal{L}(X\text{-}GC_{ac})$ and $\mathcal{L}(X\text{-}P_{ac})$, respectively. If the set E contains no edges of the form (i, N, j), then the graph-controlled grammar is said to be *without applicability checking*; the corresponding families of languages are denoted by $\mathcal{L}(X\text{-}GC)$ and $\mathcal{L}(X\text{-}P)$, respectively.

As a special variant of graph-controlled grammars we consider those where all labels are final; the corresponding family of languages generated by graph-controlled grammars of type X is abbreviated by $\mathcal{L}\left(X\text{-}GC_{ac}^{allfinal}\right)$. By definition, programmed grammars are just a subvariant where in addition all labels are also initial.

The notions *with/without applicability checking* in the original definition for string grammars were introduced as *with/without appearance checking* because the appearance of the non-terminal symbol on the left-hand side of a context-free rule was checked, which coincides with checking for the applicability of this rule in our general model; in both cases – applicability checking and appearance checking – we can use the abbreviation *ac*.

3.2 Matrix Grammars

A *matrix grammar* (with applicability checking) of type X is a construct $G_M = (G_s, M, F, \Longrightarrow_{G_M})$ where $G_s = (O, O_T, w, P, \Longrightarrow_G)$ is a grammar of type X, M is a finite set of sequences of the form $(p_1, \ldots, p_n)$, $n \geq 1$, of rules in P, and $F \subseteq P$. For $w, z \in O$ we write $w \Longrightarrow_{G_M} z$ if there are a matrix $(p_1, \ldots, p_n)$ in M and objects $w_i \in O$, $1 \leq i \leq n+1$, such that $w = w_1$, $z = w_{n+1}$, and, for all $1 \leq i \leq n$, either

– $w_i \Longrightarrow_{p_i} w_{i+1}$ or
– $w_i = w_{i+1}$, p_i is not applicable to w_i, and $p_i \in F$.

$L(G_M) = \left\{v \in O_T \mid w \Longrightarrow^*_{G_M} v\right\}$ is the language generated by G_M. The family of languages generated by matrix grammars of type X is denoted by $\mathcal{L}(X\text{-}MAT_{ac})$. If the set F is empty, then the grammar is said to be *without applicability checking* (*without ac* for short); the corresponding family of languages is denoted by $\mathcal{L}(X\text{-}MAT)$. We mention that in this paper we choose the definition where the sequential application of the rules in the final matrix may stop at any moment.

3.3 Random-Context Grammars

A *random-context grammar* G_{RC} *of type* X is a construct $(G_s, P', \Longrightarrow_{G_{RC}})$ where

– $G_s = (O, O_T, w, P, \Longrightarrow_G)$ is a grammar of type X;
– P' is a set of rules of the form (p, R, Q) where $p \in P$, $R \cup Q \subseteq P$;
– $\Longrightarrow_{G_{RC}}$ is the derivation relation assigned to G_{RC} such that for any $x, y \in O$, $x \Longrightarrow_{G_{RC}} y$ if and only if for some rule $(p, R, Q) \in P'$, $x \Longrightarrow_p y$ and, moreover, all rules from R are applicable to x as well as no rule from Q is applicable to x.

A random-context grammar $G_{RC} = (G_s, P', \Longrightarrow_{G_{RC}})$ of type X is called a *grammar with permitting contexts of type* X if for all rules (p, R, Q) in P' we have $Q = \emptyset$, i.e., we only check for the applicability of the rules in R.

A random-context grammar $G_{RC} = (G_s, P', \Longrightarrow_{G_{RC}})$ of type X is called a *grammar with forbidden contexts of type* X if for all rules (p, R, Q) in P' we have $R = \emptyset$, i.e., we only check for the non-applicability of the rules in Q. We write $X\text{-}fC_1$ if for every $p \in P$ there is only one rule of the form $(p, \emptyset, Q)$ in P'.

$L(G_{RC}) = \left\{v \in O_T \mid w \Longrightarrow^*_{G_{RC}} v\right\}$ is the language generated by G_{RC}. The families of languages generated by random context grammars, grammars with permitting contexts, and grammars with forbidden contexts of type X are denoted by $\mathcal{L}(X\text{-}RC)$, $\mathcal{L}(X\text{-}pC)$, and $\mathcal{L}(X\text{-}fC)$ or $\mathcal{L}(X\text{-}fC_1)$, respectively.

3.4 Grammars with Priority Relations on the Rules

A *grammar with a priority relation on the rules* G_{Pri} *of type* X is a construct $(G_s, \prec, \Longrightarrow_{G_{Pri}})$ where

– $G_s = (O, O_T, w, P, \Longrightarrow_G)$ is a grammar of type X;
– $\prec$ is a priority relation on the rules in P;
– $\Longrightarrow_{G_{Pri}}$ is the derivation relation assigned to G_{Pri} such that for any $x, y \in O$, $x \Longrightarrow_{G_{Pri}} y$ if and only if for some rule $q \in P$ $x \Longrightarrow_q y$ and, moreover, no rule p from P with $q \prec p$ is applicable to x.

$L(G_{Pri}) = \left\{v \in O_T \mid w \Longrightarrow^*_{G_{Pri}} v\right\}$ is the language generated by G_{Pri}. The family of languages generated by grammars with priority relations on the rules of type X is denoted by $\mathcal{L}(X\text{-}Pri)$.

3.5 Ordered Grammars

An *ordered grammar* G_O *of type* X is a grammar $(G_s, \prec, \Longrightarrow_{G_O})$ with the priority relation $\prec$ on the rules which is a partial order, i.e., $\prec$ fulfills the condition that for any $p, q, r \in P$, $p \prec q$ and $q \prec r$ implies $p \prec q$.

The family of languages generated by ordered grammars of type X is denoted by $\mathcal{L}(X\text{-}O)$.

3.6 Grammars with Activation and Blocking of Rules

We now recall the definition of sequential grammars with activation and blocking of rules in a similar way as introduced in [10, 3, 2].

A *grammar with activation and blocking of rules* (an *AB-grammar*) of type X is a construct

$$G_{AB} = (G_s, L, f_L, A, B, L_0, \Longrightarrow_{G_{AB}})$$

where $G_s = (O, O_T, w, P, \Longrightarrow_G)$ is a grammar of type X, L is a finite set of labels with each label having assigned one rule from P by the function f_L, A, B are finite subsets of $L \times L \times \mathbb{N}$, and L_0 is a finite set of tuples of the form $\left(q, Q, \bar{Q}\right)$, $q \in L$, with the elements of $Q, \bar{Q}$ being of the form (l, t), where $l \in L$ and $t \in \mathbb{N}$, $t > 1$.

A derivation in G_{AB} starts with one element $\left(q, Q, \bar{Q}\right)$ from L_0 which means that the rule labeled by q has to be applied to the initial object w in the first step and for the following derivation steps the conditions given by Q as activations of rules and $\bar{Q}$ as blockings of rules have to be taken into account in addition to the activations and blockings coming along with the application of the rule labeled by q. The role of L_0 is to get a derivation started by activating some rule for the first step(s) although no rule has been applied so far, but probably also providing additional activations and blockings for further derivation steps.

A configuration of G_{AB} in general can be described by the object derived so far and the activations Q and blockings $\bar{Q}$ for the next steps. In that sense, the starting tuple $\left(q, Q, \bar{Q}\right)$ can be interpreted as $\left(\{(q,1)\} \cup Q, \bar{Q}\right)$, and we may also simply write $\left(Q', \bar{Q}\right)$ with $Q' = \{(q,1)\} \cup Q$. We mostly will assume Q and $\bar{Q}$ to be non-conflicting, i.e., $Q \cap \bar{Q} = \emptyset$; otherwise, we interpret $\left(Q', \bar{Q}\right)$ as $\left(Q' \setminus \bar{Q}, \bar{Q}\right)$.

Given a configuration $\left(u, Q, \bar{Q}\right)$, in one step we can derive $\left(v, R, \bar{R}\right)$ – we also write $\left(u, Q, \bar{Q}\right) \Longrightarrow_{G_{AB}} \left(v, R, \bar{R}\right)$ – if and only if

- $u \Longrightarrow_G v$ using the rule r such that $(q,1) \in Q$ and $(q,r) \in f_L$, i.e., we apply the rule labeled by q activated for this next derivation step to u; the new sets of activations and blockings are defined by

$$\begin{aligned} \bar{R} &= \left\{(x,i) \mid (x,i+1) \in \bar{Q},\ i > 0\right\} \cup \left\{(x,i) \mid (q,x,i) \in B\right\}, \\ R &= \left(\{(x,i) \mid (x,i+1) \in Q,\ i > 0\} \cup \{(x,i) \mid (q,x,i) \in A\}\right) \\ &\quad \setminus \left\{(x,i) \mid (x,i) \in \bar{R}\right\} \end{aligned}$$

  (observe that R and $\bar{R}$ are made non-conflicting by eliminating rule labels which are activated and blocked at the same time);
  or

– no rule r is activated to be applied in the next derivation step; in this case we take $v = u$ and continue with $\left(v, R, \bar{R}\right)$ constructed as before provided R is not empty, i.e., there are rules activated in some further derivation steps; otherwise the derivation stops with yielding object u.

The language generated by G_{AB} is defined by

$$L(G_{AB}) = \left\{ v \in O_T \mid \left(w, Q, \bar{Q}\right) \Longrightarrow^*_{G_{AB}} \left(v, R, \bar{R}\right) \text{ for some } \left(Q, \bar{Q}\right) \in L_0 \right\}.$$

The family of languages generated by AB-grammars of type X is denoted by $\mathcal{L}(X\text{-}AB)$. If the set B of blocking relations is empty, then the grammar is said to be a *grammar with activation of rules* (an *A-grammar* for short) of type X; the corresponding family of languages is denoted by $\mathcal{L}(X\text{-}A)$.

4 General Results

We now recall the main results and proofs already established in [12] as well as recently exhibited in [10] and [3] for the control mechanisms defined in the preceding section.

Theorem 1. *For any arbitrary type X,*

$$\mathcal{L}(X\text{-}MAT_{ac}) \subseteq \mathcal{L}\left(X\text{-}GC_{ac}^{allfinal}\right) \subseteq \mathcal{L}(X\text{-}GC_{ac}) \text{ and}$$
$$\mathcal{L}(X\text{-}MAT) \subseteq \mathcal{L}\left(X\text{-}GC^{allfinal}\right) \subseteq \mathcal{L}(X\text{-}GC).$$

Proof. Let $G_M = (G_s, M, F, \Longrightarrow_{G_M})$ be a matrix grammar where

– $G_s = (O, O_T, w, P, \Longrightarrow_{G_s})$ is a grammar of type X and
– $M = \{(p_{i,1}, \ldots, p_{i,n_i}) \mid 1 \leq i \leq n\}$ with $p_{i,j} \in P$, $1 \leq j \leq n_i$, $1 \leq i \leq n$.

Then we construct the graph-controlled grammar $G_{GC} = (G_s, g, H_i, H_f, \Longrightarrow_{GC})$ with $g = (H, E, K)$, $H = \{(i,j) \mid 1 \leq j \leq n_i, 1 \leq i \leq n\}$, $K((i,j)) = \{p_{i,j}\}$, $1 \leq j \leq n_i$, $1 \leq i \leq n$,

$$\begin{aligned} E = &\{((i,j), Y, (i,j+1)) \mid 1 \leq j < n_i, 1 \leq i \leq n\} \\ &\cup \{((i,j), N, (i,j+1)) \mid 1 \leq j < n_i, 1 \leq i \leq n, p_{i,j} \in F\} \\ &\cup \{((i,n_i), Y, (j,1)) \mid 1 \leq j \leq n, 1 \leq i \leq n\} \\ &\cup \{((i,n_i), N, (j,1)) \mid 1 \leq j \leq n, 1 \leq i \leq n, p_{i,j} \in F\} \end{aligned}$$

and $H_i = \{(i,1) \mid 1 \leq i \leq n\}$. As we have assumed that the sequential application of the rules of the chosen matrix may stop at any moment, we have to take $H_f = H$. By this construction it is guaranteed that G_{GC} simulates a derivation in G_M correctly by choosing a matrix to be simulated in a non-deterministic way and then applying the rules from this matrix in the desired sequence; the application of a rule $p_{i,j}$ may be skipped if and only if $p_{i,j} \in F$. G_{GC} is without applicability checking if and only if G_M is without applicability checking, which observation completes the proof. □

By definition, we have:

Lemma 1. $\mathcal{L}(X\text{-}O) \subseteq \mathcal{L}(X\text{-}Pri)$.

The following theorem shows that forbidden contexts with only one set of forbidden rules for each rule can simulate any priority relation on the rules:

Theorem 2. *For any arbitrary type X, $\mathcal{L}(X\text{-}Pri) \subseteq \mathcal{L}(X\text{-}fC_1)$.*

Proof. Let $G_s = (O, O_T, w, P, \Longrightarrow_G)$ be a grammar of type X. Consider the grammar with a priority relation on the rules $G_{Pri} = (G_s, \prec, \Longrightarrow_{G_{Pri}})$ of type X and the corresponding grammar with forbidden contexts $G_{fC_1} = \left(G_s, P_{fC_1}, \Longrightarrow_{G_{fC_1}}\right)$ of type X where

$$P_{fC_1} = \{(p, \emptyset, Q(p)) \mid p \in P\} \quad \text{with} \quad Q(p) = \{q \mid q \in P, p \prec q\}.$$

As a rule $p \in P$ can be applied in G_{fC_1} if and only if no rule from $Q(p)$ is applicable which is the same condition as for the applicability of p in G_{Pri}, we infer $L(G_{fC_1}) = L(G_{Pri})$. □

Yet also the reverse inclusion holds, even for partial order relations, provided the type X allows for trap rules:

Theorem 3. *For any type X with trap rules, $\mathcal{L}(X\text{-}fC_1) \subseteq \mathcal{L}(X\text{-}O)$.*

Proof. Let $G_s = (O, O_T, w, P, \Longrightarrow_G)$ be a grammar of type X and consider the grammar with forbidden contexts $G_{fC_1} = \left(G_s, P_{fC_1}, \Longrightarrow_{G_{fC_1}}\right)$ of type X with $P_{fC_1} = \{(p, \emptyset, Q(p)) \mid p \in P\}$. We now extend the underlying grammar G_s by the trap rules p^- for all rules p in P, thus obtaining the grammar $G'_s = \left(O, O_T, w, P \cup P^{(-)}, \Longrightarrow_{G'_s}\right)$ where, according to the definition of grammars with trap rules,

- $P^{(-)} = \left\{p^{(-)} \mid p \in P\right\}$, $P^{(-)} \cap P = \emptyset$,
- for all $x \in O$, $p^{(-)}$ is applicable to x if and only if p is applicable to x, and
- for all $x \in O$, if $p^{(-)}$ is applicable to x, the application of $p^{(-)}$ to x yields an object y from which no terminal object can be derived anymore.

As X is a type with trap rules, G'_s again is of type X. We now define the ordered grammar $G_O = (G'_s, \prec, \Longrightarrow_{G_O})$ which by definition again is of type X, with the partial order $\prec$ on the rules in $P \cup P^{(-)}$ as follows:

$$\text{for any } p \in P,\ p \prec q^- \text{ for all } q \in Q(p).$$

This guarantees that $L(G_{fC_1}) = L(G_O)$, as a rule $p \in P$ can be applied in G_O if and only if no rule from $Q(p)$ is applicable which is the same condition as for the applicability of p in G_{fC_1}. On the other hand, the application of a rule in $P^{(-)}$ can never lead to a terminal result. Moreover, it is obvious to see that $\prec$ is a partial order, because $\prec \subseteq P \times P^{(-)}$ and, by definition, $P^{(-)} \cap P = \emptyset$. □

As an immediate consequence of Lemma 1 and Theorems 2 and 3 we infer:

Corollary 1. *For any type X with trap rules,*

$$\mathcal{L}(X\text{-}O) = \mathcal{L}(X\text{-}Pri) = \mathcal{L}(X\text{-}fC_1) \subseteq \mathcal{L}(X\text{-}fC).$$

Matrix grammars (with applicability checking) can simulate random context grammars for any arbitrary type X with unit rules and trap rules:

Theorem 4. *For any arbitrary type X with unit rules and trap rules,*

$$\mathcal{L}(X\text{-}RC) \subseteq \mathcal{L}(X\text{-}MAT_{ac}).$$

Proof. Consider a *random-context grammar* $G_{RC} = (G_s, P_{RC}, \Longrightarrow_{G_{RC}})$ where $G_s = (O, O_T, w, P, \Longrightarrow_G)$ is a grammar of a type X with unit rules and trap rules; then we define the *matrix grammar* with appearance checking $G_M = (G'_s, M, F, \Longrightarrow_M)$ of type X as follows: for each rule $(p, R, Q) \in P_{RC}$, $R = \{r_i \mid 1 \leq i \leq m\}$, $Q = \{q_j \mid 1 \leq j \leq n\}$, $m, n \geq 0$, we take the matrix $\left(r_1^{(+)}, \ldots, r_m^{(+)}, q_1^{(-)}, \ldots, q_n^{(-)}, p\right)$ into M.

In that way we obtain $G'_s = \left(O, O_T, w, P', \Longrightarrow_{G'_s}\right)$ where

$$P' = P \cup \left\{r^{(+)}, q^{(-)} \mid r \in R, q \in Q \text{ for some } (p, R, Q) \in P_{RC}\right\}$$

and $F = \left\{q^{(-)} \mid q \in Q \text{ for some } (p, R, Q) \in P_{RC}\right\}$. As X is a type with unit rules and trap rules, all the elements of G_M are well defined. Obviously, for all $x, y \in O$ we have $x \Longrightarrow_{(p,R,Q)} y$ if and only if $x \Longrightarrow_{\left(r_i^{(+)}, \ldots, r_m^{(+)}, q_1^{(-)}, \ldots, q_n^{(-)}, p\right)} y$ without trapping y, which implies $L(G_M) = L(G_{RC})$.

As a technical detail we mention that when the application of rules in the sequence of the matrix $\left(r_i^{(+)}, \ldots, r_m^{(+)}, q_1^{(-)}, \ldots, q_n^{(-)}, p\right)$ stops before having reached the end with applying p, either the underlying object has not yet changed as long as only the unit rules have been applied or else has already been trapped by the application of one of the trap rules, hence, no additional terminal results can arise from such situations. □

Omitting the forbidden rules and applicability checking, respectively, from the (proof of the) preceding theorem we immediately obtain the following result:

Corollary 2. *For any arbitrary type X with unit rules,*

$$\mathcal{L}(X\text{-}pC) \subseteq \mathcal{L}(X\text{-}MAT).$$

Already in [12] graph-controlled grammars have been shown to be the most powerful control mechanism, and they can also simulate AB-grammars with the underlying grammar being of any arbitrary type X, see [3].

Theorem 5. *For any type X, $\mathcal{L}(X\text{-}AB) \subseteq \mathcal{L}(X\text{-}GC_{ac})$.*

Proof. Let $G_{AB} = (G, L, f_L, A, B, L_0, \Longrightarrow_{G_A})$ be an AB-grammar with the underlying grammar $G = (O, O_T, w, P, \Longrightarrow_G)$ being of any type X. Then we construct a graph-controlled grammar $G_{GC} = (G, g, H_i, H_f, \Longrightarrow_{GC})$ with the same underlying grammar G. The simulation power is captured by the structure of the control graph $g = (H, E, K)$. The node labels in H, identifying the nodes of the graph in a one-to-one manner, are obtained from G_{AB} as all possible triples of the forms $\left(q, Q, \bar{Q}\right)$ or $\left(\bar{q}, Q, \bar{Q}\right)$ with $q \in L$ and the elements of $Q, \bar{Q}$ being of the form (r, t), $r \in L$ and $t \in \mathbb{N}$ such that t does not exceed the maximum time occurring in the relations in A and B, hence, this in total is a bounded number. We also need a special node labeled $\emptyset$, where a computation in G_{GC} ends in any case when this node is reached. All nodes can be chosen to be final, i.e., $H_f = H$. $H_i = L_0$ is the set of initial labels, i.e., we start with one of the initial conditions as in the AB-grammar.

The idea behind the node $\left(q, Q, \bar{Q}\right)$ is to describe the situation of a configuration derived in the AB-grammar where q is the label of the rule to be applied and $Q, \bar{Q}$ describe the activated and blocked rules for the further derivation steps in the AB-grammar. Hence, as already in the definition of an AB-grammar, we therefore assume $Q \cap \bar{Q} = \emptyset$.

Now let $g(l)$ denote the rule r assigned to label l, i.e., $(l, r) \in f_L$. Then, the set of rules assigned to $\left(q, Q, \bar{Q}\right)$ is taken to be $\{g(q)\}$. The set of rules assigned to $\emptyset$ is taken to be $\emptyset$.

As it will become clear later in the proof why, the nodes $\left(\bar{q}, Q, \bar{Q}\right)$ are assigned the set of rules $\{g(l) \mid (l, 1) \in Q,\ l \neq q\}$; we only take those nodes where this set is not empty.

When being in node $\left(q, Q, \bar{Q}\right)$, we have to distinguish between two possibilities:

- If $g(q)$ is applicable to the object derived so far, a Y-edge has to go to every node which describes a situation corresponding to what would have been the next configuration in the AB-grammar. We then compute

$$\begin{aligned} \bar{R} &= \left\{(x, i) \mid (x, i+1) \in \bar{Q},\ i > 0\right\} \cup \left\{(x, i) \mid (q, x, i) \in B\right\}, \\ R &= \left(\left\{(x, i) \mid (x, i+1) \in Q,\ i > 0\right\} \cup \left\{(x, i) \mid (q, x, i) \in A\right\}\right) \\ &\quad \setminus \left\{(x, i) \mid (x, i) \in \bar{R}\right\} \end{aligned}$$

  (observe that R and $\bar{R}$ are made non-conflicting) as well as – if it exists – $t_0 := min\{t \mid (x, t) \in R\}$, i.e., the next time step when the derivation in the AB-grammar could continue. Hence, we take a Y-edge to every node $\left(p, P, \bar{P}\right)$ where $p \in \{x \mid (x, t_0) \in R\}$ and

$$\begin{aligned} \bar{P} &= \left\{(x, i) \mid (x, i + t_0 - 1) \in \bar{R},\ i > 0\right\}, \\ P &= \left\{(x, i) \mid (x, i + t_0 - 1) \in R\right\}. \end{aligned}$$

  If $t_0 := min\{t \mid (x, t) \in R\}$ does not exist, this means that R is empty and we have to make a Y-edge to the node $\emptyset$.
- If $g(q)$ is not applicable to the object derived so far, we first have to check that none of the other rules activated at this step could have been applied,

i.e., we check for the applicability of the rules in the set of rules

$$\bar{U} := \{g(l) \mid (l,1) \in Q,\ l \neq q\}$$

by going to the node $\left(\bar{q}, Q, \bar{Q}\right)$ with a N-edge; from there no Y-edge leaves, as this would indicate the unwanted case of the applicability of one of the rules in $\bar{U}$, but with a N-edge we continue the computation in any node $\left(p, P, \bar{P}\right)$ with p, P, $\bar{P}$ computed as above in the first case. We observe that in case $\bar{R}$ is empty, we can omit the path through the node $\left(\bar{q}, Q, \bar{Q}\right)$ and directly go to the nodes $\left(p, P, \bar{P}\right)$ which are obtained as follows: we first check whether $t_0 := min\{t \mid (x,t) \in Q,\ t > 1\}$ exists or not; if not, then the computation has to end with a N-edge to node $\emptyset$. Otherwise, a N-edge goes to every node $\left(p, P, \bar{P}\right)$ with $p \in \{x \mid (x,t_0) \in Q\}$ and

$$\begin{aligned} \bar{P} &= \left\{(x,i) \mid (x, i+t_0-1) \in \bar{Q},\ i > 0\right\}, \\ P &= \left\{(x,i) \mid (x, i+t_0-1) \in Q\right\}. \end{aligned}$$

where the simulation may continue.

In this way, every computation in the AB-grammar can be simulated by the graph-controlled grammar with taking a correct path through the control graph and finally ending in node $\emptyset$; due to this fact, we could also choose the node $\emptyset$ to be the only final node, i.e., $H_f = \{\emptyset\}$. On the other hand, if we have made a wrong choice and wanted to apply a rule which is not applicable, although another rule activated at the same moment would have been applicable, we get stuck, but the derivation simulated in this way still is a valid one in the AB-grammar, although in most standard types X, which usually are strictly extended ones, such a derivation does not yield a terminal object. Having taken $H_f = \{\emptyset\}$, such paths would not even lead to successful computations in G_{GC}.

In any case, we conclude that the graph-controlled grammar G_{GC} generates the same language as the AB-grammar G_{AB}, which observation concludes the proof. □

We remark that in the construction of the graph-controlled grammar given in the preceding proof, all labels could be chosen to be final.

In the case of graph-controlled grammars with all labels being final, for any strictly extended type X with trap rules we can show that the power of rule activation is already sufficient and that the additional power of blocking is not needed.

Theorem 6. *For any strictly extended type X with trap rules,*

$$\mathcal{L}\left(X\text{-}GC_{ac}^{allfinal}\right) \subseteq \mathcal{L}\left(X\text{-}A\right).$$

Proof. Let $G_{GC} = (G_s, g, H_i, H_f, \Longrightarrow_{GC})$ be a graph-controlled grammar where $G_s = (O, O_T, w, P, \Longrightarrow_G)$ is a strictly extended grammar of type X with trap rules; $g = (H, E, K)$, $E \subseteq H \times \{Y, N\} \times H$ is the set of edges labeled by Y or

N, $K : H \to 2^P$ is a function assigning a subset of P to each node of g; $H_i \subseteq H$ is the set of initial labels, and H_f is the set of final labels coinciding with the whole set H, i.e., $H_f = H$.

Then we construct an equivalent A-grammar $G_A = (G'_s, L, f_L, A, L_0, \Longrightarrow_{G_A})$ as follows: the underlying grammar G'_s is obtained from G_s by adding all trap rules, i.e., $G'_s = \big(O, O_T, w, P', \Longrightarrow_{G'_s}\big)$ with $P' = P \cup P^{(-)}$, $P^{(-)} = \{p^- \mid p \in P\}$, $P^{(-)} \cap P = \emptyset$. G'_s again is strictly extended and $w \notin O_T$, hence, also in G_A rules have to be applied before terminal objects are obtained. For any node in g labeled by l with the assigned set of rules P_l we assume it to be described by $P_l = \{p_{l,i} \mid 1 \leq i \leq n_l\}$. For all $q \in P$ we take the labels l_{q^-} into L as well as (l_{q^-}, q^-) into f_L.

We now sketch how the transitions from a node in g labeled by l with the assigned set of rules P_l can be simulated. The assumption that all nodes are final is crucial for this construction. Arriving in some node, one of the following situations is given:

1. the underlying object is terminal and therefore no rule from P is applicable any more, as X is a strictly extended type; hence, we may stop in this node and extract the underlying object as a terminal result of the derivation, as all nodes are final;
2. the underlying object is not terminal, but no rule from $\bigcup_{i \in H} P_i$ is applicable any more; hence, even when continuing the derivation following a path through the control graph only using N-edges, the derivation cannot yield a terminal object any more; therefore, in such a case, we need not continue the derivation;
3. the underlying object is not terminal, no rule $p_{l,i}$ in P_l, $1 \leq i \leq n_l$, is applicable, but there is still some node k reachable from node l following a path through the control graph only using N-edges that contains an applicable rule;
4. the underlying object is not terminal, but there is some rule $p_{l,i}$ in P_l, $1 \leq i \leq n_l$, which is applicable.

For the simulation of these situations by the A-grammar, we therefore can restrict ourselves to the cases where when applying a rule we follow a path starting with a Y-edge and continuing with only N-edges until we reach a node containing a probably applicable rule; observe that such a path can only consist of the Y-edge, too.

In order to simulate a rule $p_{l,i}$ in P_l, $1 \leq i \leq n_l$, we take all activations into A which allow us to simulate the application of $p_{l,i}$ and to guess with which $p_{k,j}$ probably to continue afterwards. Hence, we consider all paths without loops $h_0 = l - h_1 - \cdots - h_n = k$ in the control graph g which start with a Y-edge and continue with only N-edges. For any such path we introduce labels $((l,i), h_1, \ldots, (k,j))$ in L and $((l,i), h_1, \ldots, (k,j)) : p_{l,i}$ in f_L; the set of all labels describing such paths from node l to any node k is denoted by $L_{l,i}$. Moreover, we use the following activations in A:

– $((l,i),h_1,\ldots,(k,j)),\{l_{q^-} \mid q \in \bigcup_{1\leq i\leq n-1} P_{h_i}\},1)$ is used to check in the next step that no rule along the path from node l to node k is applicable; observe that for $n=1$ the set $\bigcup_{1\leq i\leq n-1} P_{h_i}$ is empty and the whole activation can be omitted;
– in the second next step only the designated rule $p_{k,j}$ can be applied, i.e., we take $((l,i),h_1,\ldots,(k,j)),L_{k,j},2)$ into A; as with every label in $L_{k,j}$ the rule $p_{k,j}$ is assigned, the intended continuation is prepared.

How can a derivation in the A-grammar be started? As $w \notin O_T$, at least one rule must be applied to obtain a terminal object; hence, we check all possibilities that a rule in an initial node in H_i or along a path in g following only N-edges from such an initial node can be applied (observe that there are only finitely many paths without loops of that kind through the control graph); for each such rule $p_{l,i}$ in node l we take all labels from $L_{l,i}$ into L_0. As by construction $p_{l,i}$ is applicable it is guaranteed that any continuation of the computation will follow a Y-edge in g and thus the simulation in G_A will follow the simulation of an applicable rule as described above.

In total, the construction given above guarantees that the simulation of a computation in G_{GC} by a computation in G_A starts correctly and continues until no rule can be applied any more. As we have assumed all nodes in g to be final and X to be a strictly extended type, i.e., no rules can be applied to a terminal object any more, the only condition to get a result is to obtain a terminal object at the end of a computation. This observation completes our proof. □

As programmed grammars are just a special case of graph-controlled grammars with all labels being final, we immediately infer the following result:

Corollary 3. *For any strictly extended type X with trap rules,*

$$\mathcal{L}\,(X\text{-}P_{ac}) \subseteq \mathcal{L}\,(X\text{-}A)\,.$$

Combining (the proofs of) Theorems 5 and 6, we infer the following equality:

Corollary 4. *For any strictly extended type X with trap rules,*

$$\mathcal{L}\left(X\text{-}GC_{ac}^{allfinal}\right) = \mathcal{L}\,(X\text{-}A)\,.$$

5 Summary of General Results

The main results elaborated for the relations between the specific regulating mechanisms in [12] and in [3] are depicted in the following diagram.

Theorem 7. *The inclusions indicated by vectors as depicted in Figure 1 hold. Most of the relations indicated by vectors even hold for arbitrary types X; additionally needed features of being a strictly extended type or being a type with unit and/or trap rules are indicated by se, u, and t, respectively, aside the vector:*

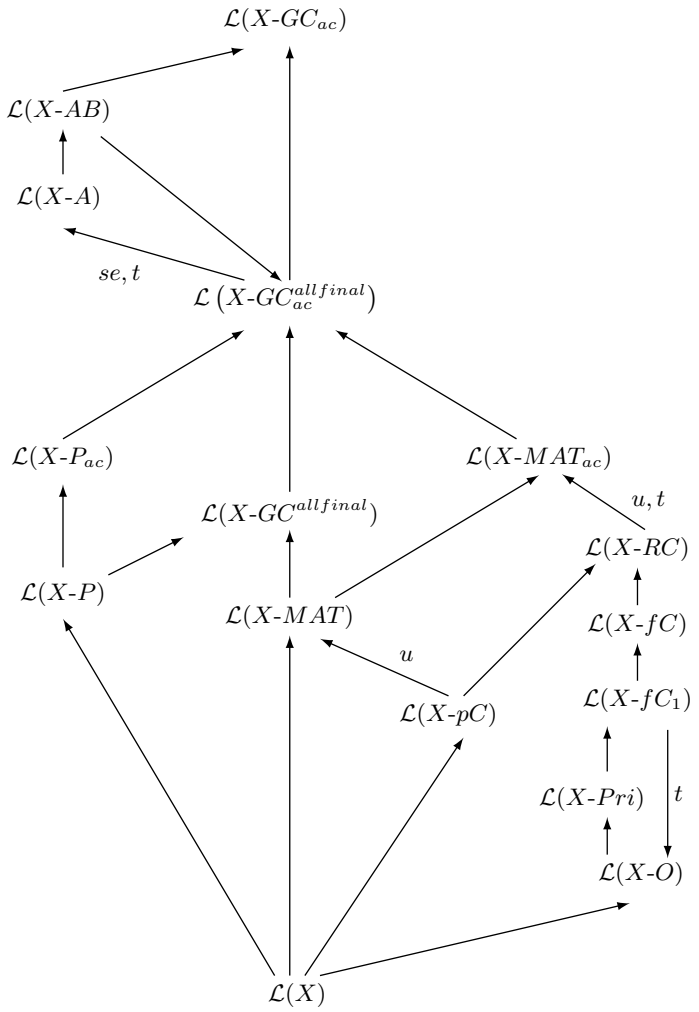


Fig. 1. Hierarchy of control mechanisms for grammars of type X.

6 Results for Strings and Multisets

As specific types of objects for the general model of a sequential grammar as introduced in Section 3 we now consider strings and multisets. We refer to [12] where some examples for string and multiset grammars of specific types illustrating the expressive power of this general framework are given.

6.1 String Grammars

In the general model, $G_S = \left((N \cup T)^*, T^*, w, P, \Longrightarrow_P\right)$ is called a *string grammar*; N is the alphabet of *non-terminal symbols*, T is the alphabet of *terminal*

symbols, $N \cap T = \emptyset$, $w \in (N \cup T)^+$, P is a finite set of *rules* of the form $u \to v$ with $u \in V^+$ and $v \in V^*$, where $V := N \cup T$; the derivation relation for $u \to v \in P$ is defined by $xuy \Longrightarrow_{u \to v} xvy$ for all $x, y \in V^*$, thus yielding the well-known derivation relation $\Longrightarrow_{G_S}$ for the string grammar G_S. We mention that the common notation for a string grammar is $G_S = (N, T, w, P)$, and usually the axiom w is supposed to be a non-terminal symbol, i.e., $w \in V \setminus T$, which then is called the *start symbol*.

As special types of string grammars we consider string grammars with arbitrary rules and context-free rules of the form $A \to v$ with $A \in N$ and $v \in V^*$. The corresponding types of grammars are denoted by ARB and CF, thus yielding the families of languages $\mathcal{L}(ARB)$, i.e., the family of recursively enumerable languages (also denoted by RE), as well as $\mathcal{L}(CF)$, i.e., the family of context-free languages, respectively. Observe that the types ARB and CF are types with unit rules and trap rules (for $p = w \to v \in P$, we can take $p^{(+)} = w \to w$ and $p^{(-)} = w \to F$ where $F \notin T$ is a new symbol – the trap symbol).

6.2 Multiset Grammars

$G_m = \left((N \cup T)^\circ, T^\circ, w, P, \Longrightarrow_{G_m}\right)$ is called a *multiset grammar*; N is the alphabet of *non-terminal symbols*, T is the alphabet of *terminal symbols*, $N \cap T = \emptyset$, w is a non-empty multiset over V, $V := N \cup T$, and P is a finite set of multiset rules yielding a derivation relation $\Longrightarrow_{G_m}$ on the multisets over V; the application of the rule $u \to v$ to a multiset x has the effect of replacing the multiset u contained in x by the multiset v. For the multiset grammar G_m, the common notation is $(N, T, w, P, \Longrightarrow_{G_m})$.

As special types of multiset grammars we consider multiset grammars with *arbitrary* rules as well as *context-free* (*non-cooperative*) rules of the form $A \to v$ with $A \in N$ and $v \in V^\circ$; the corresponding types X of multiset grammars are denoted by $mARB$ and mCF, thus yielding the families of multiset languages $\mathcal{L}(X)$. Observe that $mARB$ and mCF are types with unit rules and trap rules (for $p = w \to v \in P$, we can take $p^{(+)} = w \to w$ and $p^{(-)} = w \to F$ where F is a new symbol – the trap symbol). Even with arbitrary multiset rules, it is not possible to get $Ps(\mathcal{L}(ARB))$ [16]:

$$\mathcal{L}(mCF) = Ps(\mathcal{L}(CF)) \subsetneqq \mathcal{L}(mARB) \subsetneqq Ps(\mathcal{L}(ARB)).$$

6.3 Results for String and Multiset Grammars

It is well-known, for example see [7], that

$$\mathcal{L}(CF\text{-}RC) = \mathcal{L}(CF\text{-}P_{ac}) = \mathcal{L}(ARB) = RE.$$

Based on Theorem 7, we immediately infer the following results:

Theorem 8. *For any* $Y \in \left\{RC, MAT_{ac}, GC_{ac}^{allfinal}, GC_{ac}, P_{ac}, A, AB\right\}$,

$$\mathcal{L}(CF\text{-}Y) = \mathcal{L}(ARB) = RE.$$

As in the case of multisets the structural information contained in the sequence of symbols cannot be used, arbitrary multiset rules are not sufficient for obtaining all sets in $Ps(\mathcal{L}(ARB))$. Yet we can easily show the following:

Theorem 9.
For any $Y \in \{O, Pri, fC_1, fC, RC, MAT_{ac}, GC_{ac}^{allfinal}, GC_{ac}, A, AB\}$,

$$PsRE = Ps(\mathcal{L}(ARB)) = \mathcal{L}(mARB\text{-}Y).$$

Proof. $PsRE = Ps(\mathcal{L}(ARB)) = \mathcal{L}(mARB\text{-}O)$ was shown in [12], hence, the statement immediately follows from Theorem 7. □

But also non-cooperative multiset rules are sufficient with many control mechanisms:

Theorem 10. *For any* $Y \in \{MAT_{ac}, GC_{ac}^{allfinal}, GC_{ac}, A, AB\}$,

$$PsRE = Ps(\mathcal{L}(ARB)) = \mathcal{L}(mCF\text{-}Y).$$

Proof. $PsRE = Ps(\mathcal{L}(ARB)) = \mathcal{L}(mCF\text{-}MAT_{ac})$ was shown in [16], hence, the statement immediately follows from Theorem 7. □

6.4 Computational Completeness for Context-Free AB-Grammars with Two Non-Terminal Symbols

In this subsection, we recall complexity results for context-free string and multiset grammars as shown in [3], showing that computational completeness can already be obtained with two non-terminal symbols, which result is optimal with respect to the number of non-terminal symbols.

Theorem 11. *Any recursively enumerable set of strings can be generated by a context-free AB-grammar using only two non-terminal symbols.*

Proof. (Sketch) The main technical details of how to use only two non-terminal symbols A and B for generating a given recursively enumerable language follow the construction given in [12] for graph-controlled grammars. The most important to be shown here is how to simulate the ADD- and SUB-instructions of a deterministic register machine with the contents of the two working registers being given by the number of symbols A and B; only at the end, both numbers are zero, whereas in between, during the whole computation, at least one symbol A or B is present. The initial string is A, and one A is also the last symbol to be erased at the end in order to obtain a terminal string.

In the following, we use X to specify one of the two non-terminal symbols A and B, and Y then stands for the other one. For any label p of the register machine we use two labels p and p'. To simplify notations, we write $(p, q, t)_U$ instead of $(p, q, t) \in U$ for $U \in \{A, B\}$.

The simulations in the AB-grammar then work as follows:

– $p : (ADD(X), q)$ is simulated by $p : X \to XX$ and $p' : Y \to YX$ with $(p, p', 1)_B$ as well as $(p, q, 2)_A$, $(p, q', 3)_A$, and $(p', q, 1)_A$, $(p', q', 2)_A$;
– $p : (SUB(X), q, s)$ is simulated by $p : X \to \lambda$ and $p' : Y \to Y$ with $(p, p', 1)_B$ as well as $(p, q, 2)_A$, $(p, q', 3)_A$, and $(p', s, 1)_A$, $(p', s', 2)_A$;

in both cases, the application of the rule labeled by p blocks the rule labeled by p'; in any case, for the next rule labeled r to be simulated, both r and r' are activated, again r' following r one step later.

For the halting label h, only the labeled rule $h : A \to \lambda$ is to be activated. □

This result is optimal with respect to the number of non-terminal symbols: as it has been shown in [8], even for graph-controlled context-free grammars one non-terminal symbol is not enough, hence, the statement immediately follows from Theorem 5. A similar optimal result holds for multiset grammars.

Theorem 12. *Any recursively enumerable set of multisets can be generated by an AB-grammar using context-free multiset rules and only two non-terminal symbols.*

Proof. Given a recursively enumerable set of multisets L over the terminal alphabet $T = \{a_1, \ldots, a_k\}$, we can construct a register machine M_L generating L in the following way: instead of speaking of a number n in register r we use the notation ${a_r}^n$, i.e., a configuration of M_L is represented as a string over the alphabet $V = T \cup \{a_{k+1}, a_{k+2}\}$ with the two non-terminal symbols a_{k+1}, a_{k+2}.

We start with one a_{k+1} and first generate an arbitrary multiset over T step by step adding one element a_m from T and at the same time multiply the number of symbols a_{k+1} by p_m, where p_m is the m-th prime number. At the end of this procedure, for the multiset ${a_1}^{n_1} \ldots {a_k}^{n_k}$ we have obtained ${a_m}^{n_m}$ in each register m, $1 \leq m \leq k$, and ${a_{k+1}}^{{p_1}^{n_1} \cdots {p_k}^{n_k}}$ in register $k+1$. As for example, already shown in [17], only using registers $k+1$ and $k+2$, a deterministic register machine M'_L simulating any number of registers by this prime number encoding can compute starting with ${a_{k+1}}^{{p_1}^{n_1} \cdots {p_k}^{n_k}}$ and halt if and only if ${a_1}^{n_1} \ldots {a_k}^{n_k} \in L$. Only with halting, all registers of M'_L are cleared to zero, i.e., we end up with only one a_{k+1} in M_L when this deterministic register machine M'_L has reached its halting label h. So the last step of M_L before halting is just to eliminate this last a_{k+1}. During the whole computation of M_L, the sum of symbols a_{k+1} and a_{k+2} is greater than zero. Hence, it only remains to show how to simulate the instructions of a register machine, which is done in a similar way as in the preceding proof; we use X to specify one of the two non-terminal symbols a_{k+1} and a_{k+2}, and Y then stands for the other one, i.e., $X, Y \in \{a_{k+1}, a_{k+2}\}$. For any label p of the register machine we use two labels p and p'. The simulations in the AB-grammar work as follows:

– a non-deterministic ADD-instruction $p : (ADD(X), q, s)$ is simulated by branching into two deterministic ADD-instructions even twice:
$p : X \to X$ and $p' : Y \to Y$ with $(p, p', 1)_B$ as well as
$(p, (p, X, q), 2)_A$, $(p, (p, X, s), 2)_A$, and $(p', (p, Y, q), 1)_A$, $(p', (p, Y, s), 1)_A$;

in the third step of the simulation, we already know whether X is present or else we have to use Y; this now allows us to simulate the four deterministic ADD-instructions $(p, \alpha, \beta) : (ADD(X), \beta)$, $\alpha \in \{X, Y\}$, $\beta \in \{q, s\}$, in a simpler way by using the rules
$(p, \alpha, \beta) : \alpha \to \alpha X$
and the activations
$((p, \alpha, \beta), \beta, 1)_A$, $((p, \alpha, \beta), \beta', 2)_A$;

- $p : (ADD(X), q)$ is simulated by $p : X \to XX$ and $p' : Y \to YX$ with $(p, p', 1)_B$ as well as $(p, q, 2)_A$, $(p, q', 3)_A$, and $(p', q, 1)_A$, $(p', q', 2)_A$;

- $p : (SUB(X), q, s)$ is simulated by $p : X \to \lambda$ and $p' : Y \to Y$ with $(p, p', 1)_B$ as well as $(p, q, 2)_A$, $(p, q', 3)_A$, and $(p', s, 1)_A$, $(p', s', 2)_A$;

  in both cases, the application of the rule labeled by p blocks the rule labeled by p'; in any case, for the next rule labeled r to be simulated, both r and r' are activated, again r' following r one step later;

- for the halting label h, only the labeled rule $h : a_{r+1} \to \lambda$ is to be activated.

When the final rule $h : a_{r+1} \to \lambda$ is applied, no further rule is activated, thus the derivation ends yielding the multiset ${a_1}^{n_1} \dots {a_k}^{n_k} \in L$ as terminal result. □

7 Arrays and Array Grammars on Cayley Grids

As a natural extension of string languages (e.g., see [23, 24]), arrays on the d-dimensional grid $\mathbb{Z}^d$ have been introduced and investigated since more than four decades, for example, see [5]. Applications of array grammars and array automata especially can be found in the area of pattern and picture recognition, for instance, see [21, 22, 25].

Following some ideas of Erzsébet Csuhaj-Varjú and Victor Mitrana, the investigation of array grammars and array automata on Cayley grids of finitely presented groups was started in [13] and then continued in more detail in [14]. As a first example of arrays on a Cayley grid of a non-Abelian group we refer to [1], where arrays on the hexagonal grid were considered.

In this section, first the notions and definitions for arrays defined on Cayley grids of finitely presented groups as well as for array grammars generating sets of such arrays are recalled from [14]. Following the general results collected in Section 5, we immediately obtain many results for array grammars defined on Cayley grids of finitely presented groups equipped with these control mechanisms. When using #-context-free array productions in the underlying array grammars, together with most of these control mechanisms considered previously in this paper, the same computational power as with arbitrary array productions can be obtained, see [10].

7.1 Arrays on Cayley Grids

In this subsection we generalize the concept of d-dimensional arrays to arrays defined on Cayley grids. Let $G = \langle B \mid R \rangle$ be a finitely presented group with $B = \{e_1, \ldots, e_m\}$ and G' denoting the set of group elements; moreover, let $C(G)$ be the Cayley graph of G with respect to B. Throughout the rest of the paper we will assume that $B^{-1} \subseteq B$, i.e., B contains all inverses of its elements. For paths in the Cayley graph this means that for each path $v = w_1 \to ... \to w_n = w$ in $C(G)$ from v to w also its inverse $w = w_n \to ... \to w_1 = v$ is a path in $C(G)$.

A finite *array* $\mathcal{A}$ over an alphabet V on G' is a function $\mathcal{A} : G' \to V \cup \{\#\}$, where $shape(\mathcal{A}) = \{ v \in G' \mid \mathcal{A}(v) \neq \# \}$ is finite and $\# \notin V$ is called the *background* or *blank symbol*, i.e., the nodes of $C(G)$ get assigned elements of $V \cup \{\#\}$. We usually will write $\mathcal{A} = \{(v, \mathcal{A}(v)) \mid v \in shape(\mathcal{A})\}$.

By V^G we denote the set of arrays over V on G'; any subset of V^G is called an array language over V on G. With respect to the finite presentation of G by $C(G)$, instead of V^G we also write $V^{C(G)}$ to emphasize that.

The *empty array* in V^G has empty shape and is denoted by Λ_G. Ordering the generators in B in a specific way as $e_1 < \cdots < e_m$, for each array $\mathcal{A} = \{(v, \mathcal{A}(v)) \mid v \in shape(\mathcal{A})\}$ in $V^G \setminus \{\Lambda_G\}$ we get a canonical representation as a list $\langle (v_1, \mathcal{A}(v_1)), \ldots, (v_n, \mathcal{A}(v_n)) \rangle$ such that $\{v_i \mid 1 \leq i \leq n\} = shape(\mathcal{A})$ and $v_i < v_{i+1}$, $1 \leq i < n$, with respect to the length-plus-lexicographic ordering of strings with the elements of G written as sums of the elements in B (the length-plus-lexicographic ordering $\prec$ is a well-ordering, where for two strings u and v, $u \prec v$ if either $|u| < |v|$ or $|u| = |v|$, $u = xay$, $v = xby$, and $a < b$). In terms of $C(G)$ this means that the elements of the array are listed in the length-plus-lexicographic ordering of the paths in $C(G)$ seen from the origin (the identity).

Example 5. Consider the hexagonal grid from Example 4. Then the "position" abc can also be reached by taking the path cba from the "origin" (the identity e). Hence, with taking the ordering $a < b < c$, the canonical representation of the array $\mathcal{A} = \{(ab, X), (abc, Y) \mid v \in shape(\mathcal{A})\} \in \{X, Y\}^{C(\langle a,b,c \mid a^2, b^2, c^2, (abc)^2 \rangle)}$ is $\langle (ab, X), (abc, Y) \rangle$.

Example 6. A *d-dimensional array* is an array over the free group $\mathbb{Z}^d$. If we take the unit vectors $e_k = (0, ..., 1, ..., 0)$ and their inverses $(0, ..., -1, ..., 0)$, the resulting Cayley graph is the well-known d-dimensional grid.

For any $v \in G'$, the *translation* $\tau_v : G' \to G'$ is defined by $\tau_v(w) = w \circ v$ for all $w \in G'$, and for any array $\mathcal{A} \in V^{C(G)}$ we define $\tau_v(\mathcal{A})$, the corresponding array translated by v, by $(\tau_v(\mathcal{A}))(w) = \mathcal{A}(w \circ v^{-1})$ for all $w \in G'$.

An array $\mathcal{A} \in V^{C(G)}$ is called *k-connected* if for any two elements v and w in $shape(\mathcal{A})$ there is a path $v = w_1 \to \cdots \to w_n = w$ in $C(G)$ with $\{w_1, ..., w_n\} \subseteq shape(\mathcal{A})$ such that for the distance in $C(G)$ between w_i and w_{i-1}, $d(w_i, w_{i-1})$, we have $d(w_i, w_{i-1}) \leq k$ for all $1 < i \leq n$; the distance $d(x, y)$ between two nodes x and y in $C(G)$ is defined as the length of the shortest path between x and y in $C(G)$. The subset of k-connected arrays in $V^{C(G)}$ is denoted by $V^{C(G)_k}$.

Example 7. Consider the set of one-dimensional arrays over the alphabet $\{a\}$, i.e., $\{a\}^{C(\langle(1),(-1)\rangle)}$, which in a simpler way we will also write as $\{a\}^{\mathbb{Z}^1}$. Then the 1-dimensional array $\{((0),a),((k),a)\} \in \{a\}^{\mathbb{Z}^1}$ is m-connected, i.e., in $\{a\}^{\mathbb{Z}^1{}_m}$, if and only if $m \geq k$.

7.2 Array grammars on Cayley Grids

For a finitely presented group $G = \langle B \mid R \rangle$ with the set of elements G', we define an *array production* p over V and G as a triple $(W, \mathcal{A}_1, \mathcal{A}_2)$, where $W \subseteq G'$ is a finite set and $\mathcal{A}_1$ and $\mathcal{A}_2$ are mappings from W to $V \cup \{\#\}$ such that $shape(\mathcal{A}_1) \neq \emptyset$, where again the shape is defined to exactly contain the non-blank positions, i.e., $shape(\mathcal{A}_1) = \{ v \in W \mid \mathcal{A}(v) \neq \# \}$. We say that the array $\mathcal{C}_2 \in V^{C(G)}$ is *directly derivable* from the array $\mathcal{C}_1 \in V^{C(G)}$ by the array production $(W, \mathcal{A}_1, \mathcal{A}_2)$ if and only if there exists a $v \in G'$ such that, for all $w \in G' \setminus \tau_v(W)$, $\mathcal{C}_1(w) = \mathcal{C}_2(w)$, as well as, for all $w \in \tau_v(W)$, $\mathcal{C}_1(w) = \mathcal{A}_1(\tau_{-v}(w))$ and $\mathcal{C}_2(w) = \mathcal{A}_2(\tau_{-v}(w))$, i.e., the sub-array of $\mathcal{C}_1$ corresponding to $\mathcal{A}_1$ is replaced by $\mathcal{A}_2$, thus yielding $\mathcal{C}_2$; we also write $\mathcal{C}_1 \Longrightarrow_p \mathcal{C}_2$.

As we already see from the definitions of an array production, the conditions for an application to an array $\mathcal{B}$ and the result of an application to $\mathcal{B}$, an array production $(W, \mathcal{A}_1, \mathcal{A}_2)$ is a representative for the infinite set of equivalent array productions of the form $(\tau_v(W), \tau_v(\mathcal{A}_1), \tau_v(\mathcal{A}_2))$ with $v \in G'$. Hence, without loss of generality, we can assume $e \in W$ (e is the identity in G) as well as $\mathcal{A}_1(e) \neq \#$. Moreover, we often will omit the set W, because it is uniquely reconstructible from the description of the two mappings $\mathcal{A}_1$ and $\mathcal{A}_2$ by $\mathcal{A}_i = \{ (v, \mathcal{A}_i(v)) \mid v \in W \}$, for $1 \leq i \leq 2$. Thus, in the following, we represent the array production $(W, \mathcal{A}_1, \mathcal{A}_2)$ also by writing $\mathcal{A}_1 \to \mathcal{A}_2$, i.e., $\{(v, \mathcal{A}_1(v)) \mid v \in W\} \to \{(v, \mathcal{A}_2(v)) \mid v \in W\}$. If $|W| = 2$, i.e., $W = \{e, v\}$ for some $v \in G'$, then, for $\{(e, \mathcal{A}_1(e)), (v, \mathcal{A}_1(v))\} \to \{(e, \mathcal{A}_2(e)), (v, \mathcal{A}_2(v))\}$ we will only write $\mathcal{A}_1(e)\, v\mathcal{A}_1(v) \to \mathcal{A}_2(e)\, \mathcal{A}_2(v)$. If $|W| = 1$, i.e., $W = \{e\}$, we simply write $\mathcal{A}_1(e) \to \mathcal{A}_2(e)$.

$G_A = \left((N \cup T)^{C(G)}, T^{C(G)}, \mathcal{A}_0, P, \Longrightarrow_{G_A} \right)$ is called an array grammar over $C(G)$, where N is the alphabet of *non-terminal symbols*, T is the alphabet of *terminal symbols*, $N \cap T = \emptyset$; P is a finite non-empty set of array productions over V, where $V = N \cup T$; $\mathcal{A}_0 \in V^{C(G)}$ is the *initial array* (axiom), and $\Longrightarrow_{G_A}$ denotes the derivation relation induced by the array productions in P. In the following, we may omit $\Longrightarrow_{G_A}$ in the description of the array grammars.

In a more common notation, we also write an *array grammar* (over $C(G)$) as a septuple

$$G_A = (C(G), N, T, \#, P, \mathcal{A}_0, \Longrightarrow_{G_A}),$$

also specifying the background symbol $\# \notin N \cup T$, and, as usually done in the literature, we shall assume $\mathcal{A}_0 = \{(v_0, S)\}$, where $v_0 \in G'$ is the *start node*, and $S \in N$ is the *start symbol*.

We say that the array $\mathcal{B}_2 \in V^{C(G)}$ is *directly derivable* from the array $\mathcal{B}_1 \in V^{C(G)}$ in G_A, denoted $\mathcal{B}_1 \Longrightarrow_{G_A} \mathcal{B}_2$, if and only if there exists an array production $p = (W, \mathcal{A}_1, \mathcal{A}_2)$ in P such that $\mathcal{B}_1 \Longrightarrow_p \mathcal{B}_2$. Let $\Longrightarrow^*_{G_A}$ be the reflexive transitive closure of $\Longrightarrow_{G_A}$. The *array language generated* by the array grammar G_A, $L(G_A)$, is defined by $L(G_A) = \left\{ \mathcal{A} \mid \mathcal{A} \in T^{C(G)},\ \mathcal{A}_0 \Longrightarrow^*_{G_A} \mathcal{A} \right\}$.

An array production $p = (W, \mathcal{A}_1, \mathcal{A}_2)$ in P is called *#-context-free* (*of type* *#-CFA*), if $|shape(\mathcal{A}_1)| = 1$, i.e., $shape(\mathcal{A}_1) = \{e\}$, and $\mathcal{A}_1(e) \in N$.

For $X \in \{ARBA, \#\text{-}CFA\}$, an array grammar G is called to be of type X, if every array production in P is of the corresponding type, where $ARBA$ means that there are no restrictions on the form of the array productions. The family of k-connected array languages generated by array grammars on $C(G)$ of type X is denoted by $\mathcal{L}_k(C(G)\text{-}X)$; the family of arbitrary array languages generated by array grammars on $C(G)$ of type X is denoted by $\mathcal{L}(C(G)\text{-}X)$.

For arbitrary and #-context-free array grammars the condition to only consider languages of k-connected arrays corresponds to intersecting the generated array language with $V^{C(G)_k}$, which can be carried out by arbitrary array grammars by themselves (as, for example, proved in [10]), but is a condition imposed from "outside" when dealing with #-context-free array grammars. Yet as later we are going to show that some #-context-free array grammars equipped with specific control mechanisms can simulate any arbitrary array grammar this makes no difference any more in these cases.

Example 8. Let $G = \langle B \mid R \rangle$ be a finitely presented group and $x \in G$ with $ord(x) = \infty$. Let $b_1 \circ \ldots \circ b_k$ be the canonical representation of x in $\langle B \mid R \rangle$; then $(\{x^n \mid n \in \mathbb{Z}\}, \circ)$ is an infinite subgroup of G, and $x^n \neq x^m$ for $n \neq m$. Hence, along this "infinite line" we can argue many results obtained for $\mathbb{Z}^1$, e.g., how to embed simulations of Turing machine computations.

Remark 2. The possibility to compute along such infinite lines is also important if we want to (describe how to) simulate computations of a Turing machine – or similar computationally complete mechanisms (for strings) – using specific variants of (controlled) array grammars on Cayley graphs. For instance, for any computable finite group presentation of a group $\langle B \mid R \rangle$, we can effectively construct an encoding of any array language in $\mathcal{L}(C(G)\text{-}ARBA)$ given by an (arbitrary) array grammar and vice versa. The finite group presentation of the group $\langle B \mid R \rangle$ being computable is crucial for this result.

For simulating array grammars of type $C(G)\text{-}ARBA$, a special normal form we call *marked normal form* is very helpful; it has already been described for 1-dimensional array grammars in [11] as a special variant of the Chomsky normal form for array grammars, shown, for example, in [9], and exhibited for the general case of array grammars on Cayley grids in [10].

Lemma 2. (marked normal form)
For every array grammar of type $C(G)\text{-}ARBA$

$$G_A = (C(G), N, T, \#, P, \{(v_0, S)\}, \Longrightarrow_{G_A}),$$

we can effectively construct an equivalent array grammar of type $C(G)$*-*$ARBA$

$$\bar{G}_A = \left(C(G), N', T, \#, P', \left\{\left(v_0, \bar{S}\right)\right\}, \Longrightarrow_{\bar{G}_A}\right),$$

where $N \subseteq N'$ *and all array productions in* P' *are of one of the following forms:*

1. $\bar{A}B \to C\bar{D}$*, where* $A, B, C, D \in N' \cup T$*,* or
2. $\bar{\#} \to \#$*.*

Before the final array production $\bar{\#} \to \#$ *is applied, any intermediate array derived from the initial array* $\left\{\left(v_0, \bar{S}\right)\right\}$ *contains exactly one barred symbol.*

For applying the general results on the relations between different control mechanisms as elaborated in the rest of this section to array grammars of the types $C(G)$*-*$ARBA$ and $C(G)$*-#-*CFA, the following feature of these types is essential in some cases:

Lemma 3. *The types* $C(G)$*-*$ARBA$ *and* $C(G)$*-#-*CFA *– for a Cayley grid* $C(G)$ *– are strictly extended types with unit rules and trap rules.*

Proof. We first remark that, without loss of generality (e.g., see [10]), we may always assume that any array production contains at least one non-terminal symbol in the array on its left-hand side, i.e., in any array production $\{(v, \mathcal{A}_1(v)) \mid v \in W\} \to \{(v, \mathcal{A}_2(v)) \mid v \in W\}$ we find at least one $v_1 \in W$ such that $\mathcal{A}_1(v_1) \in N$; hence, $C(G)$*-*$ARBA$ can be assumed to be a strictly extended type for the succeeding proofs; $C(G)$*-#-*CFA is a strictly extended type already by definition. Now let

$$G_A = (C(G), N, T, \#, P, \{(v_0, S)\}, \Longrightarrow_{G_A})$$

be an array grammar of type $C(G)$*-*$ARBA$ or $C(G)$*-#-*CFA.

Then for every array production $p = (W, \mathcal{A}_1, \mathcal{A}_2)$ the corresponding *unit rule* is $p^+ = (W, \mathcal{A}_1, \mathcal{A}_1)$, which, when being applied, obviously does not change the underlying array.

Moreover, for the trap rules, take a new non-terminal symbol F, the *trap symbol*, which never can be erased any more, and for every array production $p = (W, \mathcal{A}_1, \mathcal{A}_2)$ we then define the corresponding *trap rule* $p^- = (W, \mathcal{A}_1, \mathcal{F}_W)$ with $\mathcal{F}_W(v) = F$ for all $v \in W$, which, when being applied, prohibits the derived array to become terminal no matter how the derivation proceeds.

In sum, we conclude that both $C(G)$*-*$ARBA$ and $C(G)$*-#-*CFA are strictly extended types with unit rules and trap rules. □

7.3 Results for Array Grammars on Cayley Grids

In many papers on control mechanisms for string grammars, the proof for showing that when using arbitrary productions any new control mechanism can be simulated is omitted, often simply citing the Church-Turing thesis, which usually is a legitimate claim as any formal proof would be tedious although bringing no

new insights. In case of array grammars on Cayley graphs the situation is more delicate: as long as the underlying group presentation is computable, one might still easily argue with the Church-Turing thesis as long as – for infinite groups – there is also an infinite path in the Cayley graph, which is obvious if there is a group element of infinite order – see Example 8 as well as Remark 2. Yet even if there is no such element (for examples of such group presentations we refer to [14]), in a nondeterministic way, we can find lines of arbitrary length for the necessary computations, as by definition the out-degree of every node is bounded, hence, by Königs infinity lemma such a path must exist; it is important to observe that these paths need not always be computable. Therefore, in the general case of Cayley grids we need an algorithm that works directly with the power inherent to arbitrary array productions. As, according to Theorem 7, GC_{ac} is the "strongest" control mechanism, only the following result is needed (for a proof, we refer to [10]):

Lemma 4. $\mathcal{L}\left(C\left(G\right)\text{-}ARBA\text{-}GC_{ac}\right) \subseteq \mathcal{L}\left(C\left(G\right)\text{-}ARBA\right).$

In connection with the results depicted in Theorem 7, from Lemma 4 we immediately infer the following:

Theorem 13. $\mathcal{L}\left(C\left(G\right)\text{-}ARBA\text{-}Y\right) = \mathcal{L}\left(C\left(G\right)\text{-}ARBA\right)$ *for any control mechanism* Y *in* $\{O, Pri, fC_1, fC, RC, MAT_{ac}, GC_{ac}^{allfinal}, GC_{ac}, GC^{allfinal}, GC, A, AB, pC, MAT, P, P_{ac}\}$.

Already an order relation on the rules is sufficient as a control mechanism to obtain $\mathcal{L}\left(C\left(G\right)\text{-}ARBA\right)$ with #-free array productions (see [10]):

Theorem 14. $\mathcal{L}\left(C\left(G\right)\text{-}ARBA\right) \subseteq \mathcal{L}\left(C\left(G\right)\text{-}\#\text{-}CFA\text{-}O\right).$

Proof. Let $G = \langle B \mid R\rangle$ be a finitely presented group and L be an array language on $C\left(G\right)$ given by an array grammar G_A in marked normal form, see Lemma 2. Moreover, let $G'_A = \left(C\left(G\right), N, T', \#, P, \{(v_0, S')\}, \Longrightarrow_{G'_A}\right)$ be the array grammar on $C\left(G\right)$ with $T' = \{X_a \mid a \in T\}$, i.e., we replace every terminal symbol $a \in T$ from G_A by a corresponding non-terminal symbol X_a in all the array productions of G_A. We now construct an equivalent ordered array grammar $G_O = (G_s, \prec, \Longrightarrow_{G_O})$ first simulating the derivations in G'_A corresponding to derivations in G_A with the only difference that instead of the terminal symbols $a \in T$ we have the corresponding non-terminal symbols X_a, and at the end these symbols X_a are transformed into the terminal symbols $a \in T$.

The main idea is to first generate a workspace of non-terminal symbols $X_\#$ representing the blank symbol surrounded with a border of symbols $\tilde{X}_\#$ also representing #; symbols $X_\#, \tilde{X}_\#$ still occurring in the derived array at the end of a simulation of a derivation in G'_A finally will be erased as to be described later in the proof. Moreover, at the very beginning, we generate a control symbol at some place, chosen in a non-deterministic way, not interfering with the workspace, but needed for the simulations of the application of rules in G'_A. The main task then is to show how a marked array production $\bar{A}vB \to C\bar{D}$, where $A, B, C, D \in N$, can be simulated by using a suitable order relation on the rules in G_O.

We first sketch how to obtain the control symbol and the workspace: Instead of starting with $\{(v_0, S)\}$ in G_A we start with the initial array $\{(v_0, S')\}$ in G'_A. Using any of the rules $S'v\# \to S''H_A$ for any $v \in B$ and then rules of the form $H_A u\# \to \#H_A$ for any $u \in B$, the initial control symbol H_A can move to any position (node) in the Cayley graph. Using the rule $H_A \to H_0$ ends this procedure and then allows the rule $S'' \to \tilde{S}$ to be applied, which is "dominated" by the rules in $H^- \setminus \{H_0 \to F\}$, i.e., $S'' \to \tilde{S} \prec p$ for all $p \in H^- \setminus \{H_0 \to F\}$, where $H^- = \{X \to F \mid X \in V_H\}$ and V_H denotes the set of all variants of the control variable H like H_A at the beginning.

Notation: In the following, the set of all trap rules "dominating" a rule p will be written as $P(p \prec)$, i.e., $P(p \prec) = \{q \mid p \prec q\}$.

In general, the idea with the variants of the control variable H is to guide the application of another rule p by, instead of checking for the presence of the specific variant H_α of H, ensuring the absence of all other variants of H, using the rule relations $p \prec q$ for all $q \in \{X \to F \mid X \in V_H \setminus \{H_\alpha\}\}$; hence, we also write $P(p \prec) = \{X \to F \mid X \in V_H \setminus \{H_\alpha\}\}$.

The next task is to generate sufficient workspace of symbols $X_\#$ surrounded by a layer of symbols $\tilde{X}_\#$ on the border to the remaining environment of blank symbols: We start with

$p_0 = \{(e, \tilde{S}\} \cup \{(v, \# \mid v \in B\} \to \{(e, \bar{S}\} \cup \{(v, \tilde{X}_\# \mid v \in B\}$,
$P(p_0 \prec) = \{X \to F \mid X \in V_H \setminus \{H_0\}\}$.

Iteratively, now a new "layer" of symbols $X_\#$ is added by first generating symbols $\hat{X}_\#$ from the symbols $\tilde{X}_\#$, then renaming the symbols $\tilde{X}_\#$ to $X_\#$ and finally renaming the symbols $\hat{X}_\#$ to $\tilde{X}_\#$, which is accomplished by the following rules p and the corresponding "dominating" set of rules $P(p \prec)$:

1. $H_0 \to H_1$, $P(H_0 \to H_1 \prec) = \{\tilde{S} \to F\}$;
2. for all $v \in B$,
   $p_v^1 = \{(e, \tilde{X}_\#), (v, \#)\} \to \{(e, \tilde{X}_\#), (v, \hat{X}_\#)\}$, $P(p_v^1 \prec) = \{X \to F \mid X \in V_H \setminus \{H_1\}\}$, $H_1 \to H_2$, $P(H_1 \to H_2 \prec) = \{p_v^{1^-} \mid v \in B\}$, where $p_v^{1^-}$ is the trap rule corresponding to the rule p_v^1, i.e., $p_v^{1^-} = \{(e, \tilde{X}_\#), (v, \#)\} \to \{(e, F), (v, F)\}$;
3. for all $v \in B$,
   $p_v^2 = \tilde{X}_\# \to X_\#$, $P(p_v^2 \prec) = \{X \to F \mid X \in V_H \setminus \{H_2\}\}$,
   $H_2 \to H_3$, $P(H_2 \to H_3 \prec) = \{p_v^{2^-} \mid v \in B\}$;
4. for all $v \in B$,
   $p_v^3 = \hat{X}_\# \to \tilde{X}_\#$, $P(p_v^3 \prec) = \{X \to F \mid X \in V_H \setminus \{H_3\}\}$,
   $H_3 \to H_1$, $P(H_3 \to H_1 \prec) = \{p_v^{3^-} \mid v \in B\}$; the iteration can start again with 2.
5. In order to stop the iteration, instead of $H_3 \to H_1$ we use the rule
   $H_3 \to H$, $P(H_3 \to H \prec) = \{p_v^{3^-} \mid v \in B\}$.

For the simulation in G_O we assume the marked array productions in G_A to be labeled, i.e., we write $p : \bar{A}_p v_p B_p \to C_p \bar{D}_p$.

1. We start the simulation of the application of $p : \bar{A}_p v_p B_p \to C_p \bar{D}_p$ with indicating the intention to do that by the rule $H \to H_p^1$ for the control symbol;
2. we continue with marking exactly one symbol B_p as B_p' by
   $p_1 = B_p \to B_p'$, $P(p_1 \prec) = \{X \to F \mid X \in (V_H \setminus \{H_p^1\}) \cup \{B_p'\}\}$,
   $H_p^1 \to H_p^2$, $P(H_p^1 \to H_p^2 \prec) = P_F$, $P_F = \{Xv\# \to FF \mid X \in N \cup \{X_\#\}, v \in B\}$,
   i.e., no blank symbol inside the workspace is allowed yet;
3. we now make a "#-hole" inside the workspace in such a way that the only non-terminal symbol having "access" to this blank position should be $\bar{A}_p$ by
   $p_2 = B_p' \to \#$, $P(p_2 \prec) = \{X \to F \mid X \in (V_H \setminus \{H_p^2\})\}$,
   $H_p^2 \to H_p^3$, $P(H_p^2 \to H_p^3 \prec) = P_F \setminus \{\bar{A}_p v_p \# \to FF\}$;
4. the "#-hole" made in the previous step now is filled correctly by
   $p_3 = \bar{A}_p v_p \# \to C_p \bar{D}_p$, $P(p_3 \prec) = \{X \to F \mid X \in (V_H \setminus \{H_p^3\})\}$,
   $H_p^3 \to H$, $P(H_p^3 \to H \prec) = P_F$.

Using the sequence of rules as described above, we finally have simulated the application of the rule $p : \bar{A}_p v_p B_p \to C_p \bar{D}_p$ and reached the control symbol H again, which allows us to continue with simulating the next rule. At some moment we have to check whether we can switch to the terminal procedure eliminating all non-terminal symbols from $X_\#, \tilde{X}_\#, \bar{X}_\#$ and transforming every non-terminal symbol X_a, $a \in T$, into the corresponding terminal symbol a:

1. We start with $H \to H_t$,
   $P(H \to H_t \prec) = \{X \to F \mid X \in (V \setminus (\{X_a \mid a \in T\} \cup \{X_\#, \tilde{X}_\#, \bar{X}_\#}))\}$;
2. for all $X \in \{X_\#, \tilde{X}_\#, \bar{X}_\#\}$, we take
   $p_X = X \to \#$, $P(p_X \prec) = \{X \to F \mid X \in (V_H \setminus \{H_t\})\}$;
3. for all $a \in T$, we take
   $p_a = X_a \to a$, $P(p_X \prec) = \{X \to F \mid X \in (V_H \setminus \{H_t\})\}$;
4. finally the control symbol H_t can be erased with
   $H_t \to \#$, $P(H_t \to \# \prec) = \{X \to F \mid X \in (V \setminus \{H_t\})\}$.

Based on the construction of G_O and the explanations given above we conclude $L(G_O) = L$. □

Looking at the general results collected in Theorem 7 we immediately infer the following results:

Corollary 5.
For any $Y \in \{O, Pri, fC_1, fC, RC, MAT_{ac}, GC_{ac}^{allfinal}, GC_{ac}, A, AB\}$,

$$\mathcal{L}(C(G)\text{-}ARBA) \subseteq \mathcal{L}(C(G)\text{-}\#\text{-}CFA\text{-}Y).$$

A similar result can be shown for programmed array grammars by proving the following equality (for a proof, see [10]):

Lemma 5. $\mathcal{L}(C(G)\text{-}\#\text{-}CFA\text{-}PC_{ac}) = \mathcal{L}\left(C(G)\text{-}\#\text{-}CFA\text{-}GC_{ac}^{allfinal}\right)$.

Combining all the general results depicted in this section, we obtain the main theorem for sequential array grammars on Cayley graphs of finitely presented groups with control mechanisms:

Theorem 15.
For any $Y \in \left\{O, Pri, fC_1, fC, RC, MAT_{ac}, GC_{ac}^{allfinal}, GC_{ac}, A, AB, P_{ac}\right\}$,

$$\mathcal{L}\left(C\left(G\right)\text{-}\#\text{-}CFA\text{-}Y\right) = \mathcal{L}\left(C\left(G\right)\text{-}ARBA\right).$$

Similar results hold for languages of k-connected arrays:

Theorem 16.
For any $Y \in \left\{O, Pri, fC_1, fC, RC, MAT_{ac}, GC_{ac}^{allfinal}, GC_{ac}, A, AB, P_{ac}\right\}$,

$$\mathcal{L}_k\left(C\left(G\right)\text{-}\#\text{-}CFA\text{-}Y\right) = \mathcal{L}_k\left(C\left(G\right)\text{-}ARBA\right).$$

8 Cooperating Distributed Grammar Systems

Basic results on the generating power of hybrid cooperating distributed grammar systems were established by Mitrana ([18]) and by Păun ([19]); a general overview on this area of formal language theory is given in the monograph by Csuhj-Varjú, Dassow, Kelemen, and Păun ([6]).

Let $G = (O, O_T, w, P, \Longrightarrow_G)$ be a grammar of type X; for the basic derivation modes from

$$B = \{*, t\} \cup \{\leq k, = k, \geq k \mid k \geq 1\}$$

and any objects $u, v \in O$ we define

- $u \Longrightarrow_G^* v$ to denote the usual reflexive and transitive closure of $\Longrightarrow_G$;
- $u \Longrightarrow_G^t v$ if and only if $u \Longrightarrow_G^* v$ and no rule from P is applicable to v;
- $u \Longrightarrow_G^{\leq k} v$, $u \Longrightarrow_G^{=k} v$, $u \Longrightarrow_G^{\geq k} v$ if and only if $u \Longrightarrow_G^* v$ in at most k, exactly k, at least k derivation steps.

A *hybrid cooperating distributed grammar system* (*HCDG system* for short) G_{HCDG} *of degree* n *and type* X *working in the derivation modes from* $B' \subseteq B$ is a construct

$$G_{HCDG} = (G, P_1, \ldots, P_n, f_1, \ldots, f_n, \Longrightarrow_{G_{HCDG}})$$

where $P_i \subseteq P$ and $f_i \in B'$ for $1 \leq i \leq n$, $\cup_{i=1}^n P_i = P$, and the grammars G and $G_i = (O, O_T, w, P_i, \Longrightarrow_{G_i})$, $1 \leq i \leq n$, are grammars of type X, the derivation relations $\Longrightarrow_{G_i}$ being the restrictions of $\Longrightarrow_G$ only induced by the corresponding rule sets P_i. For any $u, v \in O$, we define $u \Longrightarrow_{G_{HCDG}} v$ if and only if $u \Longrightarrow_{G_i}^{f_i} v$ for some i, $1 \leq i \leq n$. We remark that the component P_i, i.e., the grammar G_i, in each step of the derivation in G_{HCDG} is chosen in a non-deterministic way, which also means that even the same component may be taken several times in a row.

A *cooperating distributed grammar system* (*CDG system* for short) G_{CDG} *of degree* n *and type* X *working in the derivation mode* f with $f \in B$ is a special case of a hybrid cooperating distributed grammar system where all derivation modes f_i equal f, i.e., a construct $G_{CDG} = (G, P_1, \ldots, P_n, f, \Longrightarrow_{G_{CDG}})$ where $P_i \subseteq P$ for $1 \leq i \leq n$, $\cup_{i=1}^{n} P_i = P$, and the grammars G and $G_i = (O, O_T, w, P_i, \Longrightarrow_{G_i})$, $1 \leq i \leq n$, are grammars of type X. For any $u, v \in O$, we define $u \Longrightarrow_{G_{CDG}} v$ if and only if $u \Longrightarrow_{G_i}^{f} v$ for some i, $1 \leq i \leq n$.

The language generated by the HCDG system G_{HCDG} is defined by $L(G_{HCDG}) = \left\{v \in O_T \mid w \Longrightarrow_{G_{HCDG}}^{*} v\right\}$. The family of languages generated by hybrid grammar systems of degree n and of type X working in derivation modes from B' is denoted by $\mathcal{L}\left(X\text{-}HCDG_n\left(B'\right)\right)$, the family of languages generated by grammar systems of degree n and of type X working in the derivation mode f is denoted by $\mathcal{L}\left(X\text{-}CDG_n\left(f\right)\right)$; in both cases we replace n by $*$ if we consider arbitrary degrees.

As a special subset of derivation modes, we consider $B_0 = \{*, = 1, \geq 1\} \cup \{\leq k \mid k \geq 1\}$, for which the following result holds:

Theorem 17. *For any type* X*, any* $B' \subseteq B_0$*, and any* $n \geq 1$*,*

$$\mathcal{L}\left(X\right) = \mathcal{L}\left(X\text{-}HCDG_n\left(B'\right)\right).$$

Proof. Consider any HCDG system

$$G_{HCDG} = (G, P_1, \ldots, P_n, f_1, \ldots, f_n, \Longrightarrow_{G_{HCDG}})$$

where the underlying grammar is $G = \left(O, O_T, w, P, \Longrightarrow_G\right)$, $P_i \subseteq P$ and $f_i \in B'$ for $1 \leq i \leq n$, $\cup_{i=1}^{n} P_i = P$, and the grammars G and $G_i = (O, O_T, w, P_i, \Longrightarrow_{G_i})$, $1 \leq i \leq n$, are grammars of type X. Now any derivation $u \Longrightarrow_{G_i}^{f_i} v$ for some i, $1 \leq i \leq n$, using n steps can also be obtained by using n times the derivation mode $= 1$ with grammar G_i, which holds for every derivation mode f_i from B_0. On the other hand, every derivation mode f_i from B_0 allows for making only one derivation step before changing to any grammar G_j, $1 \leq j \leq n$.

As $\cup_{i=1}^{n} P_i = P$, we obtain $L(G) = L(G_{HCDG})$ for any such hybrid cooperating distributed grammar system G_{HCDG} over the set of derivation modes B_0, which observation concludes the proof. □

Theorem 18. *For any strictly extended type* X*,* $\mathcal{L}\left(X\right) = \mathcal{L}\left(X\text{-}CDG_1\left(t\right)\right)$*.*

Proof. Consider a CDG system $G_{CDG} = (G, P, t, \Longrightarrow_{G_{CDG}})$ with $G = (O, O_T, w, P, \Longrightarrow_G)$ being the underlying grammar of type X. As X is a strictly extended type, any derivation in G leading to a terminal object w is maximal, i.e., $w \in L\left(G_{CDG}\right)$, hence, $L\left(G\right) \subseteq L\left(G_{CDG}\right)$. On the other hand, as in G_{CDG} we only have one component using exactly the same rules as in G, we also have $L\left(G_{CDG}\right) \subseteq L\left(G\right)$, hence, we conclude $L\left(G\right) = L\left(G_{CDG}\right)$, and therefore $\mathcal{L}\left(X\right) = \mathcal{L}\left(X\text{-}CDG_1\left(t\right)\right)$. □

The equality relation established in the preceding theorem between $\mathcal{L}\left(X\right)$ and $\mathcal{L}\left(X\text{-}CDG_1\left(t\right)\right)$ need not be true for pure types, as the following simple example shows:

Example 9. Consider the grammar $G = \left(\{a\}^+, \{a\}^+, a, P, \Longrightarrow_G\right)$ with the set of rules $P = \left\{a \to a^2, a \to \lambda\right\}$ to constitute the simple pure type X_1.

Obviously, $L(G) = \{a\}^*$, hence, we get $\mathcal{L}(X_1) = \left\{\{a\}^*\right\}$.

The only $CDG_1(t)$ system of type X_1 is $G_{CDG} = (G, P, t, \Longrightarrow_{G_{CDG}})$, but $L(G_{CDG}) = \{\lambda\}$, because every terminating derivation in G_{CDG} must end in λ. Hence, $\mathcal{L}(X_1\text{-}CDG_1(t)) = \{\{\lambda\}\}$.

Thus, we obtain $\mathcal{L}(X_1) = \left\{\{a\}^*\right\} \neq \{\{\lambda\}\} = \mathcal{L}(X_1\text{-}CDG_1(t))$.

Again, the computational power of HCDG systems can be captured by GC_{ac} as control mechanism, which according to Theorem 7 is the "strongest" one.

Lemma 6. *Let X be any strictly extended type X, $B' \subseteq B$, $n \geq 1$, and*

$$G_{HCDG} = (G, P_1, \ldots, P_n, f_1, \ldots, f_n, \Longrightarrow_{G_{HCDG}})$$

be an arbitrary HCDG system, where $G = (O, O_T, w, P, \Longrightarrow_G)$ is the underlying grammar of type X, $P_i \subseteq P$, $f_i \in B'$ for $1 \leq i \leq n$, $\cup_{i=1}^{n} P_i = P$, and the grammars $G_i = (O, O_T, w, P_i, \Longrightarrow_{G_i})$, $1 \leq i \leq n$, are grammars of type X. Then we can construct an equivalent graph-controlled grammar (with applicability checking) of type X $G_{GC} = (G, g, H_i, H_f, \Longrightarrow_{GC})$ such that $L(G_{HCDG}) = L(G_{GC})$. Moreover, if $t \notin B'$, then applicability checking is not needed in G_{GC}.

Proof. Given the HCDG system G_{HCDG} and its underlying grammar G, which is the underlying grammar of the graph-controlled grammar G_{GC}, too, for G_{GC} we only have to specify the control graph $g = (H, E, K)$ as well as the sets $H_i \subseteq H$ and $H_f \subseteq H$ of initial and final labels, respectively. This can be achieved by defining subgraphs of g, which are constructed using different graphs for each derivation mode $f \in B$; for every $1 \leq i \leq n$, the nodes in the subgraphs described in the following then have assigned the set of rules P_i by K, whatever the corresponding $f_i \in B$ may be. The nodes in the graphs are of one of the following types:

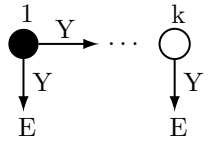
normal node node label $n \in H$: n ◯

initial node node label $n \in H_i$: n ●

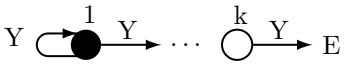
In the whole control graph, there is only one *final node* which has no rules assigned to, and its label is the only one in H_f. The simulation of derivation modes in B now can be described as follows (for the derivation modes $*$ and ≤ 1 we assume that at least one derivation step is made):

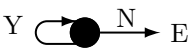
derivation mode $= 1$ or ≤ 1: ● —Y→ E

derivation mode $*$ or ≥ 1: Y ↻ ● —Y→ E

derivation mode $\leq k$, $k \geq 2$: 

derivation mode $= k$, $k \geq 2$: 

derivation mode $\geq k$, $k \geq 2$: 

derivation mode t: 

Putting together the subgraphs for the components

$$G_i = (O, O_T, w, P_i, \Longrightarrow_{G_i}), 1 \leq i \leq n,$$

we obtain the complete control graph $g = (H, E, K)$ by letting every edge pointing to E leading to the initial nodes of all the subgraphs as well as to the *final node*.

We finally observe that only in the construction of the subgraph for the derivation mode t an edge labeled by N is needed, i.e., only in this case applicability checking is needed, which observation completes the proof. □

As an immediate consequence of the preceding result, we obtain the following:

Theorem 19. *For any strictly extended type X, any $B' \subseteq B$, and any $n \geq 1$,*

$$\mathcal{L}(X\text{-}HCDG_n(B')) \subseteq \mathcal{L}(X\text{-}GC_{ac}).$$

If $t \notin B'$, then we even have $\mathcal{L}(X\text{-}HCDG_n(B')) \subseteq \mathcal{L}(X\text{-}GC)$.

9 Summary and Future Research

The formal framework for sequential grammars with regulated rewriting based on the applicability of rules has first been presented in a comprehensive way in [12] and recently extended in several papers, especially with the new concept of activation and blocking of rules, see [10] and [3].

Based on the general results obtained within this framework, many computational completeness results for sequential grammars working on strings or multisets, but also for sequential array grammars on Cayley grids can be shown.

There are still many other control mechanisms which might perfectly fit to be considered within this framework, for example, regular control or other derivation modes known from the area of grammar systems. Investigations how to include further control mechanisms as well as to prove relations between them and the control mechanisms considered so far thus remain as a challenge for future research.

Acknowledgements

I am very grateful to my colleagues and co-authors for many fruitful discussions as well as for their contributions to the topics described in this overview paper: First parts for the concept of the general framework were already discussed and elaborated during my stay in Magdeburg with Jürgen Dassow nearly thirty years ago. Afterwards, partial results were used in several papers, for example, with Henning Fernau, Markus Holzer, and Gheorghe Păun. The first comprehensive collection of results in [12] then was elaborated with my colleagues in Vienna, Marion Oswald and Marian Kogler. Recent results, especially for sequential grammars with activation and blocking of rules (see [3] and [2]), were elaborated together with Artiom Alhazov and Sergiu Ivanov.

References

1. Aizawa, K., Nakamura, A.: Grammars on the hexagonal array. In: Wang, P.S.P. (ed.) Array Grammars, Patterns and Recognizers, Series in Computer Science, vol. 18, pp. 144–152. World Scientific (1989). https://doi.org/10.1142/S0218001489000358
2. Alhazov, A., Freund, R., Ivanov, S.: P systems with activation and blocking of rules. In: Stepney, S., Verlan, S. (eds.) Unconventional Computation and Natural Computation – 17th International Conference, UCNC 2018, Fontainebleau, France, June 25-29, 2018, Proceedings. Lecture Notes in Computer Science, vol. 10867, pp. 1–15. Springer (2018). https://doi.org/10.1007/978-3-319-92435-9_1
3. Alhazov, A., Freund, R., Ivanov, S.: Sequential grammars with activation and blocking of rules. In: Machines, Computations, and Universality – 8th International Conference, MCU 2018, Fontainebleau, France, June 28-30, 2018, Proceedings. Lecture Notes in Computer Science, vol. 10881, pp. 51–68 (2018). https://doi.org/10.1007/978-3-319-92402-1_3
4. Cavaliere, M., Freund, R., Oswald, M., Sburlan, D.: Multiset random context grammars, checkers, and transducers. Theoretical Computer Science **372**(2–3), 136–151 (2007). https://doi.org/10.1016/j.tcs.2006.11.022
5. Cook, C.R., Wang, P.S.P.: A Chomsky hierarchy of isotonic array grammars and languages. Computer Graphics and Image Processing **8**, 144–152 (1978). https://doi.org/10.1016/S0146-664X(78)80022-7
6. Csuhaj-Varjú, E., Dassow, J., Kelemen, J., Păun, Gh.: Grammar Systems: A Grammatical Approach to Distribution and Cooperation. Gordon and Breach Science Publishers (1994)
7. Dassow, J., Păun, Gh.: Regulated Rewriting in Formal Language Theory, EATCS Monographs in Theoretical Computer Science, vol. 18. Springer (1989)
8. Fernau, H., Freund, R., Oswald, M., Reinhardt, K.: Refining the nonterminal complexity of graph-controlled, programmed, and matrix grammars. Journal of Automata, Languages and Combinatorics **12**(1–2), 117–138 (2007)
9. Freund, R.: Control mechanisms on #-context-free array grammars. In: Păun, Gh. (ed.) Mathematical Aspects of Natural and Formal Languages, pp. 97–137. World Scientific Publ., Singapore (1994). https://doi.org/10.1142/9789814447133_0006
10. Freund, R.: Control mechanisms for array grammars on Cayley grids. In: Machines, Computations, and Universality – 8th International Conference, MCU 2018,

Fontainebleau, France, June 28-30, 2018, Proceedings. Lecture Notes in Computer Science, vol. 10881, pp. 1–33 (2018). https://doi.org/10.1007/978-3-319-92402-1_1
11. Freund, R., Ivanov, S., Oswald, M., Subramanian, K.G.: One-dimensional array grammars and P systems with array insertion and deletion rules. In: Neary, T., Cook, M. (eds.) Proceedings Machines, Computations and Universality 2013, MCU 2013, Zürich, Switzerland, September 9-11, 2013. EPTCS, vol. 128, pp. 62–75 (2013). https://doi.org/10.4204/EPTCS.128
12. Freund, R., Kogler, M., Oswald, M.: A general framework for regulated rewriting based on the applicability of rules. In: Kelemen, J., Kelemenová, A. (eds.) Computation, Cooperation, and Life – Essays Dedicated to Gheorghe Păun on the Occasion of His 60th Birthday. Lecture Notes in Computer Science, vol. 6610, pp. 35–53. Springer (2011). https://doi.org/10.1007/978-3-642-20000-7_5
13. Freund, R., Oswald, M.: Array automata on Cayley grids. In: Neary, T., Cook, M. (eds.) Proceedings Machines, Computations and Universality 2013 Zürich, Switzerland, 9/09/2013 - 11/09/2013. EPTCS, vol. 128, pp. 27–28 (2013). https://doi.org/10.4204/EPTCS.128
14. Freund, R., Oswald, M.: Array grammars and automata on Cayley grids. Journal of Automata, Languages and Combinatorics **19**(1-4), 67–80 (2014). https://doi.org/10.25596/jalc-2014-067
15. Holt, D.F., Eick, B., O'Brien, E.A.: Handbook of Computational Group Theory. CRC Press (2005)
16. Kudlek, M., Martín-Vide, C., Păun, Gh.: Toward a formal macroset theory. In: Calude, C.S., Păun, Gh., Rozenberg, G., Salomaa, A. (eds.) Multiset Processing – Mathematical, Computer Science and Molecular Computing Points of View, Lecture Notes in Computer Science, vol. 2235, pp. 123–134. Springer (2001). https://doi.org/10.1007/3-540-45523-X_7
17. Minsky, M.L.: Computation: Finite and Infinite Machines. Prentice Hall, Englewood Cliffs, New Jersey (1967)
18. Mitrana, V.: On the generative capacity of hybrid CD grammar systems. Computers and Artificial Intelligence **12**(1), 231–244 (1993)
19. Păun, Gh.: Hybrid cooperating/distributed grammar systems. J. Inform. Process. Cybernet. EIK **30**(4), 231–244 (1994)
20. Păun, Gh., Rozenberg, G., Salomaa, A.: The Oxford Handbook of Membrane Computing. Oxford University Press, Inc., New York, NY, USA (2010)
21. Rosenfeld, A.: Picture Languages. Academic Press, Reading, MA (1979)
22. Rosenfeld, A., Siromoney, R.: Picture languages – a survey. Languages of Design **1**(3), 229–245 (1993), `http://dl.acm.org/citation.cfm?id=198440.198442`
23. Rozenberg, G., Salomaa, A. (eds.): Handbook of Formal Languages, 3 volumes. Springer, Berlin (1997)
24. Salomaa, A.: Formal Languages. Academic Press (1973)
25. Wang, P.S.P.: An application of array grammars to clustering analysis for syntactic patterns. Pattern Recognition **17**, 441–451 (1984). https://doi.org/10.1016/0031-3203(84)90073-6