Control Mechanisms for Array Grammars on Cayley Grids

Rudolf Freund(✉)

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

Abstract. In this paper, the computational power of several control mechanisms for specific variants of (sequential, isometric) array grammars generating arrays on Cayley grids of finitely presented groups is investigated. Using #-context-free array productions together with control mechanisms as control graphs, matrices, permitting and forbidden rules, partial order on rules or activation and blocking of rules the same computational power is obtained as when using arbitrary array productions.

1 Introduction

As a natural extension of string languages (e.g., see [31,32]), arrays on the d-dimensional grid $\mathbb{Z}^d$ have been introduced and investigated since more than four decades, for example, see [6,26]. Applications of array grammars and array automata especially can be found in the area of pattern and picture recognition, for instance, see [29,30,33].

Following some ideas of Csuhaj-Varjú and Mitrana [7], the investigation of arrays on Cayley grids of finitely presented groups was started in [22], presented at MCU 2013 in Zürich, Switzerland; first definitions and results for array automata on Cayley grids can be found there. Array grammars and automata on Cayley grids then were investigated in more detail in [23]. 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 paper, 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 [23]. Following the general notions for regulated rewriting based on the applicability of rules as introduced in [21], then the control mechanisms using control graphs, matrices, permitting and forbidden rules, partial order on rules or activation and blocking of rules are defined. We elaborate some relations between these control mechanisms in the general setting of sequential grammars as already done in [21] and also prove some new ones. When using #-context-free array productions in the underlying array grammars, together with any of these control mechanisms, the same computational power as with arbitrary array productions can be obtained.

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 $|M|$.

For the basic notions and results of formal language theory the reader is referred to the monographs and handbooks in this area as [8,31,32], and for the basics of group theory and group presentations to [25].

2.1 Groups and Group Presentations

Now 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 $a \circ a^{-1} = a^{-1} \circ a = e$.

Moreover, the group is called *commutative*, if for any $a, b \in G'$, $a \circ b = b \circ a$. In the following, we will not distinguish between G' and G if the group operation is obvious from the context.

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., $B^{-1} = \{b^{-1} \mid b \in B\}$. We now consider the strings in $(B \cup B^{-1})^*$ and two strings as different unless their equality follows from the group axioms, i.e., for any $a, b, c \in (B \cup B^{-1})^*$, $abb^{-1}c = ac$; using these reductions, we obtain a set of irreducible strings from those in $(B \cup B^{-1})^*$, 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, i.e., $a = b_1 \circ \cdots \circ b_m$ for $b_1, \ldots, b_m \in B$. 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 these generators. In a similar way, the group $G = \langle B \mid R \rangle$ is generated by B, we here consider the strings in B^* 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 $b_1, \ldots, b_m, c \in B$, 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 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 $\mathbb{Z}/\mathbb{Z}_n$.

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 $(0, \ldots, 1, \ldots, 0)$ and their inverses $(0, \ldots, -1, \ldots, 0)$. 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 $\mathbb{Z}$.

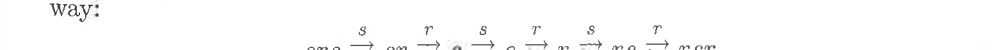
Remark 1. Given a finite presentation of a group $\langle B \mid R \rangle$, in general it is not even decidable whether the group presented in that way is finite or infinite. If we consider (infinite) groups where the word equivalence problem $u = v$ is decidable, or equivalently, there is a decision 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 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 $xa = y$.

As can be seen directly from the definition, the Cayley graph for a group G depends on its presentation by the generator set B and the relators in R.

Example 4. The dihedral group D_∞ corresponds with the Cartesian product of $\mathbb{Z}$ and $\mathbb{Z}_2$. One presentation of D_∞ is as $\left\langle r, s \mid s^2, (sr)^2 \right\rangle$, another one is $\left\langle r, s \mid r^2, s^2 \right\rangle$, the Cayley graph of which can be represented easily in the following way:



In this presentation, both generators have order two; on the other hand, an infinite line can be obtained by taking the group element rs and its powers $(rs)^n$ for $n \geq 0$, as the order of rs is infinite.

The Cayley graph for the presentation of D_∞ as $\langle r, s \mid s^2, srsr \rangle$ can be depicted as follows:

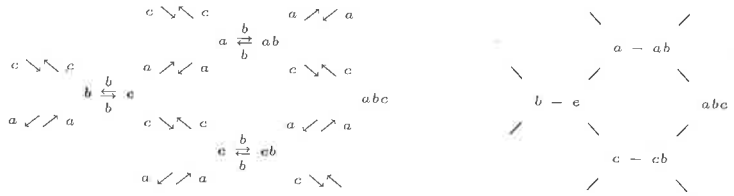
$$\begin{array}{ccccccccccccc}
\ldots & sr^2 & \xleftarrow{r} & sr & \xleftarrow{r} & s & \xleftarrow{r} & sr^{-1} & \xleftarrow{r} & sr^{-2} & \ldots \\
 & s \downarrow\uparrow s & & s \downarrow\uparrow s & & s \downarrow\uparrow s & & s \downarrow\uparrow s & & s \downarrow\uparrow s & \\
\ldots & r^{-2} & \xrightarrow{r} & r^{-1} & \xrightarrow{r} & e & \xrightarrow{r} & r & \xrightarrow{r} & r^2 & \ldots
\end{array}$$

As s is self-inverse, instead of the two directed edges $s \downarrow\uparrow s$ often only the corresponding non-directed edge $\mid s$ is depicted, i.e.,

$$\begin{array}{ccccccccccccc}
\ldots & sr^2 & \xleftarrow{r} & sr & \xleftarrow{r} & s & \xleftarrow{r} & sr^{-1} & \xleftarrow{r} & sr^{-2} & \ldots \\
 & \mid s & & \mid s & & \mid s & & \mid s & & \mid s & \\
\ldots & r^{-2} & \xrightarrow{r} & r^{-1} & \xrightarrow{r} & e & \xrightarrow{r} & r & \xrightarrow{r} & r^2 & \ldots
\end{array}$$

The lower and the upper lines are going into opposite directions, which nicely fits as a representation of double-stranded DNA molecules, i.e., the lower line going from the left 5′-end to the right 3′-end, whereas the complementary upper line goes from the right 5′-end to the left 3′-end.

Example 5. 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 $a \nearrow\swarrow a$, $\overset{b}{\underset{b}{\rightleftarrows}}$, and $c \searrow\nwarrow c$ by $/$, $-$, and $\backslash$, respectively. Both representations are depicted in the following:



3 Arrays and Array Grammars

In this section 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 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 $v = w_1 \rightarrow \ldots \rightarrow w_n = w$ in $C(G)$ from v to w also its inverse $w = w_n \rightarrow \ldots \rightarrow w_1 = v$ is a path in $C(G)$.

A finite *array* $\mathcal{A}$ over an alphabet V on G' is a function $\mathcal{A} : G' \rightarrow V \cup \{\#\}$, where $shape(\mathcal{A}) = \{v \in G' \mid \mathcal{A}(v) \ldots$

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 $V^{\langle G \rangle}$ 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 $V^{C(G)}$.

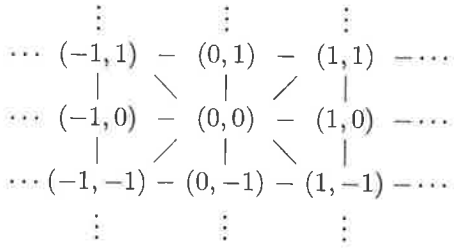
The *empty array* in V^G has empty shape and is denoted by Λ_G. Ordering the generators in B in a specific way as $e_1 < \cdots < e_m$, for each array $\mathcal{A} = \{(v, \mathcal{A}(v)) \mid v \in shape(\mathcal{A})\}$ in $V^{\langle G \rangle} \setminus \{\Lambda_G\}$ we get a canonical representation 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 6. Consider the hexagonal grid from Example 5. 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 7. A *d-dimensional array* is an array over the free group $\mathbb{Z}^d$. If we take the unit vectors $e_k = (0, \ldots, 1, \ldots, 0)$ and their inverses $(0, \ldots, -1, \ldots, 0)$, the resulting Cayley graph is the well-known d-dimensional grid, which in the 2-dimensional case can be depicted in the following way, where each horizontal line $-$ represents the two directed edges $\overset{(1,0)}{\underset{(-1,0)}{\rightleftarrows}}$ and each vertical line $\mid$ represents the two directed edges $(0,-1) \downarrow\uparrow (0,1)$:

$$\begin{array}{ccccccccccc}
 & \vdots & & \vdots & & \vdots & & \vdots & & \vdots & \\
 & \mid & & \mid & & \mid & & \mid & & \mid & \\
\cdots - & (-2,1) & - & (-1,1) & - & (0,1) & - & (1,1) & - & (2,1) & - \cdots \\
 & \mid & & \mid & & \mid & & \mid & & \mid & \\
\cdots - & (-2,0) & - & (-1,0) & - & (0,0) & - & (1,0) & - & (2,0) & - \cdots \\
 & \mid & & \mid & & \mid & & \mid & & \mid & \\
\cdots - & (-2,-1) & - & (-1,-1) & - & (0,-1) & - & (1,-1) & - & (2,-1) & - \cdots \\
 & \mid & & \mid & & \mid & & \mid & & \mid & \\
 & \vdots & & \vdots & & \vdots & & \vdots & & \vdots &
\end{array}$$

With respect to the origin $(0,0)$, the four vectors $(1,0), (-1,0), (0,1), (0,-1)$ are known as the *von Neumann neighborhood*, whereas adding the diagonal positions $(1,1), (-1,1), (-1,-1), (1,-1)$ yields the *Moore neighborhood* and thus a



For any $v \in G'$, the *translation* $\tau_v : G' \rightarrow G'$ 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

$$(\tau_v(\mathcal{A}))(w) = \mathcal{A}\left(w \circ v^{-1}\right) \text{ for all } w \in G'.$$

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 $v = w_1 \rightarrow \cdots \rightarrow w_n = w$ in $C(G)$ with $\{w_1, \ldots, w_n\} \subseteq$ *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 8. Consider the set of one-dimensional arrays over the alphabet $\{a\}^*$, i.e., $\{a\}^{C(((1),(-1)))}$, 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 $m \geq k$.

3.1 Array Grammars

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

$\mathcal{A}_2$ 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 $\mathcal{A}_1 \rightarrow \mathcal{A}_2$, i.e., $\{(v, \mathcal{A}_1(v)) \mid v \in W\} \rightarrow \{(v, \mathcal{A}_2(v)) \mid v \in W\}$. If $|W| = 2$, i.e., $W = \{e, v\}$ for some $v \in G'$, then, for $\{(e, \mathcal{A}_1(e)), (v, \mathcal{A}_1(v))\} \rightarrow \{(e, \mathcal{A}_2(e)), (v, \mathcal{A}_2(v))\}$ we will only write $\mathcal{A}_1(e) v \mathcal{A}_1(v) \rightarrow \mathcal{A}_2(e) \mathcal{A}_2(v)$. If $|W| = 1$, i.e., $W = \{e\}$, we simply write $\mathcal{A}_1(e) \rightarrow \mathcal{A}_2(e)$.

An *array grammar* (over $C(G)$) is a septuple

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

where N is the alphabet of *non-terminal symbols*, T is the alphabet of *terminal symbols*, $N \cap T = \emptyset$, $\# \notin N \cup T$; 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), for which, 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*. Moreover, $\Longrightarrow_{G_A}$ denotes the derivation relation induced by the array productions in P. In the examples given below, we will omit $\Longrightarrow_{G_A}$ in the description of the array grammars.

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 $\mathcal{A}_1(e) \in N$;
- *context-free* (*of type CFA*), if it is of type *#-CFA* and $\mathcal{A}_2(e) \neq \#$;
- *strictly context-free* (*of type SCFA*), if it is of type *#-CFA* and *shape* $(\mathcal{A}_2) = W$.

For $X \in \{ARBA, \#\text{-}CFA, CFA, SCFA\}$, 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 $\mathcal{L}(C(G)\text{-}X)$.

The main difference between array grammars of type *#-CFA* and of type *CFA* is that already by the definition of the array productions of type *CFA* it is guaranteed that all intermediate arrays derived from the initial arrays as well as the terminal arrays are k-connected, if all W in the array productions $(W, \mathcal{A}_1; \mathcal{A}_2)$ are k-connected due to the condition that no symbol can be *erased*, i.e., replaced by the blank symbol #. On the other hand, array productions of type *#-CFA* allow symbols to move as far as they want from their original

Example 9. Consider the #-context-free 1-dimensional array grammar

$$G_1 = \left(\mathbb{Z}^1, N = \{S, A\}, T = \{a\}, \#, P, \mathcal{A}_0 = \{((0), S)\}\right),$$
$$P = \{(0)S(1)\# \to (0)a(1)A, (0)A(1)\# \to (0)\#(1)A, (0)A \to (0)a\}.$$

According to our conventions, in a simpler way we can write

$$P = \{S(1)\# \to aA, A(1)\# \to \#A, A \to a\}.$$

The array language generated by G_1 is the subset of $\{a\}^{\mathbb{Z}^1}$ that can be written as $\{(0)a(m)a \mid m \in \mathbb{N}\}$ and thus for no k and no type X of array grammars is in $\mathcal{L}_k\left(\mathbb{Z}^1 - X\right)$.

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 (which will be proved later, see Lemma 2), 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.

Example 10. 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 "line" we can argue many results obtained for $\mathbb{Z}^1$, e.g., we can argue that, for any Cayley grid $C(\langle B \mid R\rangle)$,

$$\mathcal{L}(C(\langle B \mid R\rangle), CFA) \subset \mathcal{L}(C(\langle B \mid R\rangle), ARBA),$$

because the inclusion directly follows from the definitions, and the strictness follows from the well-known corresponding result for string languages using as a witness the language $(L) = \left\{a^{2^n} \mid n \in \mathbb{N}\right\}$ and the representation of the strings in it as 1-dimensional arrays. As a small technical detail we have to mention that for $x = b_1 \circ \ldots \circ b_k$, $b_1, \ldots, b_k \in B$, such witness languages have to be expanded by the homomorphism h_k with $h_k(a) = a^k$ for every symbol a in the alphabet, as in $C(\langle B \mid R\rangle)$ we now have to fill k positions instead of only one in $\mathbb{Z}^1$.

Such infinite lines can be found in various Cayley graphs described so far. For example, consider the presentation of D_∞ as $\langle r, s \mid r^2, s^2\rangle$ from Example 4; its Cayley graph $\ldots srs \underset{s}{\rightleftarrows} sr \underset{r}{\rightleftarrows} s \underset{s}{\rightleftarrows} e \underset{r}{\rightleftarrows} r \underset{s}{\rightleftarrows} rs \underset{r}{\rightleftarrows} rsr \ldots$ can somehow be seen as the line $(rs)^n$, $n \in \mathbb{Z}$, when only taking the elements having a canonical representation of even length.

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

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 $C(G)$-$ARBA$, a special normal form we call *marked normal form* is very helpful; it has already been described for 1-dimensional array grammars in [20] as a special variant of the Chomsky normal form for array grammars, shown, for example, in [16].

Lemma 1 (marked normal form). *For every array grammar of type $C(G)$-$ARBA$*

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

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

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

such that $N \subseteq N'$ 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.

We omit the proof as the arguments given in [20] for 1-dimensional array grammars can be taken over for the general case of arbitrary array grammars over Cayley grids.

We now exhibit the promised algorithm how array grammars with arbitrary rules can filter out the terminal arrays which are k-connected.

Lemma 2 (filtering out all k-connected arrays). *Let $k \in \mathbb{N}$. For every array grammar of type $C(G)$-$ARBA$*

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

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

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

such that $L(G'_A)$ contains exactly those arrays from $L(G_A)$ that are k-connected.

Proof (sketch). According to Lemma 1, without loss of generality we may assume that G_A is in the marked normal form. First, we replace every terminal symbol $a \in T$ by a corresponding non-terminal symbol X_a in all the array productions

with the only difference that instead of the terminal symbols $a \in T$ we have the corresponding non-terminal symbols X_a in all arrays occurring in any derivation.

Finally, instead of applying the final rule $\bar{\#} \rightarrow \#$ we move the bar to a symbol X_a and apply the rule $\bar{X}_a \rightarrow a$. This terminal seed a at some position v_0 now may propagate the signal *become terminal* to all positions v in the array derived so far which allow for a k-connected path on non-blank symbols from v_0 to v by using the rules of the form $buX_c \rightarrow bc$, $b, c \in T$, and any u being an element from the underlying group reachable from e in at most k steps in $C(G)$. This condition guarantees that when no rule is applicable any more, the obtained subarray only containing terminal symbols is k-connected, hence, if no non-terminal symbol has remained, the final array is terminal *and* k-connected.

As a technical detail we mention that to obtain the empty array we immediately apply the array production $\bar{\#} \rightarrow \#$. □

Remark 3. The idea of first working with non-terminal symbols X_a instead of terminal symbols a and then turning them into the corresponding terminal symbols a can be taken over for any arbitrary array grammar. Hence, without loss of generality, 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 $\{(v, \mathcal{A}_1(v)) \mid v \in W\} \rightarrow \{(v, \mathcal{A}_2(v)) \mid v \in W\}$ we find at least one $v_1 \in W$ such that $\mathcal{A}_1(v_1) \in N$.

Therefore, throughout the rest of the paper, when using the notion of an array grammar of type $C(G)$-$ARBA$ we will assume this condition to hold.

4 Standard Control Mechanisms

In this section we recall the notions and basic results for the general model of sequential grammars equipped with specific control mechanisms as elaborated in [21], based on the applicability of rules, as well as for the new concept of activation and blocking of rules as exhibited in [3,4].

Although in this paper we are only dealing with array grammars (over a Cayley graph $C(G)$), the control mechanisms will be defined for arbitrary sequential grammars; hence, we first recall the notion of a general model for sequential grammars, and then also the control mechanisms are introduced for this general model.

4.1 A General Model for Sequential Grammars

We first recall the main definitions of the general model for sequential grammars as established in [21], 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*;
- $w \in O$ 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 $G'_s = (O, O_T, w, P \cup P^{(+)}, \Longrightarrow_{G'_s})$ of type X such that $\Longrightarrow_{G_s} \subseteq \Longrightarrow_{G'_s}$ and

- $P^{(+)} = \{p^{(+)} \mid p \in P\}$,
- 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 $G'_s = (O, O_T, w, P \cup P^{(-)}, \Longrightarrow_{G'_s})$ of type X such that $\Longrightarrow_{G_s} \subseteq \Longrightarrow_{G'_s}$ and

- $P^{(-)} = \{p^{(-)} \mid p \in P\}$,
- 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.

In the following, we shall deal with array grammars of type $C(G)$-$ARBA$ and $C(G)$-#-CFA. In the general framework of sequential grammars as defined above, an array grammar over $C(G)$ of type $ARBA$ or #-CFA originally defined as

should be written as

$$G_A = \left((N \cup T)^{C(G)}, T^{C(G)}, \mathcal{A}_0, P, \Longrightarrow_{G_A} \right)$$

which should be kept in mind for the definitions of the control mechanisms given below.

For applying the general results on the relation 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 array grammars over a Cayley grid $C(G)$ are strictly extended types with unit rules and trap rules.*

Proof. According to Remark 3, $C(G)$-$ARBA$ can be seen as 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. □

Remark 4. The constructions given in the preceding proof verbatim hold true for the type $C(G)$-CFA, as the additional restriction that the non-terminal symbol in the array on the left-hand side of the array production must not be replaced by the blank symbol # does not affect the validity of the construction, hence, $C(G)$-CFA is a strictly extended type with unit rules and trap rules, too.

On the other hand, $C(G)$-$SCFA$ only is a strictly extended type with trap rules, as for a rule $p = (W, \mathcal{A}_1, \mathcal{A}_2)$ with $|W| > 1$ no unit rule p^+ not changing the underlying array can be found, as this would violate the condition that all positions $\neq e$ in W have to be occupied by non-blank symbols in $\mathcal{A}_2$.

4.2 Graph-Controlled and Programmed Grammars

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

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*.

4.3 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

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

$L(G_M) = \{v \in O_T \mid w \Longrightarrow^*_{G_M} v\}$ 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 of the final matrix may stop at any moment.

4.4 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 (q, R, Q) where $q \in P$, $R \cup Q \subseteq P$;
- $\Longrightarrow_{G_{RC}}$ 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 $(q, R, Q) \in P'$, $x \Longrightarrow_q y$ 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 (q, 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 (q, R, Q) in P' we have $R = \emptyset$, i.e., we only check for the non-applicability of the rules in Q.

$L(G_{RC}) = \{v \in O_T \mid w \Longrightarrow^*_{G_{RC}} v\}$ 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 $\mathcal{L}(X\text{-}fC)$, respectively.

4.5 Ordered Grammars

An *ordered grammar* G_O *of type* X is a construct

$$(G_s, \prec, \Longrightarrow_{G_O})$$

where

- $G_s = (O, O_T, w, P, \Longrightarrow_G)$ is a grammar of type X;
- $\prec$ is a partial order relation on the rules in P;
- $\Longrightarrow_{G_O}$ is the derivation relation assigned to G_O such that for any $x, y \in O$, $x \Longrightarrow_{G_O} y$ 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.

$L(G_O) = \{v \in O_T \mid w \Longrightarrow^*_{G_O} v\}$ is the language generated by G_O. The fam-

4.6 General Results

We now recall the main results and proofs established for the control mechanisms introduced so far in [21].

Theorem 1. *For any arbitrary type* X,

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

Proof (see [21]*).* Let $G_M = (G_s, M, F, \Longrightarrow_{G_M})$ be a matrix grammar with $G_s = (O, O_T, w, P, \Longrightarrow_{G_s})$ being a grammar of type X; let

$$M = \{(p_{i,1}, \ldots, p_{i,n_i}) \mid 1 \le i \le n\}$$

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

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

as well as $H_i = \{(i,1) \mid 1 \le i \le n\}$. 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 $p_{i,j} \in F$; hence, G_{GC} is without applicability checking if and only if G_M is without applicability checking, which observation completes the proof. □

The following theorem shows that forbidden contexts can simulate a partial order relation on the rules:

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

Proof (see [21]*).* Let $G_s = (O, O_T, w, P, \Longrightarrow_G)$ be a grammar of type X. Consider the ordered grammar $G_O = (G_s, \prec, \Longrightarrow_{G_O})$ of type X and the corresponding grammar with forbidden contexts $G_{fC} = (G_s, P_{fC}, \Longrightarrow_{G_{fC}})$ of type X where

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

It is easy to see that $L(G_{fC}) = L(G_O)$, because a rule $p \in P$ can be applied in G_{fC} if and only if no rule from $Q(p)$ is applicable which is the same condition

Yet also the reverse inclusion holds, provided the type X allows for trap rules:

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

Proof. Let $G_s = (O, O_T, w, P, \Longrightarrow_G)$ be a grammar of type X and consider the grammar with forbidden contexts $G_{fC} = (G_s, P_{fC}, \Longrightarrow_{G_{fC}})$ of type X with

$$P_{fC} = \{(p, \emptyset, Q(p)) \mid p \in P\}.$$

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,

- $P^{(-)} = \{p^{(-)} \mid p \in P\}$,
- 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 $L(G_{fC}) = L(G_O)$, 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}. On the other hand, the application of a rule in $P^{(-)}$ can never lead to a terminal result. □

The following result is an immediate consequence of the two previous theorems:

Corollary 1. *For any type X with trap rules, $\mathcal{L}(X\text{-}fC) = \mathcal{L}(X\text{-}O)$.*

As all the types defined for array grammars on Cayley grids in this paper are at least types with trap rules as argued above, see Lemma 3 and Remark 4, we obtain:

Corollary 2. *$\mathcal{L}(X\text{-}fC) = \mathcal{L}(X\text{-}O)$ for all types $C(G)\text{-}Y$ with $Y \in \{ARBA, \#\text{-}CFA, CFA, SCFA\}$.*

Matrix grammars (with applicability checking) can simulate random context

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 $(p, R, Q) \in P_{RC}$, $R = \{r_i \mid 1 \le i \le m\}$, $Q = \{q_j \mid 1 \le j \le n\}$, $m, n \ge 0$, we take the matrix $\left(r_1^{(+)}, \ldots, r_m^{(+)}, q_1^{(-)}, \ldots, q_n^{(-)}, p\right)$ into M. In that way we obtain $G'_s = (O, O_T, w, P', \Longrightarrow_{G'_s})$ 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 $x \Longrightarrow_{\left(r_i^{(+)}, \ldots, r_m^{(+)}, q_1^{(-)}, \ldots, q_n^{(-)}, p\right)} y$, 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 3. *For any arbitrary type X with unit rules,*

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

The main results elaborated for the relations between the specific regulating mechanisms in [21] and in this paper are depicted in the following diagram; most of these relations even hold for arbitrary types X.

Theorem 5. *The inclusions indicated by vectors as depicted in Fig. 1 hold, the additionally needed features of having unit and/or trap rules indicated by u and t, respectively, aside the vector:*

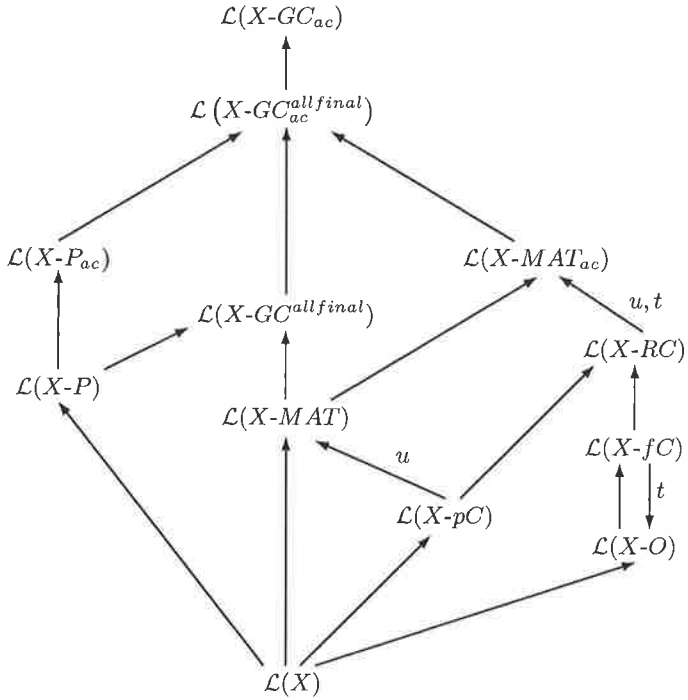


Fig. 1. Hierarchy of control mechanisms for grammars of type X.

5 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 [3–5].

A *grammar with activation and blocking of rules* (an *AB-grammar* for short) 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 $(q, Q, \bar{Q})$, $q \in L$, 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 $(q, Q, \bar{Q})$ 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

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 $(q, Q, \bar{Q})$ can be interpreted as $(\{(q,1)\} \cup Q, \bar{Q})$, and we may also simply write $(Q', \bar{Q})$ 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 $(Q', \bar{Q})$ as $(Q' \setminus \bar{Q}, \bar{Q})$.

Given a configuration $(u, Q, \bar{Q})$, in one step we can derive $(v, R, \bar{R})$, and we also write

$$(u, Q, \bar{Q}) \Longrightarrow_{G_{AB}} (v, R, \bar{R}),$$

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

$$\bar{R} = \{(x,i) \mid (x,i+1) \in \bar{Q},\ i > 0\} \cup \{(x,i) \mid (q,x,i) \in B\},$$
$$R = (\{(x,i) \mid (x,i+1) \in Q,\ i > 0\} \cup \{(x,i) \mid (q,x,i) \in A\}) \setminus \{(x,i) \mid (x,i) \in \bar{R}\}$$

  (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 $(v, R, \bar{R})$ constructed as before provided R is not empty, i.e., there are rules activated in some further derivation steps; otherwise the derivation stops.

The language generated by G_{AB} is defined by

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

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 $\mathcal{L}(X\text{-}A)$.

5.1 AB-Grammars and Graph-Controlled Grammars

Already in [21] 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.

Theorem 6. *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 $(q, Q, \bar{Q})$ or $(\bar{q}, Q, \bar{Q})$ 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 $(q, Q, \bar{Q})$ 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 $(q, Q, \bar{Q})$ 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 $(\bar{q}, Q, \bar{Q})$ 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 $(q, Q, \bar{Q})$, 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

$$\bar{R} = \{(x, i) \mid (x, i+1) \in \bar{Q},\ i > 0\} \cup \{(x, i) \mid (q, x, i) \in B\},$$
$$R = (\{(x, i) \mid (x, i+1) \in Q,\ i > 0\} \cup \{(x, i) \mid (q, x, i) \in A\}) \setminus \{(x, i) \mid (x, i) \in \bar{R}\}$$

(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 $(p, P, \bar{P})$ where $p \in \{x \mid (x, t_0) \in R\}$ and

$$\bar{P} = \{(x, i) \mid (x, i + t_0 - 1) \in \bar{R},\ i > 0\},$$
$$P = \{(x, i) \mid (x, i + t_0 - 1) \in R\}.$$

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 $(\bar{q}, Q, \bar{Q})$ 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 $(p, P, \bar{P})$ 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 $(\bar{q}, Q, \bar{Q})$ and directly go to the nodes $(p, P, \bar{P})$ 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 $(p, P, \bar{P})$ with $p \in \{x \mid (x, t_0) \in Q\}$ and

$$\bar{P} = \{(x, i) \mid (x, i + t_0 - 1) \in \bar{Q},\ i > 0\},$$
$$P = \{(x, i) \mid (x, i + t_0 - 1) \in Q\}.$$

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 an exciting result exhibiting that the power of rule activation is really strong and that the additional power of blocking is not needed.

Theorem 7. *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

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., $G'_s = \left(O, O_T, w, P', \Longrightarrow_{G'_s}\right)$ with $P' = P \cup \{p^- \mid p \in P\}$. 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 extendable 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

$((l,i), h_1, \ldots, (k,j))$ 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_C} 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 4. *For any strictly extended type X with trap rules,*

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

Combining Theorems 6 and 7, we infer the following equality:

Corollary 5. *For any strictly extended type X with trap rules,*

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

6 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 the examples in Subsect. 2.2 and Example 10 as well as Remark 2. Yet even if there is no such element (for examples of such group presentations we refer to [23]), 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 König's infinity lemma such a path must exist; it is important to observe that these paths need not be computable in the general case. Therefore, in the general case of Cayley grids we need an algorithm that works directly with the power inherent to arbitrary array productions.

Theorem 8. *For any control mechanism* Y,

$$Y \in \{O, fC, pC, RC, P, P_{ac}, MAT, MAT_{ac}, GC, GC_{ac}, A, AB\},$$

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

Proof (sketch). Given any array grammar with the control mechanism Y and with the underlying sequential array grammar being of type $ARBA$, we can construct an equivalent sequential array grammar of type $ARBA$ as follows:

The simulation of the application of an array production is obvious. The main difficulty which usually arises is to check that a specific array production is not applicable at any position in the array derived so far. In order to accomplish this task, from the beginning of the derivation we mark all positions ever visited by an array production with non-blank symbols which store the parent-children relation, i.e., as a child the information from where the underlying position has been affected is stored, and as a parent the information which children have been"born" is stored. Then, whenever we want to check that a specific array production is not applicable at any position in the Cayley grid, we send out a *checking signal* which propagates from the start node along the parent-children relations; whenever a node in the Cayley graph has no children any more and the array production under question is not applicable from that node, a *No*-signal is back-propagated along the children-parent relations, i.e., when all children have answered *No* and the rule under consideration is also not applicable at this current position in the Cayley graph, a signal *No* can be sent back from this parent node to its own parent. This algorithm ends in a successful way if the start node has received all *No*-answers from its children and the rule under consideration is also not applicable from the start node. Such information then can be moved along in the Cayley grid to a node where an array production is to be applied under the condition that specific other productions are not applicable in the whole current array.

This idea not only works for forbidding rules or for array grammars with a

to another one in the control graph of a graph-controlled grammar along an N-edge as well as for checking that no activated rule is applicable.

At the end of the simulation, the intermediate non-terminal symbols have to be erased or to be changed into their corresponding terminal symbols including the blank symbol; we here also refer to the algorithm described in the proof of Lemma 2. □

Already an order relation on the rules is sufficient as a control mechanism to obtain $\mathcal{L}(C(G)\text{-}ARBA)$:

Theorem 9. $\mathcal{L}(C(G)\text{-}ARBA) \subseteq \mathcal{L}(C(G)\text{-}\#\text{-}CFA\text{-}O)$.

Proof (sketch). Let the array language be given by an array grammar

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

on $C(G)$ in marked normal form, see Lemma 1. The underlying finitely presented group is $G = \langle B \mid R \rangle$, G' denotes the set of group elements.

We now sketch how to construct an equivalent array grammar

$$G_O = (G'_s, \prec, \Longrightarrow_{G_O}).$$

simulating the derivations in G_A.

The main idea is to first generate a workspace of non-terminal symbols $X_\#$ representing the blank symbol; such symbols $X_\#$ 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 needed for the simulations of the application of rules in G_A. In the general case, another construction is needed for that than the one exhibited in [16] for 1- and 2-dimensional array grammars. 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 $\{(v_0, S)\}$ we use a new start symbol S' and the new initial array $\{(v_0, S')\}$. Using one of the rules $S'v\# \to S''H_A$ and then the rules $H_Av\# \to \#H_A$ for any $v \in B$, the initial control symbol H_A can move to any position (node) in the Cayley graph. At some moment we use the rule $H_A \to H_0$, which ends this travel and then allows the rule $S' \to \tilde{S}$ to be applied; this rule 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 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

specific variant H_v of H, ensuring the absence of all other variants of H, using the rule relations $p \prec q$ for all $q \in \{X \to F \mid X \in V_H \setminus \{H_v\}\}$; hence, we also write $P(p \prec) = \{X \to F \mid X \in V_H \setminus \{H_v\}\}$.

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

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

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

2. we continue with marking exactly one symbol B_p as B_p' by

$$p_1 = B_p \to B_p',$$
$$P(p_1 \prec) = \{X \to F \mid X \in (V_H \setminus \{H_p^1\}) \cup \{B_p'\}\},$$

$H_p^1 \to H_p^2$, $P(H_p^1 \to H_p^2 \prec) = P_F$,
$P_F = \{\{(e, X), (v, \#)\} \to FF \mid X \in N \cup \{X_\#\}\}$,
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 guess whether we can switch to the terminal procedure eliminating all non-terminal symbols:

1. We start with $H \to H_t$; the only symbols allowed in the current array in order to obtain a terminal array are terminal symbols, the workspace symbols $X_\#$ and $\tilde{X}_\#$ as well as (one) symbol $\bar{X}_\#$ indicating that in the simulated array grammar G_A in marked normal form the final rule $\bar{X}_\# \to \#$ could be applied; hence, we take

$$P(H \to H_t \prec) = \{X \to F \mid X \in (V \setminus (T \cup \{X_\#, \tilde{X}_\#, \bar{X}_\#\}))\};$$

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. if all other non-terminal symbols have been erased, finally the control symbol H_t can be erased, too, using the rule $H \to \#$, with

$$P(H \to \# \prec) = \{X \to F \mid X \in (V \setminus \{H_t\})\};$$

According to the construction of G_O and the explanation given above we conclude that $L(G_O) = L(G_A)$. □

Looking at the general results collected in Theorem 5 we see that a partial order on the rules is the weakest control mechanism in the inclusion line of the control mechanisms

$$O - fC - RC - MAT_{ac} - GC_{ac}^{allfinal} - GC_{ac}$$

and therefore we immediately infer the following result:

Corollary 6. *For any control mechanism Y,*

$$Y \in \{O, fC, RC, MAT_{ac}, GC_{ac}^{allfinal}, GC_{ac}\},$$

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

A similar result can be shown for programmed array grammars by proving the following equality:

Lemma 4.

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

Proof. It is sufficient to show

$$\mathcal{L}(C(G)\text{-}\#\text{-}CFA\text{-}PC_{ac}) \supseteq \mathcal{L}\left(C(G)\text{-}\#\text{-}CFA\text{-}GC_{ac}^{allfinal}\right),$$

which can be proved using standard arguments already used for proving similar results for strings in [8] and for 1- and 2-dimensional arrays in [16]:

Given a graph-controlled array grammar with all nodes being final, we take a new non-terminal symbol S' as the new start symbol, i.e., instead of starting with $\{(v_0, S)\}$ we use new initial array $\{(v_0, S')\}$, and add one additional node to the control graph, to which we assign the new array production $S' \to S$; from this new node, Y-edges lead to every initial node in the original control graph.

As the new set of initial nodes we now can take every node in the new control graph, as the only array production applicable to the new initial array $\{(v_0, S')\}$ is the new array production $S' \to S$ assigned to the new node (which in fact could be the only initial node). Having all nodes being initial and final ones, the constructed new graph-controlled array grammar is a programmed one, too. □

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

Theorem 10. *For any control mechanism Y,*

$$Y \in \{O, fC, RC, P_{ac}, MAT_{ac}, GC_{ac}^{allfinal}, GC_{ac}, A, AB\},$$

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

Proof. For $Y \in \{O, fC, RC, MAT_{ac}, GC_{ac}^{allfinal}, GC_{ac}\}$, the result follows from Corollary 6 and Theorem 8.

For $Y = P_{ac}$, we apply the result stated in Lemma 4, i.e.,

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

For $Y \in \{A, AB\}$, we can use the general result stated in Corollary 5, i.e.,

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

Moreover, even using activation *and blocking* of rules does not add additional computational power beyond $\mathcal{L}(C(G)\text{-}ARBA)$, as has been shown in Theorem 8. □

Based on Lemma 2, we obtain similar results for languages of k-connected arrays; the corresponding families of languages of k-connected arrays are marked with subscript $_k$, i.e., we write $\mathcal{L}_k$ instead of $\mathcal{L}$:

Theorem 11. *For any control mechanism Y,*
$Y \in \{O, fC, RC, P_{ac}, MAT_{ac}, GC_{ac}, A, AB\}$,

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

7 Summary and Future Research

The notion of arrays as well as the concept of array grammars can be extended from the d-dimensional grid $\mathbb{Z}^d$ to arrays defined on Cayley graphs of finitely presented groups. We have investigated arrays defined on Cayley graphs of finitely presented groups and shown that the families of languages of such arrays generated by arbitrary array grammars coincide with those generated by #-context-free array grammars equipped with one out of various control mechanisms – control graphs, matrices, permitting and forbidden rules, or activation and blocking of rules. These results only need a few direct proof constructions, yet most of them directly follow from general results obtained for the relation between these control mechanisms for sequential grammars of arbitrary type.

Besides #-context-free array productions there are other types of rules to be considered in this framework of arrays defined on Cayley graphs of finitely presented groups together with these control mechanisms. For example, we are going to investigate whether similar results can be obtained when using insertion and deletion rules on arrays, for example, see [14,20]. Theorem 8 still remains valid when using array insertion and deletion rules together with the control mechanisms considered in this paper, yet showing that array insertion and deletion rules together with different control mechanisms reach the computational power of arbitrary array grammars needs careful proofs again.

There are also other control mechanisms to be considered, for example, using the structural power of tissue P systems, i.e., in a network of cells different rules

Another interesting topic is to consider accepting array grammars with control mechanisms, an investigation already having been started two decades ago, see [10]: a given input array is accepted if it can be reduced to the initial array (in the accepting case better called the *goal* array). The type of an accepting array production $(W, \mathcal{A}_2, \mathcal{A}_1)$ is defined as the type of the corresponding generating array production $(W, \mathcal{A}_1, \mathcal{A}_2)$. In [10] specific results for d-dimensional accepting array grammars together with the control mechanisms of having an order relation on the rules or control graphs were established.

Acknowledgements. The current paper has been inspired by the discussions and the exchange of ideas with many of my friends and co-authors, yet I can only mention some of them being most influential for my work: I already became interested in the concepts of regulated rewriting during my studies even a decade before the book of Jürgen Dassow and Gheorghe Păun *Regulated Rewriting in Formal Language Theory* appeared in 1989, see [8], and collected the main results on control mechanisms for the string case in a comprehensive way.

Yet since the beginning, I have always tried to also apply these control mechanisms to grammars dealing with other objects as graphs, for instance, see [19], or d-dimensional arrays, for example, see [24], the first paper with Gheorghe, as well as [16]. Already more than twenty-five years ago, together with Jürgen I started the investigation of a general framework, yet it took a long time before the initial ideas condensed in *A General Framework for Regulated Rewriting Based on the Applicability of Rules*, see [21].

Array grammars have also been an interesting topic in picture generation and acceptance, and the first paper together with Jürgen and Gheorghe on *Cooperating Array Grammar Systems* was exploring some possibilities of how to use controlled array grammars for picture generation. Cooperating systems (of grammars) are another interesting way of controlling the application of rules by allowing different sets of rules to be applied according to a given strategy as, for example, to work as long as possible or a certain number of steps on the underlying object (an array in this case), also see [17].

Variants of array grammars were investigated to be applied for character recognition, for example, see [9], which paper started a very fruitful cooperation with Henning Fernau. The topic of character recognition was continued together with Markus Holzer, see [11–13]. Recently the cooperation with Henning (and his team in Trier) has been resumed with several papers dealing with control mechanisms for several variants of array grammars, for example, see [2,14,15].

With the concept of membrane systems introduced by Gheorghe Păun, see [27,28], the Handbook of Membrane Computing, a new model emerged, controlling the applicability of rules by using the hierarchical membrane structure where together with the application of a rule the current array can be sent to the outer or an inner membrane, which makes other sets of rules applicable, for example, see [18], the first paper on that topic for the sequential derivation mode with arrays as underlying objects, as well as [14,20], the contributions to MCU 2013.

From the beginning of this century, Marion Oswald, my colleague and friend working with me at the TU Wien for a very long period, had become my favorite co-author, especially in the area of P systems, but she also co-authored several papers on regulated rewriting, especially [21] on *A General Framework for Regulated Rewriting*

Based on the Applicability of Rules, and on variants of array grammars, especially the main papers introducing *Array Grammars and Automata on Cayley Grids*, see [22,23].

More recently, my friends from Moldova Artiom Alhazov and Sergiu Ivanov have become the motivating force to continue the research on new control mechanisms, not only in the area of P systems, but also with respect to new control mechanisms, especially also for array grammars, for instance, see [2,14,20]. The new concept of activation and blocking of rules in its final form has been developed during the *Brainstorming Week on Membrane Computing* this year in Sevilla mainly together with Sergiu, and different facets of this new control mechanism are presented in several papers, see [3–5], including this paper.

Both the list of references and the list of colleagues who contributed to my research on control mechanisms and/or on array grammars are far from being complete. I want to express my deep respect and my gratitude to all my colleagues and friends who helped me to develop new ideas and concepts, some of them presented in this paper.

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, River Edge (1989)
2. Alhazov, A., Fernau, H., Freund, R., Ivanov, S., Siromoney, R., Subramanian, K.G.: Contextual array grammars with matrix control, regular control languages, and tissue P systems control. Theor. Comput. Sci. **682**, 5–21 (2017). https://doi.org/10.1016/j.tcs.2017.03.012
3. Alhazov, A., Freund, R., Ivanov, S.: Introducing the concept of activation and blocking of rules in the general framework for regulated rewriting in sequential grammars. In: Proceedings of BWMC 2018 (2018)
4. Alhazov, A., Freund, R., Ivanov, S.: P systems with activation and blocking of rules. In: Verlan, S. (ed.) Proceedings of UCNC 2018. Lecture Notes in Computer Science. Springer (2018)
5. Alhazov, A., Freund, R., Ivanov, S.: Sequential grammars with activation and blocking of rules. In: Durand-Lose, J., Verlan, S. (eds.) MCU 2018. LNCS, vol. 10881, pp. 51–68. Springer, Cham (2018)
6. Cook, C.R., Wang, P.S.P.: A Chomsky hierarchy of isotonic array grammars and languages. Comput. Graph. Image Process. **8**, 144–152 (1978)
7. Csuhaj-Varjú, E., Mitrana, V.: Array grammars on Cayley grids, private communication
8. Dassow, J., Păun, Gh.: Regulated Rewriting in Formal Language Theory. EATCS Monographs in Theoretical Computer Science, vol. 18. Springer, Heidelberg (1989)
9. Fernau, H., Freund, R.: Bounded parallelism in array grammars used for character recognition. In: Perner, P., Wang, P., Rosenfeld, A. (eds.) SSPR 1996. LNCS, vol. 1121, pp. 40–49. Springer, Heidelberg (1996). https://doi.org/10.1007/3-540-61577-6_5
10. Fernau, H., Freund, R.: Accepting array grammars with control mechanisms. In: Păun, Gh., Salomaa, A. (eds.) New Trends in Formal Languages. LNCS, vol. 1218, pp. 95–118. Springer, Heidelberg (1997). https://doi.org/10.1007/3-540-62844-4_7
11. Fernau, H., Freund, R., Holzer, M.: Character recognition with k-head finite array automata. In: Amin, A., Dori, D., Pudil, P., Freeman, H. (eds.) SSPR /SPR 1998. LNCS, vol. 1451, pp. 282–291. Springer, Heidelberg (1998). https://doi.org/10.

12. Fernau, H., Freund, R., Holzer, M.: Regulated array grammars of finite index, Part I: theoretical investigations. In: Păun, Gh., Salomaa, A. (eds.) Grammatical Models of Multi-Agent Systems. Topics in Computer Mathematics, vol. 8, pp. 157–181. Gordon and Breach Science Publishers, London (1999)
13. Fernau, H., Freund, R., Holzer, M.: Regulated array grammars of finite index. Part II: syntactic pattern recognition. In: Păun, Gh., Salomaa, A. (eds.) Grammatical Models of Multi-Agent Systems. Topics in Computer Mathematics, vol. 8, pp. 284–296. Gordon and Breach Science Publishers, London (1999)
14. Fernau, H., Freund, R., Ivanov, S., Schmid, M.L., Subramanian, K.G.: Array insertion and deletion P systems. In: Mauri, G., Dennunzio, A., Manzoni, L., Porreca, A.E. (eds.) UCNC 2013. LNCS, vol. 7956, pp. 67–78. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-39074-6_8
15. Fernau, H., Freund, R., Siromoney, R., Subramanian, K.G.: Non-isometric contextual array grammars and the role of regular control and local selectors. Fundam. Inform. **155**(1–2), 209–232 (2017). https://doi.org/10.3233/FI-2017-1582
16. 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)
17. Freund, R.: Array grammar systems. J. Autom. Lang. Comb. **5**(1), 13–30 (2000)
18. Freund, R.: P systems working in the sequential mode on arrays and strings. In: Calude, C.S., Calude, E., Dinneen, M.J. (eds.) DLT 2004. LNCS, vol. 3340, pp. 188–199. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-30550-7_16
19. Freund, R., Haberstroh, B.: Attributed elementary programmed graph grammars. In: Schmidt, G., Berghammer, R. (eds.) WG 1991. LNCS, vol. 570, pp. 75–84. Springer, Heidelberg (1992). https://doi.org/10.1007/3-540-55121-2_7
20. 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, 9–11 September 2013. EPTCS, vol. 128, pp. 62–75 (2013). https://doi.org/10.4204/EPTCS.128
21. 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. LNCS, vol. 6610, pp. 35–53. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-20000-7_5
22. Freund, R., Oswald, M.: Array automata on Cayley grids. In: Neary, T., Cook, M. (eds.) Proceedings Machines, Computations and Universality 2013 Zürich, Switzerland, 9–11 September 2013. EPTCS, vol. 128, pp. 27–28 (2013). https://doi.org/10.4204/EPTCS.128
23. Freund, R., Oswald, M.: Array grammars and automata on Cayley grids. J. Autom. Lang. Comb. **19**(1–4), 67–80 (2014). https://doi.org/10.25596/jalc-2014-067
24. Freund, R., Păun, Gh.: One-dimensional matrix array grammars. Elektronische Informationsverarbeitung und Kybernetik **29**(6), 357–374 (1993)
25. Holt, D.F., Eick, B., O'Brien, E.A.: Handbook of Computational Group Theory. CRC Press, Hoboken (2005)
26. Krithivasan, K., Siromoney, R.: Array automata and operations on array languages. Int. J. Comput. Math. **4**(1), 3–30 (1974). https://doi.org/10.1080/00207167408803078
27. Păun, Gh.: Computing with Membranes. J. Comput. Syst. Sci. **61**, 108–143 (1998)
28. Păun, Gh., Rozenberg, G., Salomaa, A.: The Oxford Handbook of Membrane Com-

29. Rosenfeld, A.: Picture Languages. Academic Press, Reading (1979)
30. Rosenfeld, A., Siromoney, R.: Picture languages - a survey. Lang. Des. **1**, 229–245 (1993)
31. Rozenberg, G., Salomaa, A. (eds.): Handbook of Formal Languages: Volume 3 Beyond Words. Springer, Heidelberg (1997). https://doi.org/10.1007/978-3-642-59126-6
32. Salomaa, A.: Formal Languages. Academic Press, New York (1973)
33. Wang, P.S.P.: An application of array grammars to clustering analysis for syntactic patterns. Pattern Recogn. **17**, 441–451 (1984)