Introducing the Concept of Activation and Blocking of Rules in the General Framework for Regulated Rewriting in Sequential Grammars*

Artiom Alhazov[1], Rudolf Freund[2], and Sergiu Ivanov[3]

[1] Institute of Mathematics and Computer Science
Academiei 5, Chişinău, MD-2028, Moldova
artiom@math.md

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

[3] IBISC, Université Évry, Université Paris-Saclay
23 Boulevard de France, 91025, Évry, France
sergiu.ivanov@univ-evry.fr

Summary. We introduce new possibilities to control the application of rules based on the preceding application of rules which can be defined for a general model of sequential grammars and we show some similarities to other control mechanisms as graph-controlled grammars and matrix grammars with and without applicability checking as well as grammars with random context conditions and ordered grammars. Using both activation and blocking of rules, in the string and in the multiset case we can show computational completeness of context-free grammars equipped with the control mechanism of activation and blocking of rules even when using only two nonterminal symbols.

1 Introduction

Nearly thirty years ago, the monograph on regulated rewriting by Jürgen Dassow and Gheorghe Păun [2] already gave a first comprehensive overview on many concepts of regulated rewriting, especially for the string case. 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 [4]. Even in the emerging field of P systems [10, 14] where mostly multisets are considered, such regulating mechanisms were used [1]. As exhibited in [6], for comparing the generating power of grammars working in the sequential derivation

* The work is supported by National Natural Science Foundation of China (61320106005, 61033003, and 61772214) and the Innovation Scientists and Technicians Troop Construction Projects of Henan Province (154200510012).

mode, many relations between various regulating mechanisms can be established in a very general setting without any reference to the underlying objects the rules are working on, using a general model for graph-controlled, programmed, random-context, and ordered grammars of arbitrary type based on the applicability of rules.

In the second section, we recall some notions from formal language theory as well as the main definitions of the general framework for sequential grammars elaborated in [6]. Then we define the new concept of activation and blocking of rules based on the applicability of rules within this general framework for regulated rewriting. In Section 3 some general results for sequential grammars using the control mechanism of activation or activation and blocking of rules are established. Specific results on computational completeness for strings, multisets, and arrays as underlying objects then are shown in Section 4. In Section 5 we establish our main results for strings and multisets showing that context-free (string and multiset) grammars with activation and blocking of rules are computationally complete even when only two non-terminal symbols are used, which establishes a sharp border as one non-terminal symbol is not sufficient. Finally, a summary of the results obtained in this paper and some future research topics extending the notions and results obtained in this paper are given in Section 6.

2 Definitions

After some preliminaries from formal language theory, we define our general model for grammars and recall some notions for string, array, and multiset grammars and languages in the general setting of this paper. Then we formulate the models of graph-controlled, programmed, matrix grammars with and without appearance checking, as well as random-context grammars, based on the applicability of rules.

2.1 Preliminaries

The set of integers is denoted by $\mathbb{Z}$, the set of non-negative integers by $\mathbb{N}_0$, and the set of positive integers (natural numbers) by $\mathbb{N}$. An *alphabet* V is a finite 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^+. 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 more details of formal language theory the reader is referred to the monographs and handbooks in this area [2, 12].

2.2 A General Model for Sequential Grammars

We first recall the main definitions of the general model for sequential grammars as established in [6], grammars generating a set of terminal objects by derivations where in each derivation step exactly one rule is applied to exactly one object. This does not cover rules involving more than one object – as, for example, splicing rules – or other derivation modes – as, for example, the maximally parallel mode considered in many variants of P systems [10].

A *(sequential) grammar* G is a construct $(O, O_T, w, P, \Longrightarrow_G)$ 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*;
- $\Longrightarrow_G \subseteq O \times O$ is the *derivation relation* of G.
  We assume that each of the rules $p \in P$ induces a relation $\Longrightarrow_p \subseteq O \times O$ with respect to $\Longrightarrow_G$ fulfilling at least the following conditions: (i) for each object $x \in O$, $(x, y) \in \Longrightarrow_p$ for only finitely many objects $y \in O$; (ii) there exists a finitely described mechanism as, for example, a Turing machine, which, given an object $x \in O$, computes all objects $y \in O$ such that $(x, y) \in \Longrightarrow_p$. 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 $\Longrightarrow_G$ is the union of all $\Longrightarrow_p$, i.e., $\Longrightarrow_G = \cup_{p \in P} \Longrightarrow_p$. The reflexive and transitive closure of $\Longrightarrow_G$ is denoted by $\overset{*}{\Longrightarrow}_G$.

In the following we shall consider different types of grammars depending on the components of G (where the set of objects O is infinite, e.g., V^*, the set of strings over the alphabet V), especially with respect to different types of rules (e.g., context-free string rules). Some specific conditions on the elements of G, especially on the rules in P, may 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 (we also assume $v \in O_T$ to be decidable for every $v \in O$) derivable from the axiom, i.e.,

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

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

Let $G = (O, O_T, w, P, \Longrightarrow_G)$ be a grammar of type X. If for every G 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 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 $G = (O, O_T, w, P, \Longrightarrow_G)$ of type X a grammar $G' = \left(O, O_T, w, P \cup P^{(+)}, \Longrightarrow_{G'}\right)$ of type X exists such that $\Longrightarrow_G \subseteq \Longrightarrow_{G'}$ 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 $G = (O, O_T, w, P, \Longrightarrow_G)$ of type X a grammar $G' = \left(O, O_T, w, P \cup P^{(-)}, \Longrightarrow_{G'}\right)$ of type X exists such that $\Longrightarrow_G \subseteq \Longrightarrow_{G'}$ 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 an object y from which no terminal object can be derived anymore.

2.3 Specific Types of Objects

String grammars

In the general notion as defined above, a *string grammar* G_S is represented as

$$\left((N \cup T)^*, T^*, w, P, \Longrightarrow_P\right)$$

where 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 $u \in V^*$ (for generating grammars, $u \in V^+$) and $v \in V^*$ (for accepting grammars, $v \in V^+$), with $V := N \cup T$; 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. In the following, we shall also use the common notation $G_S = (N, T, w, P)$ instead, too. We remark that, usually, the axiom w is supposed to be a non-terminal symbol, i.e., $w \in V \setminus T$, and 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 an 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 familiy 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).

We refer to [6] where some examples for string grammars of specific types illustrating the expressive power of this general framework are given.

Array grammars

We now introduce the basic notions for n-dimensional arrays and array grammars, for example, see [4, 11, 13].

Let $d \in \mathbb{N}$. Then a *d-dimensional array* $\mathcal{A}$ over an alphabet V is a function $\mathcal{A} : \mathbb{Z}^d \to V \cup \{\#\}$, where $shape(\mathcal{A}) = \left\{v \in \mathbb{Z}^d \mid \mathcal{A}(v) \neq \#\right\}$ is finite and $\# \notin V$ is called the *background* or *blank symbol*. We usually write $\mathcal{A} = \{(v, \mathcal{A}(v)) \mid v \in shape(\mathcal{A})\}$.

The set of all d-dimensional arrays over V is denoted by V^{*d}. The *empty array* in V^{*d} with empty shape is denoted by Λ_d. Moreover, we define $V^{+d} = V^{*d} \setminus \{\Lambda_d\}$.

Let $v \in \mathbb{Z}^d$, $v = (v_1, \ldots, v_d)$. The *translation* $\tau_v : \mathbb{Z}^d \to \mathbb{Z}^d$ is defined by $\tau_v(w) = w + v$ for all $w \in \mathbb{Z}^d$, and for any array $\mathcal{A} \in V^{*d}$ we define $\tau_v(\mathcal{A})$, the corresponding d-dimensional array translated by v, by $(\tau_v(\mathcal{A}))(w) = \mathcal{A}(w - v)$ for all $w \in \mathbb{Z}^d$. The vector $(0, \ldots, 0) \in \mathbb{Z}^d$ is denoted by Ω_d.

A *d-dimensional array rule* p over V is a triple $(W, \mathcal{A}_1, \mathcal{A}_2)$, where $W \subseteq \mathbb{Z}^d$ is a finite set and $\mathcal{A}_1$ and $\mathcal{A}_2$ are mappings from W to $V \cup \{\#\}$ such that $shape(\mathcal{A}_1) \neq \emptyset$. We say that the array $\mathcal{B}_2 \in V^{*d}$ is *directly derivable* from the array $\mathcal{B}_1 \in V^{*d}$ by the d-dimensional array rule $(W, \mathcal{A}_1, \mathcal{A}_2)$, i.e., $\mathcal{B}_1 \Longrightarrow_p \mathcal{B}_2$, if and only if there exists a vector $v \in \mathbb{Z}^d$ such that $\mathcal{B}_1(w) = \mathcal{B}_2(w)$ for all $w \in \mathbb{Z}^d \setminus \tau_v(W)$ as well as $\mathcal{B}_1(w) = \mathcal{A}_1(\tau_{-v}(w))$ and $\mathcal{B}_2(w) = \mathcal{A}_2(\tau_{-v}(w))$ for all $w \in \tau_v(W)$, i.e., the subarray of $\mathcal{B}_1$ corresponding to $\mathcal{A}_1$ is replaced by $\mathcal{A}_2$, thus yielding $\mathcal{B}_2$. In the following, we shall also write $\mathcal{A}_1 \to \mathcal{A}_2$, because W is implicitly given by the finite arrays $\mathcal{A}_1, \mathcal{A}_2$.

A *d-dimensional array grammar* G_A is represented as

$$\left((N \cup T)^{*d}, T^{*d}, \{(v_0, S)\}, P, \Longrightarrow_{G_A}\right) \text{ where}$$

- N is the alphabet of *non-terminal symbols*;
- T is the alphabet of *terminal symbols*, $N \cap T = \emptyset$;
- $\{(v_0, S)\}$ is the *start array (axiom)* with $S \in N$ and $v_0 \in \mathbb{Z}^d$;
- P is a finite set of *d-dimensional array rules* over V, $V := N \cup T$;
- $\Longrightarrow_{G_A}$ is the derivation relation induced by the array rules in P according to the explanations given above, i.e., for arbitrary $\mathcal{B}_1, \mathcal{B}_2 \in V^{*d}$, $\mathcal{B}_1 \Longrightarrow_{G_A} \mathcal{B}_2$ if

and only if there exists a d-dimensional array rule $p = (W, \mathcal{A}_1, \mathcal{A}_2)$ in P such that $\mathcal{B}_1 \Longrightarrow_p \mathcal{B}_2$.

A d-dimensional array rule $p = (W, \mathcal{A}_1, \mathcal{A}_2)$ in P is called *#-context-free*, if $shape(\mathcal{A}_1) = \{\Omega_d\}$. A d-dimensional array grammar is said to be of type d-$ARBA$, d-#-CFA if every array rule in P is of the corresponding type, i.e., an arbitrary and #-context-free d-dimensional array rule, respectively. The corresponding families of d-dimensional array languages of type X are denoted by $\mathcal{L}(X)$, i.e., $\mathcal{L}(d\text{-}ARBA)$ and $\mathcal{L}(d\text{-}\#\text{-}CFA)$ are the families of recursively enumerable and #-context-free d-dimensional array languages, respectively.

Observe that the types d-$ARBA$ and d-#-CFA are types with unit rules and trap rules – for $p = (W, \mathcal{A}_1, \mathcal{A}_2)$, we can take $p^{(+)} = (W, \mathcal{A}_1, \mathcal{A}_1)$ and $p^{(-)} = (W, \mathcal{A}_1, \mathcal{A}_F)$ with $\mathcal{A}_F(v) = F$ for $v \in W$, where F is a new non-terminal symbol – the trap symbol.

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 we also write $(N, T, w, P, \Longrightarrow_{G_m})$.

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 $Ps(\mathcal{L}(ARB))$ [8]:

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

2.4 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 $k + 2$ registers, for instance, see [9].

2.5 Graph-controlled and Programmed Grammars

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

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

where $G = (O, O_T, w, P, \Longrightarrow_G)$ 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 $\Longrightarrow_G$ 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 and concepts *with/without applicability checking* were introduced as *with/without appearance checking* in the original definition for string grammars 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*.

2.6 Matrix Grammars

A *matrix grammar* (with applicability checking) of type X is a construct

$$G_M = (G, M, F, \Longrightarrow_{G_M})$$

where $G = (O, O_T, w, P, \Longrightarrow_G)$ 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

- $w_i \Longrightarrow_G w_{i+1}$ 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*; 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.

2.7 Random-Context Grammars

The following general notion of a random context-grammar had already been introduced in [7, 1] in a similar way before it was formulated in [6].

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

- $G = (O, O_T, w, P, \Longrightarrow_G)$ 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 $G_{RC} = (G, P', \Longrightarrow_{G_{RC}})$ 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 $G_{RC} = (G, P', \Longrightarrow_{G_{RC}})$ 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.

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

2.8 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.

$L(G_O) = \left\{v \in O_T \mid w \Longrightarrow^*_{G_O} v\right\}$ is the language generated by G_O. The family of languages generated by ordered grammars of type X is denoted by $\mathcal{L}(X\text{-}O)$.

2.9 Grammars with Activation and Blocking of Rules

We now define our new concept of regulating the application of rules at a specific moment by activation and blocking relations.

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

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

where $G = (O, O_T, w, P, \Longrightarrow_G)$ 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_M 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 although no rule has been applied so far, but probably also providing additional activations and blockings for further derivation steps.

A configuration of G_M 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 $\left(v, R, \bar{R}\right)$, and 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

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

  (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.

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 $\mathcal{L}(X\text{-}A)$. In this case we might not allow the second case in a derivation of the A-grammar that in a derivation step no rule is activated to be applied. Moreover, an A-grammar is called an *A1-grammar* if for all $(p,q,t) \in A$ we have $t = 1$, which means that only the rule applied in one derivation step activates the rules which can be applied in the next step; in this case we may only write (p,q) instead of $(p,q,1)$. Moreover, in L_0 we may simply list the labels of the rules to be applied in the first step.

Example 1. Consider the string grammar $G_S = \left((N \cup T)^*, T^*, w, P, \Longrightarrow_P\right)$ with
$N = \{A,B,C\}$, $T = \{a,b,c\}$, $w = ABC$, and the set of rules
$P = \{A \to aA, B \to bB, C \to cC, A \to \lambda, B \to \lambda, C \to \lambda\}$,
as well as the A1-grammar
$G_A = (G, L, f_L, A, L_0, \Longrightarrow_{G_A})$ with
$L = \{p_a, p_b, p_c, p_A, p_B, p_C\}$, and, writing $p : r$ for the pairs (p,r) in f_L,

$$\begin{aligned}f_L &= \{p_a : A \to aA, p_b : B \to bB, p_c : C \to cC\} \\ &\quad \cup \{p_A : A \to \lambda, p_B : B \to \lambda, p_C : C \to \lambda\} \\ A &= \{(p_a,p_b), (p_b,p_c), (p_c,p_a), (p_c,p_A), (p_A,p_B), (p_B,p_C)\}, \text{ and} \\ P_0 &= \{p_a, p_A\}.\end{aligned}$$

The underlying string grammar generates the regular set $\{a\}^* \{b\}^* \{c\}^*$, whereas the A1-grammar G_A generates $\{a^n b^n c^n \mid n \in \mathbb{N}_0\}$: starting with the rule labeled by p_a from L_0, the rules corresponding to the sequence of labels $p_a p_b p_c$ is applied $n \geq 1$ times, and finally we switch to the sequence of rules given by $p_A p_B p_C$ whereafter no rule can be applied any more. Starting with p_A yields the empty string.

Only allowing blocking of rules would not make sense, but if we implicitly have all rules activated in every derivation step, then blocking some of the rules with the application of a rule in a derivation step for the next derivation step(s) allows us to speak of a *grammar with blocking of rules* (a *B-grammar* for short) of type X; the corresponding family of languages is denoted by $\mathcal{L}(X\text{-}B)$. Moreover, a B-grammar is called a $B1$-*grammar* if for all $(p,q,t) \in B$ we have $t = 1$, which means that the rule applied in one derivation step can only block the rules to be applied in the next step; in this case we again only write (p,q) instead of $(p,q,1)$. Moreover, in L_0 we may simply list the labels of the rules to be applied in the first step.

Example 2. We consider the same underlying string grammar as in Example 1, $G_S = \left((N \cup T)^*, T^*, w, P, \Longrightarrow_P\right)$ with $N = \{A,B,C\}$, $T = \{a,b,c\}$, $w = ABC$, and the set of rules
$P = \{A \to aA, B \to bB, C \to cC, A \to \lambda, B \to \lambda, C \to \lambda\}$. From the A1-grammar as constructed in Example 1, we construct an equvalent B1-grammar
$G_B = (G, L, f_L, B, L_0, \Longrightarrow_{G_B})$ with
$L = \{p_a, p_b, p_c, p_A, p_B, p_C\}$, and, writing $p : r$ for the pairs (p,r) in f_L,
$f_L = \{p_a : A \to aA, p_b : B \to bB, p_c : C \to cC\}$
$\quad \cup \{p_A : A \to \lambda, p_B : B \to \lambda, p_C : C \to \lambda\}$
$B = \{(p_a, L \setminus \{p_b\}), (p_b, L \setminus \{p_c\}), (p_c, L \setminus \{p_a, p_A\})\}$
$\quad \cup \{(p_A, L \setminus \{p_B\}), (p_B, L \setminus \{p_C\})\}$, and
$P_0 = \{p_a, p_A\}$.

This B1-grammar G_B generates the same language as the A1-grammar G_A constructed in Example 1, i.e., $\{a^n b^n c^n \mid n \in \mathbb{N}_0\}$: instead of activating the next rules to be applied, we block all the other rules.

3 General Results

In this section, we elaborate some general results holding true for many types of grammars, some even holding for any type X, whereas some of them rely on specific conditions on X.

3.1 General Results for Standard Control mechanisms

The main results elaborated for the relations between the specific regulating mechanisms in [6] and in [5] (not including the new mechanism of activation and blocking of rules) are depicted in Figure 1; most of these relations even hold for arbitrary types X.

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

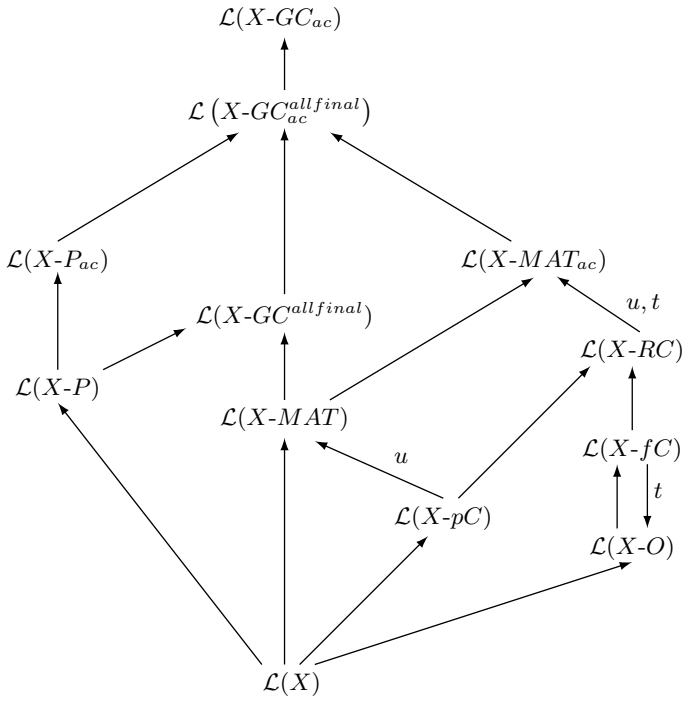

Fig. 1. Hierarchy of control mechanisms for grammars of type X.

3.2 A1-Grammars and B1-Grammars

There is an interesting relation between A1-Grammars and B1-Grammars which is quite surprising as usually forbidding rules to be applied does not yield the same computational power as prescribing the rules to be applied in the next step as, for example, in matrix grammars without *ac*. The conceptual reason behind this result is that in B-grammars, by default, all rules are activated for every derivation step.

Theorem 2. *For any type* X, $\mathcal{L}\left(X\text{-}A1\right) = \mathcal{L}\left(X\text{-}B1\right)$.

Proof. We first show $\mathcal{L}\left(X\text{-}A1\right) \subseteq \mathcal{L}\left(X\text{-}B1\right)$.

Let $G_A = (G, L, f_L, A, L_0, \Longrightarrow_{G_A})$ be an A1-grammar where the underlying grammar $G = (O, O_T, w, P, \Longrightarrow_G)$ is 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 is a finite subset of $L \times L$, and $L_0 \subseteq L$ is the set of initial rule labels.

Then we define the equivalent B1-grammar of type X G_B as follows:

$$\begin{aligned} G_B &= (G, L, f_L, B, L_0, \Longrightarrow_{G_B}), \\ B &= \{(l, L \setminus \{m \mid (l, m) \in A\}) \mid l \in L\}, \end{aligned}$$

i.e., B is constructed in such a way that instead of activating the rules to be applied in the next derivation step, we block all the other rules – observe that by default in B-grammars all rules are activated for every derivation step (compare this with the construction of the B1-grammar in Example 2 from the A1-grammar given in Example 1).

We now show the other direction, $\mathcal{L}(X\text{-}A1) \supseteq \mathcal{L}(X\text{-}B1)$.

Let $G_B = (G, L, f_L, B, L_0, \Longrightarrow_{G_B})$ be a B1-grammar where the underlying grammar $G = (O, O_T, w, P, \Longrightarrow_G)$ is of type X, L is a finite set of labels with each label having assigned one rule from P by the function f_L, B is a finite subset of $L \times L$, and $L_0 \subseteq L$ is the set of initial rule labels.

Then we define the equivalent A1-grammar of type X G_A as follows:

$$\begin{aligned} G_A &= (G, L, f_L, A, L_0, \Longrightarrow_{G_A}), \\ A &= \{(l, L \setminus \{m \mid (l, m) \in B\}) \mid l \in L\}, \end{aligned}$$

i.e., A is constructed from B in such a way that only those rules are activated to be applied in the next derivation step which are not blocked according to B. □

It remains as an open question if a similar result also holds for arbitrary A- and B-grammars.

3.3 Matrix Grammars and A1-Grammars

Our first result shows a close connection between matrix grammars without appearance checking and A1-grammars:

Theorem 3. *For any type X, $\mathcal{L}(X\text{-}MAT) \subseteq \mathcal{L}(X\text{-}A1)$.*

Proof. Let $G_M = (G, M, F, \Longrightarrow_{G_M})$ be a matrix grammar with the underlying grammar $G = (O, O_T, w, P, \Longrightarrow_G)$ being a grammar of type X; let $M = \{(p_{i,1}, \ldots, p_{i,n_i}) \mid 1 \leq i \leq n\}$ with $p_{i,j} \in P$, $1 \leq j \leq n_i$, $1 \leq i \leq n$.

We construct the equivalent A1-grammar

$$\begin{aligned} G_A &= (G, L, f_L, A, L_0, \Longrightarrow_{G_A}), \\ L &= \{l_{i,j} \mid 1 \leq j \leq n_i,\ 1 \leq i \leq n\}, \\ f_L &= \{(l_{i,j}, p_{i,j}) \mid 1 \leq j \leq n_i,\ 1 \leq i \leq n\}, \\ A &= \{(l_{i,j}, l_{i,j+1}) \mid 1 \leq j < n_i, 1 \leq i \leq n\} \\ &\quad \cup \{(l_{i,n_i}, l_{j,1}) \mid 1 \leq j \leq n, 1 \leq i \leq n\}, \\ L_0 &= \{l_{i,1} \mid 1 \leq i \leq n\}. \end{aligned}$$

We mention that according to our definitions the sequential application of the rules of the chosen matrix may stop at any moment if the next rule cannot be applied, in which case also the simulation in the A1-grammar stops. □

We immediately infer the following for the special cases of strings, multisets, and arrays as underlying objects:

Corollary 1. *For $X \in \{CF, mCF\} \cup \{d\text{-}\#\text{-}CFA \mid d \in \mathbb{N}\}$,*

$$\mathcal{L}(X\text{-}MAT) \subseteq \mathcal{L}(X\text{-}A1).$$

3.4 Random Context Grammars and AB-Grammars

For any type X with unit rules, random context grammars of type X can be simulated by AB-grammars of type X.

Remark 1. In order to keep proofs shorter, in the following, instead of specifying the set of rules P, the set of labels L, and the function f_L assigning rules to the labels separately, we will only specify the corresponding labeled rules of the form $l : r$ with $l \in L$, $r \in P$, and $(l, r) \in f_L$. Moreover, for $X \in \{A, B\}$, instead of $(p, q, t) \in X$, we write $(p, q, t)_X$.

Theorem 4. *For any type X with unit rules, $\mathcal{L}(X\text{-}RC) \subseteq \mathcal{L}(X\text{-}AB)$.*

Proof. Let $(G, R, \Longrightarrow_{G_{RC}})$ be a random context grammar with the underlying grammar $G = (O, O_T, w, P, \Longrightarrow_G)$ being of a type X with unit rules, where

$$\begin{aligned} R &= \{(r_i, P_i, Q_i) \mid 1 \le i \le n\}, r_i \in P,\ 1 \le i \le n, \\ P_i &= \{p_{i,j} \mid 1 \le j \le m_i,\ 1 \le i \le n\}, m_i \ge 0,\ 1 \le i \le n, \\ Q_i &= \{q_{i,j} \mid 1 \le j \le n_i,\ 1 \le i \le n\}, n_i \ge 0,\ 1 \le i \le n. \end{aligned}$$

Then we construct an AB-grammar G_{AB} of type X as follows:

$$\begin{aligned} G_{AB} &= (G', L, f_L, A, B, L_0, \Longrightarrow_{G_A}), \\ G' &= (O, O_T, w, P', \Longrightarrow_{G'}), \\ P' &= P \cup \{r^+ \mid r \in P\}; \\ L_0 &= \{l_{r_i} \mid 1 \le i \le n\}; \end{aligned}$$

the application of a random context rule (r_i, P_i, Q_i) is simulated by the following sequence of labeled rules together with suitable activations and blockings of rules:

- $l_{r_i} : {r_i}^+$, $(l_{r_i}, l_{r_i,1})_A$, $(l_{r_i}, \bar{l}_{r_i,j}, m_i+j)_A$, $1 \le j \le n_i$; at the beginning, the checking of all rules which should not be applicable is initiated, and the sequence of applicability checkings for the rules in P_i is started;
- $l_{r_i,j} : {p_{i,j}}^+$, $(l_{r_i,j}, l_{r_i,j+1})_A$, $1 \le j < m_i$;
- $l_{r_i,m_i} : {p_{i,m_i}}^+$, $(l_{r_i,m_i}, \hat{l}_{r_i}, n_i + 1)_A$; when all rules in P_i have been checked to be applicable, the application of rule r_i after further n_i steps is activated; yet if any of the rules in Q_i is applicable, then this application of rule r_i is blocked;
- $\bar{l}_{r_i,j} : {q_{i,j}}^+$, $(\bar{l}_{r_i,j}, \hat{l}_{r_i}, n_i - j + 1)_B$, $1 \le j \le n_i$;
- $\hat{l}_{r_i} : r_i$, $(\hat{l}_{r_i}, l_{r_k})$, $1 \le k \le n$; after the successful application of rule r we may continue with trying to apply any random context rule from R.

We finally observe that only unit rules and no trap rules as in other simulations known from [6] are needed to obtain this result. □

3.5 AB-Grammars and Graph-Controlled Grammars

Already in [6] 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 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

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

  (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

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

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. □

The power of rule activation is really strong and in most cases the additional power of blocking is not needed. 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 $\mathcal{L}\left(X\text{-}GC_{ac}^{allfinal}\right)$.

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

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

Proof. Let

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

be a graph-controlled grammar where $G = (O, O_T, w, P, \Longrightarrow_G)$ is a strictly extended grammar of type X; $g = (H, E, K)$, $E \subseteq H \times \{Y, N\} \times H$ 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

$$G_A = (G', L, f_L, A, L_0, \Longrightarrow_{G_A})$$

as follows:

The underlying grammar G' is obtained from G by adding all unit and trap rules, i.e., $G' = (O, O_T, w, P', \Longrightarrow_{G'})$ with $P' = P \cup \{p^+, p^- \mid p \in P\}$. G' 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 $P_l = \{p_{l,i} \mid 1 \leq i \leq n_l\}$. Moreover, for $p_{l,i}$ we take a label (l, i) into L and $((l, i), p_{l,i})$ 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:

For each rule $p_{l,i}$ in P_l, $1 \leq i \leq n_l$, $(l, Y, k) \in E$ and $p_{k,j} \in P_k$, we take $((l, i), (k, j))$ into A.

If no rule in P_l can be applied, a trickier construction is needed: as long as we assume that at some moment when going through the control graph a rule will be applicable, we guess in which node k this will happen as well as a path $h_0 = l - h_1 - \cdots - h_n = k$ in g following only N-edges from node l to node k which does not contain a loop. For any such path we introduce a label $(\bar{l}, h_1, \ldots, (k, j))$ in L and $(\bar{l}, h_1, \ldots, (k, j)) : {p_{k,j}}^+$ in f_L. Moreover, we use the following activations in A:

- $((\bar{l}, h_1, \ldots, (k, j)), \{q^- \mid q \in \bigcup_{0 \leq i \leq k-1} P_{h_i}, 1)$ is used to check in the next step that no rule along the path from node l to node k is applicable, whereas in the second next step only the designated rule $p_{k,j}$ can be applied, i.e., we take
- $((\bar{l}, h_1, \ldots, (k, j)), p_{k,j}, 2)$ into A.

What remains to be settled is how a derivation in the A-grammar starts:

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; for each such rule $p_{k,j}$ in node k we introduce an initial label $\overline{(k, j)}$ in L and also take it into L_0 as well as $\overline{(k, j)} : {p_{k,j}}^+$ into f_L which allows for starting with $p_{k,j}$ using the activation $(\overline{(k, j)}, (k, j))$ in A. As by construction $p_{k,j}$ 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 also follow the simple simulation of an applicable rule and its continuation with a direct activation of rule in a set assigned to a node directly reachable from node k by a Y-edge.

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 2. *For any strictly extended type X with unit rules and trap rules,*

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

4 Special Results for Specific Objects

In this section we show computational completeness results for AB-grammars based on corresponding well-known computational completeness for other control mechanisms.

4.1 Special Results for Arrays

In both the one- and the two-dimensional case, it has been shown, see [4], that even matrix grammars without ac are sufficient to generate any recursively enumerable array language, i.e., for $d \in \{1,2\}$, $\mathcal{L}\,(d\text{-}\#\text{-}CFA\text{-}MAT) = \mathcal{L}\,(d\text{-}ARBA)$ (the main reason for such a result is the "#-sensing" ability of the rules of type d-#-CFA). Based on Theorem 3, we immediately infer the following result:

Theorem 7. *For $d \in \{1,2\}$,*

$$\mathcal{L}\,(d\text{-}\#\text{-}CFA\text{-}A1) = \mathcal{L}\,(d\text{-}\#\text{-}CFA\text{-}MAT) = \mathcal{L}\,(d\text{-}ARBA)\,.$$

For arbitrary dimensions $d \in \mathbb{N}$, we have (see [4])

$$\mathcal{L}\,(d\text{-}\#\text{-}CFA\text{-}O) = \mathcal{L}\,(d\text{-}ARBA)\,.$$

Hence, based on Corollary 2 and Theorem 1 we obtain the following result:

Theorem 8. *For any $d \in \mathbb{N}$ and for any control mechanism Y,*
$Y \in \{O, fC, RC, MAT_{ac}, P_{ac}, GC_{ac}^{allfinal}, GC_{ac}, A, AB\}$,

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

4.2 Special Results for Strings

It is well-known, for example see [2], that $\mathcal{L}(CF\text{-}RC) = \mathcal{L}(ARB)$. Based on Theorem 4, we immediately infer the following computational completeness result:

Theorem 9. $\mathcal{L}(CF\text{-}AB) = \mathcal{L}(CF\text{-}RC) = \mathcal{L}(ARB) = RE.$

Based on Corollary 2, we even obtain the following stronger result:

Theorem 10. $\mathcal{L}(CF\text{-}A) = \mathcal{L}(CF\text{-}P_{ac}) = \mathcal{L}(CF\text{-}GC_{ac}) = \mathcal{L}(CF\text{-}RC) = RE.$

4.3 Special Results for Multisets

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 show that even with A-grammars we obtain the following:

Theorem 11. $PsRE = Ps(\mathcal{L}(ARB)) = \mathcal{L}(mARB\text{-}A).$

Proof. It is folklore, for example see [8] and [6], that

$$PsRE = Ps(\mathcal{L}(ARB)) = \mathcal{L}(mARB\text{-}fC) = \mathcal{L}(mARB\text{-}RC),$$

hence, by Theorem 4, we also obtain $PsRE = \mathcal{L}(mARB\text{-}AB)$. Based on Corollary 2, we even obtain $PsRE = \mathcal{L}(mARB\text{-}P_{ac}) = \mathcal{L}(mARB\text{-}A)$. □

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

In this section, we state our main results for context-free string and multiset grammars 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 12. *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 [6] 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'. The simulations in the AB-grammar 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 [3], even for graph-controlled context-free grammars one non-terminal symbol is not enough, hence, the statement immediately follows from Theorem 5.

We now show a similar result for multset grammars.

Theorem 13. *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 ${a_{k+1}}^{{p_1}^{n_1} \ldots {p_k}^{n_k}}$ in register $k+1$. As for example, already shown in [9], 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 ${a_{k+1}}^{{p_1}^{n_1} \ldots {p_k}^{n_k}}$ 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. □

6 Conclusion

We have considered the concept of regulating the applicability of rules based on the application of rules in the preceding step(s) within a very general model for sequential grammars and compared the resulting computational power in relation to various other control mechanisms based on the applicability of rules in the underlying grammar, especially for graph-controlled and matrix grammars as well as random context grammars. Even only using the structural features of the sequences of applied rules, yet not taking into account the features of the underlying objects (e.g., strings, multisets, arrays), general simulation results are obtained. Then we also established some special computational completeness results for string, array, and multiset grammars only using activation of rules. Using both activation and blocking of rules in the case of string and multiset grammars with context-free rules, computational completeness can already be obtained with only two non-terminal symbols, which is a sharp result, as only one non-terminal symbol is not sufficient.

The concept of activation and blocking of rules can also be used when rules are applied in parallel, which is an attractive idea for the area of P systems where several variants of parallel derivation modes are common.

References

1. Cavaliere, M., Freund, R., Oswald, M., Sburlan, D.: Multiset random context grammars, checkers, and transducers. Theoretical Computer Science **372**(2–3), 136–151 (2007)
2. Dassow, J., Păun, Gh.: Regulated Rewriting in Formal Language Theory. Springer-Verlag, Berlin (1989)
3. 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)
4. 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)
5. Freund, R.: Control mechanisms for array grammars on cayley grids. In: Durand-Lose, J., Verlan, S. (eds.) Proceedings of MCU 2018. Lecture Notes in Computer Science, Springer (2018)
6. 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)
7. Freund, R., Oswald, M.: Modelling grammar systems by tissue P systems working in the sequential mode. Fundamenta Informaticae **76**(3), 305–323 (2007)
8. 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-Verlag, Berlin (2001)
9. Minsky, M.L.: Computation: Finite and Infinite Machines. Prentice-Hall, Inc., Upper Saddle River, NJ, USA (1967)
10. Păun, Gh., Rozenberg, G., Salomaa, A. (eds.): The Oxford Handbook of Membrane Computing. Oxford University Press, Inc., New York, NY, USA (2010)
11. Rosenfeld, A.: Picture Languages. AcademicPress, Reading, MA (1979)
12. Rozenberg, G., Salomaa, A. (eds.): Handbook of Formal Languages, 3 volumes. Springer, New York, NY, USA (1997)
13. Wang, P.S.P. (ed.): Array Grammars, Patterns and Recognizers, World Scientific Series in Computer Science, vol. 18. World Scientific Publ., Singapore (1989)
14. The P Systems Website. `http://ppage.psystems.eu/`