The Descriptional Complexity of Rewriting Systems - Some Classical and Non-Classical Models

by
György Vaszil

Doctoral dissertation presented to the
Hungarian Academy of Sciences

2013.

Contents

Chapter 1

Introduction

In this chapter we give a brief overview of the topics presented in this dissertation. First we will introduce the models of computation we will study. We start with the classical model of formal grammars and describe some of their regulated variants. Next we move on to the field of grammar systems dealing with networks of grammars which cooperate in jointly generating a language, and then we introduce the non-classical notion of membrane systems, a chemical type of computing model dealing with the transformations of multisets of objects. Finally we describe our understanding of the term *descriptional complexity*, a field of research which we will be dealing with in the rest of the chapters.

1.1 Formal grammars with regulation

First we deal with regulated grammars, that is, context-free grammars with some additional control mechanism to regulate the use of the rules during the derivations.

The need for rewriting devices which use rules of a simple form but still have a considerable generative power is justified by the study of phenomena occurring in different areas of mathematics, linguistics, or even developmental biology. To study problems in these areas which cannot be described by the capabilities of context-free languages, it is often desirable to construct generative mechanisms which have as many context-free-like properties as possible, but are also able to describe the non-context-free features of the specific problem in question. See (Dassow and Păun, 1989) for a discussion about non-context-free phenomena in different areas, and also (Dassow et al., 1997a) for regulated rewriting in general.

Tree controlled grammars were introduced in (Čulik II and Maurer, 1977) as a pair (G, R) where G is a context-free grammar and R is a regular set, called the control language. The control language contains words composed of the terminal and nonterminal alphabets of G, and it is used to control the work of G by restricting the set of derivations which G is allowed to make. Only those words belong to the generated language of the tree controlled grammar which can be generated by the context-free grammar G, and moreover, which have a derivation tree where all the strings obtained by reading from left to right the symbols labeling nodes which belong to the same level of the tree (with the exception of the last level) are elements of the regular set R.

As it was already shown in (Čulik II and Maurer, 1977), tree controlled grammars are able to generate any recursively enumerable language. Variants of the notion with control sets which are not regular but belong to different classes from the Chomsky hierarchy were studied in (Păun, 1979), the power of subregular control sets were examined in more detail in (Dassow et al., 2010).

We also consider a different way of using the sentential form to control the rule application. It is based on the presence/absence of certain symbols or substrings in the sentential forms. One variant of this general idea is realized by conditional grammars, sometimes also called grammars with regular restriction, see (Frĭs, 1968), (Salomaa, 1973). The derivations in these mechanisms are controlled by regular languages associated to the context-free rules: A rule can only be applied to the sentential form if it belongs to the language associated to the given rule.

A weaker restriction is used in the generative mechanisms called semi-conditional grammars, see (Kelemen, 1984), (Păun, 1985). These are context-free grammars with a permitting and a forbidding condition associated to each production. The conditions are given in the form of two words: a permitting and a forbidding word. A production can only be used on a given sentential form if the permitting word is a subword of the sentential form and the forbidding word is not a subword of the sentential form. Note that semi-conditional grammars are special cases of conditional grammars, since the sets of those sentential forms which satisfy the conditions associated to the rules are regular sets. Semi-conditional grammars are also known to generate the class of recursively enumerable languages.

The study of semi-conditional grammars continued with the introduction of *simple semi-conditional grammars* in (Meduna and Gopalaratnam, 1994). We speak of a simple semi-conditional grammar if each rule has at most one nonempty condition, that is, no controlling context condition at all, or either a permitting, or a forbidding one. Simple semi-conditional grammars were introduced in (Meduna and Gopalaratnam, 1994) where they were also were shown to be able to generate all recursively enumerable languages.

Finally, we will study *scattered context grammars*. Scattered context grammars, introduced in (Greibach and Hopcroft, 1969), also represent a class of generative devices which use the presence/absence of certain symbols or substrings in their current sentential forms to achieve additional control over the application of their rewriting rules.

The productions of these grammars are ordered sequences of context-free rewriting rules which have to be applied in parallel on nonterminals appearing in the sentential form in the same order as the nonterminals on the left-hand sides of the rules appear in the production sequence. Scattered context grammars are known to characterize recursively enumerable languages, see (Gonczarowski and Warmuth, 1989), (Meduna, 1995b), (Meduna, 1995a), and (Virkkunen, 1973).

1.2 Parallel communicating grammar systems

Due to their theoretical and practical importance, computational models based on distributed problem solving systems have been in the focus of interest for a long time. A challenging area of research concerning these systems is to elaborate syntactic frameworks to describe the behavior of communities of communicating agents which cooperate in solving a common problem, where the framework is sufficiently sophisticated but relatively easy to handle. The theory of grammar systems ((Dassow et al., 1997b), (Csuhaj-Varjú et al., 1994)) offers several constructs for this purpose. One of them is the parallel communicating (PC) grammar system (introduced in (Păun and Sântean, 1989)), a model for communities of cooperating problem solving agents which communicate with each other by dynamically emerging requests.

In these systems, several grammars perform rewriting steps on their own sentential forms in a synchronized manner until one or more query symbols, that is, special nonterminals corresponding to the components, are introduced in the strings. Then the rewriting process stops and one or more communication steps are performed by substituting all occurrences of the query symbols with the current sentential forms of the component grammars corresponding to the given symbols. Two types of PC grammar systems are distinguished: In the case of returning systems, the queried component returns to its start symbol after the communication and begins to generate a new string. In non-returning systems the components continue the rewriting of their current sentential forms. The language defined by the system is the set of terminal words generated by a dedicated component grammar, the master. In this framework, the grammars represent problem solving agents, the nonterminals open problems to be solved, the query symbols correspond to questions addressed to the agents, and the generated language represents the problem solution. Non-returning systems describe the case when the agent after communication preserves the information obtained so far, while in the case of returning systems, the agent starts its work again from the beginning after communicating information. The reader may notice that the framework of PC grammar systems is also suitable for describing other types of distributed systems, for example, networks with information providing nodes.

The relationship between the class of languages generated by returning and non-returning PC grammar systems have long been an important open problem in the field of PC grammars. First in (Mihalache, 1994), non-returning centralized PC grammar systems were simulated by returning but non-centralized systems, then in (Dumitrescu, 1996) a simulation was presented also for the general non-centralized case.

The exact characterization of the generative power of PC grammar systems has been an open problem for about ten years, until in (Csuhaj-Varjú and Vaszil, 1999) returning PC grammar systems were shown to characterize the class of recursively enumerable languages. At the same time, (Mandache, 2000) showed that all recursively enumerable languages can also be generated by non-returning systems, but the construction of the proof did not provide an upper bound for the number of component grammars.

Returning to the original motivation, namely, to the description of the behavior of communities of problem solving agents, several natural problems arise. One of them is to study systems where clusters or teams of agents represent themselves as separate units in the problem solving process. Notice that this question is also justified by the area of computer-supported team work or cluster computing. The idea of teams has already been introduced in the theory of grammar systems for so-called cooperating distributed (CD) grammar systems and eco-grammar systems, with the meaning that a team is a collection of components which act simultaneously (see (Kari et al., 1995),(Păun and Rozenberg, 1994), (Mateescu et al., 9394), (ter Beek, 1996), (ter Beek, 1997), (Csuhaj-Varjú and Mitrana, 2000), (Lázár et al., 2009)).

Inspired by these considerations, in (Csuhaj-Varjú et al., 2011) we introduced the notion of a parallel communicating grammar system with clusters of components (a clustered PC grammar system, in short) where instead of individual components, each query symbol refers to a set of components, a (predefined) cluster. Contrary to the original model where any addressee of the issued query is precisely identified, i.e., any query symbol refers to exactly one component, here a cluster is queried, and anyone of its components is allowed to reply. This means that the clusters of components behave as separate units, that is, the individual members of the clusters cannot be distinguished at the systems' level.

1.3 Membrane systems

Membrane systems, or P systems were introduced in (Păun, 2000) as computing models inspired by the functioning of the living cell. Their main components are membrane structures consisting of membranes hierarchically embedded in the outermost skin membrane. Each membrane encloses a region containing a multiset of objects and possibly other membranes. Each region has an associated set of operators working on the objects contained by the region. These operators can be of different types, they can change the objects present in the regions or they can provide the possibility of transferring the objects from one region to another one. The evolution of the objects inside the membrane structure from an initial configuration to a somehow specified end configuration correspond to a computation having a result which is derived from some properties of the specific end configuration. Several variants of the basic notion have been introduced and studied proving the power of the framework, see the monograph (Păun, 2002) for a comprehensive introduction, the recent handbook (Păun et al., 2010) for a summary of notions and results of the area, and (Ciobanu et al., 2006) for various applications.

One of the most interesting variants of the model was introduced in (Păun and Păun, 2002) called P systems with symport/antiport. In these systems the modification of the objects present in the regions is not possible, they may only move through the membranes from one region to another. The movement is described by communication rules called symport/antiport rules associated to the regions. (This phenomenon also has analogues in biology, see (Alberts et al., 1994) for more details).

A symport rule specifies a multiset of objects that might travel through a given membrane in a given direction, an antiport rule specifies two multisets of objects which might simultaneously travel through a given membrane in the opposite directions. The result can be read as the number of objects present inside a previously given output membrane after the system reaches a halting configuration, that is, a configuration when no application of any rule in any region is possible.

Note the important role that the environment plays in the computations of membrane systems which use only communication rules: the type and number of objects inside the system can only be changed by sending out some of them into the environment, and importing some others from the environment. Thus, we need to make assumptions also about the environment in which the system is placed.

The situation is similar also in the case of P colonies, the other membrane system variant we will discuss. They represent a class of membrane systems similar to so-called colonies of simple formal grammars (Kelemen and Kelemenová, 1992) (see also (Csuhaj-Varjú et al., 1994; Dassow et al., 1997b) for basic elements of grammar systems and (Csuhaj-Varjú et al., 1997) for the Artificial Life inspired eco-grammar systems).

1.4 Descriptional complexity - A brief overview of the following chapters

In this dissertation we understand the term *descriptional complexity* as an area of theoretical computer science studying various measures of complexity of grammars, automata, or related system (measuring the succintness of their descriptions) and the relationships, trade-offs, between the different variants of systems for a given measure, or the different variants of measures for a given system.

Descriptional complexity aspects of systems (automata, grammars, rewriting systems, etc.) have been a subject of intensive research since the beginning of computer science, but the field has also been actively studied in recent years. Examples of some early results of the type we are interested in, appeared in (Gruska, 1969) about the size of context-free grammars, and the size measures of the number of nonterminal symbols and the number of productions were also introduced, see also (Kelemenová, 1982) for a survey of "grammatical complexity" of context-free grammars. The succintness of representations of languages by different variants of automata were also considered by several authors, see (Goldstine et al., 2002) and (Holzer and Kutrib, 2011) for more details, and a survey of results in these areas.

It is clear that the fact that a system is able to simulate some universal device implies that its size parameters can be bounded. This holds, since by simulating the universal device, all computations are carried out by a fixed (universal) system (having, therefore, fixed size parameters). On the other hand, it is still interesting to look for the best possible values of the bounds, or to study the relationship of certain size parameters with each other or with other properties of the given system.

In Chapter 3 we consider some variants of regulated grammars. Natural measures of descriptional complexity for a formal grammar (regulated or not) are the number of nonterminal symbols and the number of productions (rewriting rules). In addition to measuring the complexity of the grammar, studying measures which also take into account the complexity of the control mechanism is also of interest. Concerning tree controlled grammars we will study the number of nonterminals of the underlying grammar plus the minmal number of nonterminals necessary to generate the control language which is used to regulate the derivations. In simple semi-conditional grammars, we investigate the number of productions and the size of the context conditions used for controlling the rule application. Finally, for scattered-context grammars, we consider the number of nonterminal symbols and the number of context sensing priductions.

In the Chapter 4 we deal with networks of cooperating grammars and focus our attention on parallel communicating grammar systems. These are systems of cooperating context-free grammars, so instead of the complexity of the individual components, we would like to focus our attention on the complexity of the system as described by its size, that is, the number of components, and the complexity of the cooperation, that is, the communication protocol. Concerning the types of communication, we first describe the relationship of the returning or non-returning variants, then we reduce the complexity of the communication in the network by showing how to group some of the individual components into clusters which are, in some sense, indistinguishable at the systems' level.

In Chapter 5 we first investigate some descriptional complexity aspects of symport/antiport P systems. We consider symport/antiport rules with minimal cooperation, that is, symport rules moving just one object and antiport rules moving two objects, one in each direction. (These are in some sense the "smallest" possible rules of this type.) We examine the number of membranes necessary for systems having these simple rules to reach the maximal power, that is, to generate recursively enumerable sets. In the second part of the chapter we consider P colonies. First we show how to bound the number of cells, the number of programs, or both of these measures simultaneously. Then we further simplify the already very simple components of these systems by introducing insertion/deletion programs in such a way that some cells are only able to export objects into the environment, while others are only able to consume objects from the environment. Finally we restrict the number of objects inside the cells of the P colony and study systems which have only one object, the minimal possible amount, inside the cells.

Chapter 2

Some Preliminary Definitions and Notation

In this chapter we present some general definitions and notation which will be used in the subsequent parts of the text. The reader is assumed to be familiar with the basic notions of formal languages and automata theory, more details can be found in the handbook (Rozenberg and Salomaa, 1997), or the monographs (Salomaa, 1973) and (Dassow and Păun, 1989).

2.1 Generative grammars

A finite set of symbols T is called an *alphabet*. The cardinality, that is, the number of elements of T is denoted by $|T|$. The set of non-empty words over the alphabet T is denoted by T^+; the empty word is ε, and $T^* = T^+ \cup \{\varepsilon\}$. A set $L \subseteq T^*$ is called a language over t. For a word $w \in T^*$ and a set of symbols $A \subseteq T$, we denote the length of w by $|w|$, and the number of occurrences of symbols from A in w by $|w|_A$. If A is a singleton set, i.e., $A = \{a\}$, then we write $|w|_a$ instead of $|w|_{\{a\}}$. The concatenation of two sets $L_1, L_2 \subseteq V^*$, denoted as L_1L_2, is defined as $L_1L_2 = \{w_1w_2 \mid w_1 \in L_1, w_2 \in L_2\}$.

A *generative grammar* G is a quadruple $G = (N, T, S, P)$ where N and T are the disjoint sets of nonterminal and terminal symbols, $S \in N$ is the initial nonterminal, and P is a set of rewriting rules (or productions) of the form $\alpha \to \beta$ where $\alpha, \beta \in (N \cup T)^*$ with $|\alpha|_N \geq 1$. A string v can be derived from a string u, denoted as $u \Rightarrow v$ for some $u, v \in (N \cup T)^*$, if they can be written as $u = u_1\alpha u_2$, $v = v_1\beta v_2$ for a rewriting rule $\alpha \to \beta \in P$.

The reflexive and transitive closure of the relation $\Rightarrow$ is denoted by $\Rightarrow^*$. The language generated by the grammar G is the set of terminal strings which can be derived from the initial nonterminal, that is, $L(G) = \{w \in T^* \mid S \Rightarrow^* w\}$. It is known that any recursively enumerable language can be generated by a generative grammar defined as above.

A generative grammar is *context-free*, if the rewriting rules $\alpha \to \beta$ are such, that $\alpha \in N$. A context-free grammar is *regular*, if in addition to the property that $\alpha \in N$, it also holds, that $\beta \in T^* \cup T^*N$. The classes of recursively enumerable, context-free, and regular grammars and languages are denoted by RE, CF, REG, $\mathcal{L}$(RE), $\mathcal{L}$(CF), and $\mathcal{L}$(REG), respectively.

2.2 Counter machines

An *n-counter machine* $M = (T \cup \{Z, B\}, E, R, q_0, q_F)$, $n \geq 1$, is an $n+1$-tape Turing machine where T is an *alphabet*, E is a set of *internal states* with two distinct elements $q_0, q_F \in E$, and R is a set of *transition rules*. The machine has a read-only input tape and n semi-infinite storage tapes (the counters). The alphabet of the storage tapes contains only two symbols, Z and B (blank), while the alphabet of the input tape is $T \cup \{B\}$. The symbol Z is written on the first, leftmost cells of the storage tapes which are scanned initially by the storage tape heads, and may never appear on any other cell. An integer t can be stored by moving a tape head t cells to the right of Z. A stored number can be incremented or decremented by moving the tape head right or left. The machine is capable of checking whether a stored value is *zero* or not by looking at the symbol scanned by the storage tape heads. If the scanned symbol is Z, then the value stored in the corresponding counter is *zero* (which cannot be decremented since the tape head cannot be moved to the left of Z). We will sometimes refer to the number stored in the counter as the contents of the counter, and we will call a counter empty when the stored number is zero.

The rule set R contains transition rules of the form $(q, x, c_1, \ldots, c_n) \to (q', e_1, \ldots, e_n)$ where $x \in T \cup \{B\} \cup \{\varepsilon\}$ corresponds to the symbol scanned on the input tape in state $q \in E$, and $c_1, \ldots, c_n \in \{Z, B\}$ correspond to the symbols scanned on the storage tapes. By a rule of the above form, M enters state $q' \in E$, and the counters are modified according to $e_1, \ldots, e_n \in \{-1, 0, +1\}$. If $x \in T \cup \{B\}$, then the machine scanned x on the input tape, and the head moves one cell to the right; if $x = \varepsilon$, then the machine performs the transition irrespectively of the scanned input symbol, and the reading head does not move.

A configuration of the n-counter machine M can be denoted by a $(n+2)$-tuple $(w, q, j_1, \ldots, j_n)$ where $w \in T^*$ is the unread part of the input word written on the input tape, $q \in Q$ is the state of the machine, and $j_i \in \mathbb{N}$, $1 \leq i \leq n$, is the value stored on the ith counter tape of M. For two configurations, C and C', we write $C \vdash_a C'$ if M is capable of changing the configuration $C = (aw, q, j_1, \ldots, j_n)$ to $C' = (w, q', j'_1, \ldots, j'_n)$ by reading a symbol $a \in T \cup \{\varepsilon\}$ from the input tape and applying one of its transition rules.

A word $w \in T^*$ is accepted by the machine if starting in the initial configuration $(w, q_0, 0, \ldots, 0)$, the input head eventually reaches and reads the rightmost non-blank symbol on the input tape, and M is in the accepting state q_F, that is, it reaches a configuration $(\varepsilon, q_F, j_1, \ldots, j_n)$ for some $j_i \in \mathbb{N}$, $1 \leq i \leq n$. The language accepted by M is denoted by $L(M)$.

It is known that 2-counter machines (written from now on as *two-counter machines*) are computationally complete; they are able to recognize any recursively enumerable language (see (Fischer, 1966)). Obviously, n-counter machines for any $n > 2$ are of the same accepting power.

We will also use a variant of the above notion from (Minsky, 1967), the notion of a register machine. It consists of a given number of registers each of which can hold an arbitrarily large non-negative integer number (we say that the register is empty if it holds the value zero), and a set of labeled instructions which specify how the numbers stored in registers can be manipulated.

Formally, a *register machine* is a construct $M = (m, H, l_0, l_h, R)$, where m is the number of registers, H is the set of instruction labels, l_0 is the start label, l_h is the halting label, and R is the set of instructions; each label from H labels only one instruction from R. There are several types of instructions which can be used. For $l_i, l_j, l_k \in H$ and $r \in \{1, \ldots, m\}$ we have

- $l_i : (\mathtt{nADD}(r), l_j, l_k)$ - *nondeterministic add*: Add 1 to register r and then go to one of the instructions with labels l_j or l_k, nondeterministically chosen.

- $l_i : (\mathtt{ADD}(r), l_j)$ - *deterministic add*: Add 1 to register r and then go to the instruction with label l_j.

- $l_i : (\mathtt{SUB}(r), l_j)$ - *subtract*: If register r is non-empty, then subtract one from it, otherwise leave it unchanged, and go to the instruction with label l_j in both cases.

- $l_i : (\texttt{CHECK(r)}, l_j, l_k)$ - *zero check*: If the value of register r is zero, go to instruction l_j, otherwise go to l_k.
- $l_h : \texttt{HALT}$ - *halt*: Stop the machine.

A register machine M computes a set $N(M)$ of numbers in the following way: It starts with empty registers by executing the instruction with label l_0 and proceeds by applying instructions as indicated by the labels (and made possible by the contents of the registers). If the halt instruction is reached, then the number stored at that time in register 1 is said to be computed by M. Because of the nondeterminism in choosing the continuation of the computation in the case of `nADD` instructions, $N(M)$ can be an infinite set.

Note that register machines can be defined without the `nADD` instructions as deterministic computing devices which compute some function of an input value placed initially in an input register. It is known (see, e.g., (Minsky, 1967)) that in this way they can compute all functions which are Turing computable. We added the nondeterministic add instruction in order to obtain a device which generates sets of numbers starting from a unique initial configuration. As any recursively enumerable set can be obtained as the range of a Turing computable function on the set of non-negative integers, this way we can generate any recursively enumerable set of numbers.

2.3 Multisets

We end this chapter by presenting the necessary notions and notions necessary for the use of multisets. A *multiset* over an arbitrary (not necessarily finite) set V is a mapping $M : V \to \mathbb{N}$ which assigns to each object $a \in V$ its multiplicity $M(a)$ in M. The support of M is the set $supp(M) = \{a \mid M(a) \geq 1\}$. If V is a finite set, then M is called a finite multiset. The set of all finite multisets over the finite set V is denoted by V^*. We say that $a \in M$ if $M(a) \geq 1$, the cardinality of M, $card(M)$ is defined as $card(M) = T_{a \in M} M(a)$. For two multisets $M_1, M_2 : V \to \mathbb{N}$, $M_1 \subseteq M_2$ holds, if for all $a \in V$, $M_1(a) \leq M_2(a)$. The union of M_1 and M_2 is defined as $(M_1 \cup M_2) : V \to \mathbb{N}$ with $(M_1 \cup M_2)(a) = M_1(a) + M_2(a)$ for all $a \in V$, the difference is defined for $M_2 \subseteq M_1$ as $(M_1 - M_2) : V \to \mathbb{N}$ with

$(M_1 - M_2)(a) = M_1(a) - M_2(a)$ for all $a \in V$. A multiset M is empty if its support is empty, $supp(M) = \emptyset$.

We will represent a finite multiset M over V by a string w over the alphabet V with $|w|_a = M(a)$, $a \in V$, and ε will represent the empty multiset.

Chapter 3

Grammars with Regulation

In this chapter we present results about computational models studied in the area of regulated rewriting. These models are constructed by adding some kind of a control mechanism to ordinary (usually context-free) grammars which restricts the application of the rules in such a way that some of the derivations which are possible in the usual derivation process are eliminated from the controlled variant. This means that the set of words generated by the controlled device is a subset of the original (context-free) language generated without the control mechanism. As these generated subsets can be non-context-free languages, these mechanisms are usually more powerful than ordinary (context-free) grammars. See (Dassow and Păun, 1989) for more details, and also (Dassow et al., 1997a) for regulated rewriting in general.

Since it has been always important to describe formal languages as concisely and economically as possible, it is of interest to study these mechanisms from the point of view of descriptional complexity. As the measures of complexity of the descriptions, we will study the number of nonterminals, the number of production rules, and the complexity of the added control mechanism.

3.1 Preliminaries

In the next sections we will use normal form results from (Geffert, 1988) stating that all recursively enumerable languages can be generated by grammars in which there are only a few rules which are not context-free (linear context-free, to be more precise). The idea of the normal forms is based on the fact that for each $L \in \mathcal{L}(\mathrm{RE})$ there are two homomorphisms h_1, h_2, such that $w \in L$, if and only if there is an α with $h_1(\alpha) = h_2(\alpha)w$. (This can be shown similarly to the undecidability of the so called Post correspondence problem.) Based on the above homomorphisms, one can construct a grammar which generates words of the form $w\beta_1\beta_2$ where $\beta_1 = h(h_1(\alpha))$ and β_2 is the reverse of $h(h_2(\alpha)w)$ for some encoding h, and then generates w by deleting $\beta_1\beta_2$ if and only if they are mirror images of each other.

These ideas were converted into normal form results by (Geffert, 1988) as follows. If $L \subseteq T^*$ is a recursively enumerable language, then L can be generated by a grammar

$$G = (N, T, P \cup \{AB \to \varepsilon,\ CD \to \varepsilon\}, S),$$

such that $N = \{S, S', A, B, C, D\}$, and P contains only context-free productions. Furthermore, the context-free rules of G are of the form

$$\begin{array}{ll} S \to zSx, & \text{where } z \in \{A, C\}^*, x \in T, \\ S \to S', & \\ S' \to uS'v, & \text{where } u \in \{A, C\}^*, v \in \{B, D\}^*, \\ S' \to \varepsilon. & \end{array}$$

Considering the rules above, we can distinguish three phases in the generation of a terminal word $x_1 \dots x_n, x_i \in T, 1 \leq i \leq n$.

1. $S \Rightarrow^* z_n \dots z_1 S x_1 \dots x_n \Rightarrow z_n \dots z_1 S' x_1 \dots x_n$,

where $z_i \in \{A, C\}^*, 1 \leq i \leq n$.

2. $z_n \dots z_1 S' x_1 \dots x_n \Rightarrow^* z_n \dots z_1 u_m \dots u_1 S' v_1 \dots v_m x_1 \dots x_n \Rightarrow$
   $z_n \dots z_1 u_m \dots u_1 v_1 \dots v_m x_1 \dots x_n$,

where $u_j \in \{A, C\}^*, v_j \in \{B, D\}^*, 1 \leq j \leq m$. Now the terminal string $x_1 \dots x_n$ is generated by G if and only if, using $AB \to \varepsilon$ and $CD \to \varepsilon$ the substring $z_n \dots z_1 u_m \dots u_1 v_1 \dots v_m$ can be erased

3. $z_n \dots z_1 u_m \dots u_1 v_1 \dots v_m x_1 \dots x_n \Rightarrow^* x_1 \dots x_n$.

This can be successful if and only if $z_n \dots z_1 u_m \dots u_1 = g(v_1 \dots v_m)^R$ where $g : \{B, D\} \to \{A, C\}$ is a morphism with $g(B) = A$ and $g(D) = C$.

We will refer to the three stages of a derivation of a grammar in the normal form above as the *first*, the *second*, and the *third phase*.

It is not difficult to see that if we encode the nonterminal A to XY, the nonterminal B to Z, the nonterminal C to X, and the nonterminal D to YZ, then one erasing rule of the form $XYZ \to \varepsilon$ is sufficient to have the same effect as $AB \to \varepsilon$ and $CD \to \varepsilon$ together.

Thus, all recursively enumerable languages $L \subseteq T^*$ can be generated by a grammar
$$G = (N, T, P \cup \{ABC \to \varepsilon\}, S),$$
such that $N = \{S, S', A, B, C\}$, and P contains only context-free productions of the form
$$\begin{aligned}
&S \to zSx, && \text{where } z \in \{A, B\}^*, x \in T,\\
&S \to S', &&\\
&S' \to uS'v, && \text{where } u \in \{A, B\}^*, v \in \{B, C\}^*,\\
&S' \to \varepsilon. &&
\end{aligned}$$
The three phases of the derivation can also be distinguished in this case. In order to successfully generated a terminal word $x_1 \dots x_n, x_i \in T, 1 \leq i \leq n$, first we need to generate a sentential form with the terminal prefix $x_1 \dots x_n$, then continue with the generation of the nonterminal suffix (these are the first two phases), and finally (in the third phase) we need to erase all the nonterminals.

Notice that the nonterminal S' and the rule $S \to S'$ can be eliminated in both normal form variants above. The three derivational phases cannot be "mixed" in any successful derivation even if only S is used instead of S and S'.

The number of nonterminals of a generative grammar $G = (N, T, S, P)$ is denoted by $\mathrm{Var}(G)$, that is, $\mathrm{Var}(G) = |N|$. For a language L and a class of grammars $X \in \{\mathrm{REG}, \mathrm{CF}\}$, we denote by $\mathrm{Var}_X(L)$ the minimal number of nonterminals necessary to generate L with a grammar of type X, that is, $\mathrm{Var}_X(L) = \min\{\mathrm{Var}(G) \mid L = L(G) \text{ and } G \text{ is of type } X \in \{\mathrm{REG}, \mathrm{CF}\}\}$.

3.2 Tree controlled grammars - The number of nonterminals and the complexity of the control langauge

Investigations concerning the nonterminal complexity of tree controlled grammars began in (Turaev et al., 2011b) where this measure was defined as the sum of the number of nonterminals of the context-free grammar and the number of nonterminals which are necessary to generate the regular control language. They showed that nine nonterminals altogether are sufficient to generate any recursively enumerable language with a tree controlled grammar. Then this bound was improved to seven in (Turaev et al., 2011a) by simulating a phrase structure grammar being in the Geffert normal form (see the second variant presented in Section 3.1 without the distinguished S and S'), that is, having four nonterminals and only linear context-free productions, together with one non-context-free production in addition which is able to erase three neighboring nonterminals as it is of the form $ABC \to \varepsilon$.

In this section we show how to improve the bound from seven to six using a similar technique as in (Turaev et al., 2011a), but simulating a grammar being in a different version of the Geffert normal form (see the first variant presented in Section 3.1): Instead of the non-context-free production $ABC \to \varepsilon$, it has two non-context-free productions $AB \to \varepsilon$, $CD \to \varepsilon$. Since the nonterminals, A, B, C, D, will be encoded in the simulating tree controlled grammar by strings over just two symbols, it does not matter that instead of the symbols A, B, C we need to simulate a grammar which uses more symbols, A, B, C, D. On the other hand, the fact that instead of three, only two neighboring symbols have to be deleted simultaneously, helps to construct a simulating tree controlled grammar which uses one nonterminal less than the one in (Turaev et al., 2011a), thus helps to reduce the bound on the number of necessary nonterminals from seven to six.

For the introduction of tree controlled grammars, we need the notion of the derivation tree. An ordered tree is a *derivation tree* of a context-free grammar $G = (N, T, S, P)$ if its nodes are labeled with symbols from $N \cup T \cup \{\varepsilon\}$ in a way which satisfies the following properties: (a) The root is labeled with S, (b) the leaves are labeled with symbols from $T \cup \{\varepsilon\}$, and (c) every interior vertex is labeled from N in such a way that if a vertex has a label $A \in N$ and its children are labeled from left to right with $x_1, x_2, \ldots, x_m$, $x_i \in N \cup T \cup \{\varepsilon\}$, $1 \leq i \leq m$, then there is a production $A \to x_1 x_2 \ldots x_m$ in P. A derivation tree corresponds to a terminal word w from $L(G)$ if the concatenation of the symbols labeling the leaves of the tree from left to right coincide with w.

The distance of a vertex t from the root is the length of the shortest path leading to t from the root node. The string corresponding to the ith level of a derivation tree for some $i \geq 0$ is the word obtained by concatenating from left to right the symbols labeling those nodes of the tree which are at distance i from the root.

Definition 3.2.1. A *tree controlled grammar* is a pair $G = (G', R)$ where $G' = (N, T, S, P)$ is a context-free grammar and R is a regular language over the alphabet $N \cup T$. The language $L(G)$ generated by the tree controlled grammar G contains all words $w \in T^*$ from $L(G')$ which have a derivation tree where the strings corresponding to each different level, except the last one, belong to the regular set R.

The nonterminal complexity of a tree controlled grammar $G = (G', R)$ is the number of nonterminals of G' plus the minimal number of nonterminals that a regular grammar needs for generating the language R, that is, $\mathrm{Var}(G) = \mathrm{Var}(G') + \mathrm{Var}_{\mathrm{REG}}(R)$.

To illustrate the notion of tree controlled grammars, let us recall an example from (Dassow and Păun, 1989).

Example 3.2.1. *Let $G = (G', R)$ where $G' = (\{S\}, \{a\}, S, P)$ with*

$$P = \{S \to SS, S \to a\} \text{ and } R = \{S\}^*.$$

As the control language R contains words which are sequences of the nonterminal symbol S, all the nodes of every level (except the last one) of the derivation tree of a word $w \in L(G)$ are labeled by the symbol S. This means that all the nonterminals, with the exception of the ones labeling the nodes directly above the last level of the tree, are rewritten by the rule $S \to SS$, and the nonterminals of the level directly above the last one are rewritten by $S \to a$. Thus, the language generated by G is $L(G) = \{a^{2^n} \mid n \geq 0\}$.

We will also need the notion of *unique-sum sets* which was introduced in (Frisco, 2009) as follows. A set of nonnegative integers $U = \{u_1, \ldots, u_p\}$ having the sum $\sigma_U = \Sigma_{i=1}^{p} u_i$ is said to be a unique-sum set, if the equation $\Sigma_{i=1}^{p} c_i u_i = \sigma_U$ for $c_i \in \mathbb{N}$ has the only solution $c_i = 1,\ 1 \leq i \leq p$. Examples of unique-sum sets are $\{2, 3\}$, $\{4, 6, 7\}$, or $\{8, 12, 14, 15\}$, while the set $\{4, 5, 6\}$ is not unique-sum, as $4 + 5 + 6 = 15 = 5 + 5 + 5$. It is clear that any subset of a unique-sum set is unique-sum, and that the sum of any two numbers from the set, $\sigma_{i,j} = u_i + u_j$, cannot be produced as the linear combination of elements of the set in any other way.

3.2.1 The number of nonterminals

Now we show that every recursively enumerable language can be generated by a tree controlled grammar with six nonterminals.

Theorem 3.2.1. *For any recursively enumerable language L, there exists a tree controlled grammar G with $L = L(G)$, such that* $\mathrm{Var}(G) = 6$.

Proof. Let $L \subseteq T^*$ be a recursively enumerable language generated by the Geffert normal form grammar $G_1 = (\{S, A, B, C, D\}, T, S, P)$ where $T = \{a_1, a_2, \dots, a_t\}$ and $P = \{AB \to \varepsilon, CD \to \varepsilon, S \to \varepsilon\} \cup \{S \to z_i S a_i, S \to u_j S v_j \mid z_i, u_j \in \{A, C\}^*, v_j \in \{B, D\}^*, 1 \le i \le t, 1 \le j \le s\}$.

Let us define the morphism $h : \{A, B, C, D\}^* \to \{0, \$\}^*$ by $h(A) = \$0^6\$$, $h(B) = \$0^{10}\$$, $h(C) = \$0^{12}\$$, $h(D) = \$0^{13}\$$ which encodes four of the nonterminals of the grammar G_1 as strings over two symbols. Notice that the length of the coding sequences forms the unique-sum set $\{8, 12, 14, 15\}$.

Let us now construct the tree controlled grammar $G = (G', R)$ where $G' = (N, T, S, P')$ with $N = \{S, S', \$, 0, \#\}$,

$$\begin{aligned} P' = {} & \{S \to h(z)Sa \mid S \to zSa \in P, a \in T, z \in \{A, C\}^*\} \cup \\ & \{S \to S'\} \cup \\ & \{S' \to h(u)S'h(v) \mid S \to uSv \in P, u \in \{A, C\}^*, v \in \{B, D\}^*\} \cup \\ & \{S' \to \varepsilon, \$ \to \$, \$ \to \#, 0 \to 0, 0 \to \#, \# \to \varepsilon\}, \end{aligned}$$

and

$$R = (\{S, S'\} \cup T \cup X_1 \cup X_2)^* \{\#^{20}, \#^{29}, \varepsilon\},$$

where

$$X_1 = \{\$0^6\$, \$0^{10}\$, \$0^{12}\$, \$0^{13}\$\}, \tag{3.1}$$
$$X_2 = \{\$0^6\$, \$0^{10}\$\}\{\#^{20}, \#^{29}\}\{\$0^{12}\$, \$0^{13}\$\}. \tag{3.2}$$

First we show that any terminal derivation of G_1 can be simulated by the tree controlled grammar G, that is, $L(G_1) \subseteq L(G)$. Let $w \in L(G_1)$ and let

$$S \Rightarrow^* zSw \Rightarrow^* zuSvw \Rightarrow zuvw \tag{3.3}$$

be the first and the second phases of a derivation of w in G_1 where $z, u \in \{A, C\}^*, v \in \{B, D\}^*$. We can generate $h(zuv)w$, the encoded version of $zuvw$ with the rules of G as follows

$$S \Rightarrow^* h(z)Sw \Rightarrow h(z)S'w \Rightarrow^* h(zu)S'h(v)w \Rightarrow h(zuv)w, \tag{3.4}$$

$h(zu) \in \{\$0^6\$, \$0^{12}\$\}^*, h(v) \in \{\$0^{10}\$, \$0^{13}\$\}^*$. If we use the chain rules, $\$ \to \$$ and $0 \to 0$, we can make sure that the word corresponding to each level of the derivation tree belongs to the regular set R, and moreover, that $h(zuv)$ is the string corresponding to the last level of the derivation tree which belongs to the derivation (3.4) of G above simulating the first two phases of the derivation of the word w in G_1 depicted at (3.3).

Now w can be derived in G_1 if zuv can be erased by using the rules $AB \to \varepsilon$ and $CD \to \varepsilon$. If AB or CD is a substring of zuv, then $h(AB) = \$0^6\$\$0^{10}\$$ or $h(CD) = \$0^{12}\$\$0^{13}\$$ is a substring of $h(zuv)$, thus, one of the derivations

$$h(zuv) \Rightarrow h(zu')\#^{20}h(v')w \Rightarrow h(zu'v')w,$$

or

$$h(zuv) \Rightarrow h(zu')\#^{29}h(v')w \Rightarrow h(zu'v')w$$

can be executed in G using the chain rules as above, and the rules $0 \to \#$, $\$ \to \#$, $\# \to \varepsilon$ in such a way that $h(zu'v')$ is again the string which corresponds to the last level of the derivation tree of $h(zu'v')w$.

It is clear, that whenever zuv can be erased in G_1, then $h(zuv)$ can also be erased in G, thus, w can also be generated by G which means that $w \in L(G)$.

Now we show that $L(G) \subseteq L(G_1)$. To see this, we have to show that any $w \in L(G)$ can also be generated by G_1. Consider the derivation tree corresponding to a derivation of $w \in L(G)$ in G and look at the words corresponding to the different levels of the tree.

Notice the following: (A) There is no symbol # appearing in the levels as long as S or S' is present. This statement holds because the words in R have a special form: They are the concatenations of "complete" coding sequences of A, B, C, or D, that is, each subword over $\{\$, 0\}$ is a concatenation of coding strings of the form $\$0^i\$$ (for some $i \in \{6, 10, 12, 13\}$). Thus, if # symbols appear in a word corresponding to a level of the derivation tree, then either all symbols of such a coding subword are rewritten to #, or no symbol of such a coding subword is rewritten to #. Recall that the lengths of these coding sequences form a unique-sum set, $\{8, 12, 14, 15\}$, thus, 20 and 29 can only arise through some linear combination of the elements as $20 = 8 + 12$, and $29 = 14 + 15$. This, together with the above considerations, means that $\#^{20}$ or $\#^{29}$ can only be obtained by rewriting all symbols of $\$0^6\$\$0^{10}\$$ or $\$0^{12}\$\$0^{13}\$$ to #. Notice that when S or S' is present, then no sequence over $\{\$0^6\$, \$0^{12}\$\}$ can be followed directly by a sequence over $\{\$0^{10}\$, \$0^{13}\$\}$, thus, when S or S' is present no neighboring code sequences of length 20 or

29 can occur which means that the words cannot contain $\#^{20}$ or $\#^{29}$ as a subsequence.

Statement (A) above implies that as long as S or S' is present in the words corresponding to the levels of the derivation tree, the chain rules $\$ \to \$$ and $0 \to 0$ have to be used on the symbols $\$, 0$ when passing to the next level of the derivation tree. This is also true for the word corresponding to the first level in which S' disappears after using a rule of the form $S' \to h(u)h(v)$, since $uv \neq \varepsilon$. Note that the part of the derivations of G with the presence of S and the presence of S' corresponds to the first and the second phases of the derivations of the Geffert normal form grammar G_1, respectively.

Consider now the first such level of the derivation tree corresponding to a derivation of w in G in which none of the symbols S or S' are present. From the above considerations it follows that the string corresponding to this level has the form $h(zu)h(v)$ where $h(zu) \in \{\$0^6\$, \$0^{12}\$\}^*$, $h(v) \in \{\$0^{10}\$, \$0^{13}\$\}^*$, and $zuvw$ can also be derived in the grammar G_1.

Note also: (B) There cannot be two distinct subsequences of the symbols $\#$ in any of the words corresponding to any level of the derivation tree of the word $w \in L(G)$. To see this, consider the first level of the tree which is without S and S', and denote the string corresponding to this level as $h(zuv)$. Recall that $h(zuv) = \alpha_1\alpha_2$ where $\alpha_1 \in \{\$0^6\$, \$0^{12}\$\}^*$, $\alpha_2 \in \{\$0^{10}\$, \$0^{13}\$\}^*$, so subwords of the form $\{\$0^6\$, \$0^{12}\$\}^*\{\#^{20}, \#^{29}\}\{\$0^{10}\$, \$0^{13}\$\}^*$ can only be present in the words corresponding to subsequent levels of the tree in such a way that the sequence of $\#$ symbols is the result of rewriting a suffix of α_1 and a prefix of α_2 to $\#$.

Property (B) above implies that in order to be in the control set R, a word which corresponds to some level of the derivation tree and also contains $\#$, must be of the form $\{\$0^6\$, \$0^{12}\$\}^*\{\#^{20}, \#^{29}\}\{\$0^{10}\$, \$0^{13}\$\}^*$ where $\#^{20}$ or $\#^{29}$ is obtained from the word corresponding to the previous level of the tree by rewriting each symbol in a substring $\$0^6\$\$0^{10}\$$ or $\$0^{12}\$\$0^{13}\$$ to $\#$, respectively. Therefore, the word corresponding to the previous level of the tree is either $\alpha_1'\$0^6\$\$0^{10}\$\alpha_2'$ or $\alpha_1'\$0^{12}\$\$0^{13}\$\alpha_2'$ where α_1' and α_2' satisfy either $h^{-1}(\alpha_1')AB\ h^{-1}(\alpha_2') = zuv$ or $h^{-1}(\alpha_1')CD\ h^{-1}(\alpha_2') = zuv$ provided that $\alpha_1\alpha_2 = h(zuv)$.

This means that the uncoded version of the word corresponding to the next level of the derivation tree, where the $\#$ symbols are erased, can also be derived in G_1 by the rules $AB \to \varepsilon$ or $CD \to \varepsilon$. More precisely, the word corresponding to the next level of the derivation tree is either of the form $\alpha_1'\alpha_2'$ or $\alpha_1''\{\#^{20}, \#^{29}\}\alpha_2''$, all of them corresponding to the sentential form

$h^{-1}(\alpha'_1)h^{-1}(\alpha'_2)w$ which can also be derived in G_1.

Continuing the above reasoning, we obtain that the word corresponding to the level which is above the last one in the derivation tree of $w \in L(G)$ is of the form $\#^{20}$ or $\#^{29}$, corresponding to the sentential form ABw or CDw in G_1, thus, if w can be generated by the tree controlled grammar G with the control set R, then w can also be generated by the Geffert normal form grammar G_1.

This means that $L(G) \subseteq L(G_1)$, and since we have already shown the that the opposite inclusion holds, we have $L(G) = L(G_1)$. As the control set R can be generated by the regular grammar $G_2 = (\{A\}, T \cup \{0, \$, \#, S, S'\}, A, P_2)$ with $P_2 = \{A \to xA, A \to \#^{20}, A \to \#^{29}, A \to \varepsilon \mid x \in \{S, S'\} \cup T \cup X_1 \cup X_2$ where X_1 and X_2 is defined as above at (3.1) and (3.2), respectively, and this grammar has just one nonterminal, we have proved the statement of the theorem. □

3.2.2 Remarks

We have shown how to reduce the nonterminal complexity of tree controlled grammars from seven to six. This result first appeared in (Vaszil, 2012). We have used a similar technique as was used in (Turaev et al., 2011a), namely, we have simulated phrase structure grammars in the Geffert normal. There are two important differences, however, which have made it possible to realize the simulation with six nonterminals which number is one less than needed in (Turaev et al., 2011a) which contains the previously known best result.

First, instead of the normal form with the single erasing rule $ABC \to \varepsilon$, we have used the variant with two erasing rules $AB \to \varepsilon$, $CD \to \varepsilon$, thus we needed to simulate the simultaneous erasing of only two nonterminals, as opposed to the simultaneous erasing of the three symbols in $ABC \to \varepsilon$. The increase of the number of nonterminals (from A, B, C to A, B, C, D) does not show up in the nonterminal complexity of the simulating grammar, since the nonterminal symbols are coded as words over two nonterminals.

The second modification concerns the way of coding the four nonterminals. We have used code words with lengths which form a unique-sum set. This made the decoding possible with one less nonterminal than in (Turaev et al., 2011a).

Other language classes were also examined from the point of view of nonterminal complexity with respect to tree controlled grammars in (Turaev et al., 2012). Regular, linear, and regular simple matrix languages are shown to be generated with three nonterminals which is an optimal bound, since there are regular languages which cannot be generated with two nonterminals. For context-free languages, on the other hand, four nonterminals are sufficient, although it is not known whether this bound cannot be decreased to three or not. In this paper it was also proved that any recursively enumerable language can be generated by a tree controlled grammar with seven nonterminals, but at the time of writing this dissertation, the result presented in the previous section is still the best result concerning the nonterminal complexity of tree controlled grammars generating recursively enumerable languages.

3.3 Simple semi-conditional grammars - The number of conditional productions and the length of the context conditions

In the case of semi-conditional grammars, the complexity of productions is measured by their degree, defined as the maximal length of the context conditions as a pair of integers, (i, j), where i and j is the length of the longest permitting and the longest forbidding word, respectively. In (Păun, 1985), recursively enumerable languages were characterized by semi-conditional grammars of degree $(2, 1)$ or $(1, 2)$, while grammars of degree $(1, 1)$ without erasing productions were shown to generate only a subclass of context-sensitive languages. The investigation of the generative power of grammars having only permitting or only forbidding conditions were also started, the language classes generated by grammars of degree $(1, 0)$ or $(0, 1)$ without erasing productions were shown to be strictly included in the class of context-sensitive languages.

In simple semi-conditional grammars, each rule has at most one nonempty condition, that is, no controlling context condition at all, or either a permitting, or a forbidding one. In (Meduna and Gopalaratnam, 1994), simple semi-conditional grammars of degree $(1, 2)$ and $(2, 1)$ were shown to be able to generate all recursively enumerable languages, but the number of rules necessary to obtain this power was unbounded.

The study of the number of productions necessary to obtain all recursively enumerable languages has been started in (Meduna and Švec, 2002) where the authors prove that the class of recursively enumerable languages can be characterized by simple semi-conditional grammars of degree $(2, 1)$ with only twelve conditional productions.

In this section we show how to improve the above mentioned bounds. We prove that ten conditional productions are sufficient to generate all recursively enumerable languages with grammars of degree $(2, 1)$, or eight of them are sufficient to generate all recursively enumerable languages with grammars of degree $(3, 1)$.

Definition 3.3.1. A *semi-conditional grammar* is a construct $G = (N, T, P, S)$ with nonterminal alphabet N, terminal alphabet T, a start symbol $S \in N$, and a set of productions of the form $(X \to \alpha, u, v)$ with $X \in N$, $\alpha \in (N \cup T)^*$, and $u, v \in (N \cup T)^+ \cup \{0\}$ where $0 \notin (N \cup T)$ is a special symbol. If either $u \neq 0$ or $v \neq 0$, then the production is said to be *conditional*.

A semi-conditional grammar G has *degree* (i, j) if for all productions $(X \to \alpha, u, v)$, $u \neq 0$ implies $|u| \leq i$, and $v \neq 0$ implies $|v| \leq j$.

We speak of a *simple semi-conditional grammar* if each production has at most one nonempty condition, that is, if $G = (N, T, P, S)$ is a simple semi-conditional grammar, then $(X \to \alpha, u, v) \in P$ implies $0 \in \{u, v\}$.

We say that $x \in (N \cup T)^+$ directly derives $y \in (N \cup T)^*$ according to the rule $(X \to \alpha, u, v) \in P$, denoted by $x \Rightarrow y$, if $x = x_1Xx_2$, $y = x_1\alpha x_2$ for some $x_1, x_2 \in (N \cup T)^*$, and furthermore, $u \neq 0$ implies that $u \in sub(x)$ and $v \neq 0$ implies that $v \notin sub(x)$.

If we denote the reflexive and transitive closure of $\Rightarrow$ by $\Rightarrow^*$, then the language generated by a semi-conditional grammar G is $L(G) = \{w \in T^* \mid S \Rightarrow^* w\}$.

Now let us recall an example from (Meduna and Gopalaratnam, 1994).

Example 3.3.1. *Let $G = (\{S, A, X, C, Y\}, \{a, b, c\}, P, S)$ be a simple semi-conditional grammar with*

$$\begin{aligned} P \;=\; & \{(S \to AC, 0, 0), A \to aXb, Y, 0), (C \to Y, A, 0), (Y \to Cc, 0, A), \\ & (A \to ab, Y, 0), (Y \to c, 0, A), (X \to A, C, 0)\}. \end{aligned}$$

Note that G is a simple semi-conditional grammar of degree $(1, 1)$, having six conditional productions.

Derivations of G start with

$$S \Rightarrow AC \Rightarrow AY.$$

Assume that a string of the form $a^kAb^kYc^k$, $k \geq 0$, can be generated by G. We show that either $a^{k+1}Ab^{k+1}Yc^{k+1}$ or $a^{k+1}b^{k+1}c^{k+1}$ is generated by G, or the derivation is blocked.

The context conditions only allow the use of one of the rules $A \to aXb$ or $A \to ab$. If the first one is chosen, then

$$a^kAb^kYc^k \Rightarrow a^{k+1}Xb^{k+1}Yc^k$$

is obtained and then one of the rules $Y \to c$ or $Y \to Cc$ must be used. If $Y \to c$ is chosen, the derivation can not continue because X can only be rewritten in the presence of a symbol C, so we must have

$$a^{k+1}Xb^{k+1}Yc^k \Rightarrow a^{k+1}Xb^{k+1}Cc^{k+1} \Rightarrow a^{k+1}Ab^{k+1}Cc^{k+1} \Rightarrow a^{k+1}Ab^{k+1}Yc^{k+1}.$$

If having $a^kAb^kYc^k$ we use the rule $A \to ab$, we obtain

$$a^kAb^kYc^k \Rightarrow a^{k+1}b^{k+1}Yc^k,$$

and the rule $Y \to Cc$ can not be used since C can only be rewritten in the presence of an A. So we must have

$$a^{k+1}b^{k+1}Yc^k \Rightarrow a^{k+1}b^{k+1}c^{k+1}.$$

From these considerations we can see that G generates the non-context-free language $L(G) = \{a^nb^nc^n \mid n \geq 1\}$.

3.3.1 The number of conditional productions

In (Meduna and Švec, 2002) the authors show that an erasing rule of the form $XY \to \varepsilon$ (X and Y being two nonterminals) can be simulated by six conditional productions of a simple semi-conditional grammar, thus, to simulate a grammar in the Geffert normal form (see the first variant presented in Section 3.1), simple semi-conditional grammars of degree $(2, 1)$ with twelve conditional productions are sufficient.

Now we show that any grammar being in the second variant of the Geffert normal form, thus, having only one non-context-free rule of the form $ABC \to \varepsilon$, (see Section 3.1), can be simulated by simple semi-conditional grammars of degree $(2, 1)$ with ten conditional productions.

Theorem 3.3.1. *Every recursively enumerable language can be generated by a simple semi-conditional grammar of degree $(2, 1)$ having ten conditional productions.*

Proof. Let $L \subseteq T^*$ be a recursively enumerable language generated by the grammar

$$G = (N, T, P \cup \{ABC \to \varepsilon\}, S)$$

as above.

Now we construct G', a simple semi-conditional grammar of degree $(2,1)$ as follows. Let

$$G' = (N', T, P', S)$$

where $N' = \{S, S', A, A', B, B', B'', C, C', L, L', R\}$, and

$$\begin{aligned} P' &= \{(X \to \alpha, 0, 0) \mid X \to \alpha \in P\} \\ &\cup \{(A \to LA', 0, L), (B \to B', 0, B'), (C \to C'R, 0, R), \\ &\quad (A' \to \varepsilon, A'B', 0), (C' \to \varepsilon, B'C', 0), (B' \to B'', LB', 0), \\ &\quad (B'' \to \varepsilon, B''R, 0), (L \to L', LR, 0), (R \to \varepsilon, 0, L), \\ &\quad (L' \to \varepsilon, 0, R)\}. \end{aligned}$$

By observing the productions of P', we can see that the terminal words generated by G can also be generated by the simple semi-conditional grammar G'. In the following, we show that G' cannot generate words that are not generated by G.

We will examine the possible derivations of G' starting with S and leading to a terminal word. The first two phases of a derivation by G can be reproduced using the non-conditional rules of P', the rules of the form $(X \to \alpha, 0, 0)$ where $X \to \alpha \in P$. Since the conditional rules do not involve the symbols S and S' neither on the left or right sides, nor in the conditions, if we can apply conditional rules before S and S' both disappeared, then we can apply them in the same way also afterwards. According to this observation, we can assume that the first application of a conditional rule happens when neither S, nor S' is present in the sentential form, that is, when the generated word is of the form

$$zuvw, \text{ where } z, u \in \{A, B\}^*, v \in \{B, C\}^*, w \in T^*.$$

Now we show that the prefix zuv can be deleted by the conditional rules of G' if and only if it can be deleted by the rule $ABC \to \varepsilon$ of G.

By continuing the derivation, at most one application of each of the rules $(A \to LA', 0, L)$, $(B \to B', 0, B')$, or $(C \to C'R, 0, R)$ can follow. If these rules do not produce any of the subwords $A'B'$ or $B'C'$, the derivation cannot continue, so if $x = zu$ and $y = v$, it is sufficient to check derivation paths starting from the strings

1. $x_1LA'B'x_2y_1C'Ry_2w$, where $x = x_1ABx_2$, $y = y_1Cy_2$, or
2. $x_1LA'x_2y_1B'C'Ry_2w$, where $x = x_1Ax_2$, $y = y_1BCy_2$

because A can only occur in x, and C can only occur in y.

Now we show that if we continue the derivation, it either enters a blocking configuration, or after deleting one occurrence of the substring ABC we obtain a string which is either of one of the four types above or a terminal string.

Let us follow the derivations starting with each of these strings. We first assume that the substring $LA'B'C'R$ is not present in any of the above cases, that is, $x_2y_1 \neq \varepsilon$.

(1) From the first sentential form we obtain either

- $x_1'B'x_1''LB''x_2y_1C'Ry_2w$, where $x_1'Bx_1'' = x_1$,
- $x_1LB''\alpha'B'\alpha''C'Ry_2w$, where $\alpha'B\alpha'' = x_2y_1$, $\alpha'' \neq \varepsilon$,
- $x_1LB''x_2y_1'B'Ry_2w$, where $y_1'B = y_1$, or
- $x_1LB''x_2y_1C'Ry_2'B'y_2''w$, where $y_2'By_2'' = y_2$,

(2) from the second sentential form we obtain

- $x_1LA'x_2y_1B'Ry_2w$.

The derivation cannot continue from any of these sentential forms, thus, we need to have a string of the following form

$$zuvw = xyw = x_1LA'B'C'Ry_2w,$$

where $x = zu \in \{A, B\}^*$, $y = v \in \{B, C\}^*$, and moreover, $x_1AB = x$ and $Cy_2 = y$, or $x_1A = x$ and $BCy_2 = y$. In two derivation steps we might obtain the following two strings:

$$x_1LA'B'C'Ry_2w \Rightarrow^2 x_1LB''C'Ry_2w, \text{ or } x_1LA'B'C'Ry_2w \Rightarrow^2 x_1LB'Ry_2w.$$

The derivation from the first string cannot be continued, so let us consider the second possibility, and follow each derivation path starting with this string.

First the rule $(B' \to B'', LB', 0)$ must be used producing $x_1LB''Ry_2w$. Now observe that independently of the substring $LB''R$, there is the possibility of rewriting one B to B' in x_1 or in y_2, so let us denote by $\bar{x}_1$ and $\bar{y}_2$ the strings with $g(\bar{x}_1\bar{y}_2) = x_1y_2$ and $|\bar{x}_1\bar{y}_2|_{B'} \leq 1$, where $g(B') = B$ and $g(X) = X$ for all $X \in N' \cup T$, $X \neq B$. Then the possible derivations are the following:
$\bar{x}_1LB''R\bar{y}_2w \Rightarrow \bar{x}_1LR\bar{y}_2w \Rightarrow \bar{x}_1L'R\bar{y}_2w \Rightarrow$

1. $\bar{x}_1'LA'\bar{x}_1''L'R\bar{y}_2w$, where $\bar{x}_1'A\bar{x}_1'' = \bar{x}_1$,
2. $\bar{x}_1L'\bar{y}_2w \Rightarrow$
   (a) $\bar{x}_1L'\bar{y}_2'C'R\bar{y}_2''w$, where $\bar{y}_2'C\bar{y}_2'' = \bar{y}_2$,
   (b) $\bar{x}_1'LA'\bar{x}_1''L'\bar{y}_2w$, where $\bar{x}_1'A\bar{x}_1'' = \bar{x}_1$,
   (c) $\bar{x}_1\bar{y}_2w$.

Note that these cases do not distinguish between sentential forms with different $\bar{x}_1$ and $\bar{y}_2$, as long as $g(\bar{x}_1\bar{y}_2) = x_1y_2$.

The derivation cannot be continued from the sentential forms of case (1) and of case (2)(a), so let us consider now the sentential form of case (2)(b). If $C'R$ is introduced before L' is deleted, the derivation is blocked. Otherwise, by erasing L' first, we can obtain a string that either does not contain any of the substrings $A'B'$ or $B'C'$ (in which case the derivation is blocked), or it is of one of the two forms given at the beginning of our reasoning. The same holds for the sentential form of case (2)(c). This word is either terminal, or we can obtain from it a string of one of the two forms above, or the derivation is blocked.

We have seen that the derivations starting with the sentential form $zuvw$, as above, either enter a blocking configuration, or exactly one occurrence of the substring ABC can be deleted by the rules of P'. If we note that P' contains ten conditional productions and that the degree of G' is $(2, 1)$, then the proof is complete. $\square$

Now we continue by investigating simple semi-conditional grammars having a degree different from $(2, 1)$. In the next theorem we show that the number of conditional productions can be decreased further if we allow permitting conditions of length three, that is, grammars of degree $(3, 1)$.

Theorem 3.3.2. *Every recursively enumerable language can be generated by a simple semi-conditional grammar of degree* $(3, 1)$ *having eight conditional productions.*

Proof. Let $L \subseteq T^*$ be a recursively enumerable language generated by the grammar

$$G = (N, T, P \cup \{ABC \to \varepsilon\}, S)$$

in the Geffert normal form.

Now we construct G', a simple semi-conditional grammar of degree $(3, 1)$ as follows. Let

$$G' = (N', T, P', S)$$

where $N' = \{S, S', A, A', A'', B, B', B'', C, C', C''\}$, and

$$\begin{array}{rcl} P' & = & \{(X \to \alpha, 0, 0) \mid X \to \alpha \in P\} \\ & \cup & \{(X \to X', 0, X') \mid X \in \{A, B, C\}\} \\ & \cup & \{(C' \to C'', A'B'C', 0), (A' \to A'', A'B'C'', 0), \\ & & \quad (B' \to B'', A''B'C'', 0), (A'' \to \varepsilon, 0, C''), (C'' \to \varepsilon, 0, B'), \\ & & \quad (B'' \to \varepsilon, 0, 0)\}. \end{array}$$

The first two phases of generating a terminal word with the grammar G can be reproduced by G' using the rules of P', the rules of the form $(X \to \alpha, 0, 0)$, $X \to \alpha \in P$. The third phase, the application of the erasing production $ABC \to \varepsilon$, is simulated by the additional rules. By observing these additional rules, we can see that all words generated by G can also be generated by G'. In the following we show that G' does not generate words that cannot be generated by G.

Let us follow the possible paths of derivation of G' generating a terminal word. The derivations start with S. While the sentential form contains S or S', it is of the form zSw or $zuS'vw$, $z, u, v \in \{A, B, C, A', B', C'\}^*, w \in T^*$, where if $g(X') = X$ for $X \in \{A, B, C\}$ and $g(X) = X$ for all other symbols of $N \cup T$, then $g(zSw)$ or $g(zuS'vw)$ are valid sentential forms of G. Furthermore, zu contains at most one occurrence of A', v contains at most one occurrence of C', and the whole sentential form, or to put it in an other way, zuv contains at most one occurrence of B'. (To see this, note the forbidding conditions on the rules $(X \to X', 0, X'), X \in \{A, B, C\}$.) After the rule $S' \to \varepsilon$ is used, we get a sentential form $zuvw$ with z, u, v, and w as above, and $g(zuvw)$ being a valid sentential form of G.

Now we show that zuv can be erased by G' if and only if $g(zuv)$ can be erased by G. We do this by showing that if we start from a sentential form $zuvw$ containing single occurrences of each primed symbol A', B', C', then in the next at most nine derivation steps, the derivation either enters a blocking configuration, or the three primed symbols formed a substring $A'B'C'$ which is erased, and nothing else is erased. (Thus, the conditional rules of P' really simulate the rule $ABC \to \varepsilon$ of P.)

If we start with a sentential form $zuvw$ containing single occurrences of each primed symbol, then to be able to continue the derivation, these symbols must form a substring $A'B'C'$, so the sentential form must be of the form $z\bar{u}A'B'C'\bar{v}$, where either $u = \bar{u}A'B'$ and $v = C'\bar{v}$, or $u = \bar{u}A'$ and $v = B'C'\bar{v}$.

Until B' does not disappear (or equivalently, until B'' is not introduced), none of the erasing productions can be applied, so after the first use of the production $(B' \to B'', A''B'C'', 0)$ we have a sentential form of one of the following forms:

- $z\bar{u}A''B''C''\bar{v}w$,
- $zu_1A'u_2A''B''C''\bar{v}w$, where $u_1Au_2 = \bar{u}$,
- $z\bar{u}A''B''C''v_1C'v_2w$, where $v_1Cv_2 = \bar{v}$, or
- $zu_1A'u_2A''B''C''v_1C'v_2w$, where $u_1Au_2 = \bar{u}$, $v_1Cv_2 = \bar{v}$.

Now we denote by $xA''B''C''yw$ one of the sentential forms above, and observe all possible derivations.

The first step can be taken in four different ways:
$xA''B''C''yw \Rightarrow$

1. $x_1B'x_2A''B''C''yw \Rightarrow x_1B'x_2A''C''yw$, where $x_1Bx_2 = x$,
2. $xA''B''C''y_1B'y_2w \Rightarrow xA''C''y_1B'y_2w$, where $y_1By_2 = y$,
3. $xA''C''yw$, or
4. $xA''B''yw$.

In cases (1) and (2), the derivation cannot continue because B' is present, so no erasing production can be applied, and because it is impossible to have $A'B'C'$ or $A'B'C''$ as a substring. The derivation paths starting from (3) are as follows:

3. $xA''C''yw \Rightarrow$
   (a) $x_1B'x_2A''C''yw$,

(b) $xA''C''y_1B'y_2w$,

(c) $xA''yw \Rightarrow$

  i. $xyw \Rightarrow$

    A. $x_1B'x_2yw$,

    B. $xy_1B'y_2w$,

  ii. $x_1B'x_2A''yw \Rightarrow x_1B'x_2yw$,

  iii. $xA''y_1B'y_2w \Rightarrow xy_1B'y_2w$,

where $x_1Bx_2 = x$ and $y_1By_2 = y$. In cases (3)(a) and (3)(b), the derivation cannot be continued, in the sentential forms of cases (3)(c)(i)(A), (3)(c)(i)(B), (3)(c)(ii), and (3)(c)(iii), the substring $A''B''C''$ is removed, and they contain at most one occurrence of A', B', and C'.

Let us now consider the derivation paths starting from (4).

4. $xA''B''yw \Rightarrow$

(a) $xA''yw \Rightarrow$

  i. $xyw \Rightarrow$

    A. $x_1B'x_2yw$,

    B. $xy_1B'y_2w$,

  ii. $x_1B'x_2A''yw \Rightarrow x_1B'x_2yw$,

  iii. $xA''y_1B'y_2w \Rightarrow xy_1B'y_2w$,

(b) $xB''yw \Rightarrow$

  i. $xyw \Rightarrow$

    A. $x_1B'x_2yw$,

    B. $xy_1B'y_2w$,

  ii. $x_1B'x_2B''yw \Rightarrow x_1B'x_2yw$,

  iii. $xB''y_1B'y_2w \Rightarrow xy_1B'y_2w$,

(c) $x_1B'x_2A''B''yw \Rightarrow$

  i. $x_1B'x_2B''yw \Rightarrow x_1B'x_2yw$,

  ii. $x_1B'x_2A''yw \Rightarrow x_1B'x_2yw$,

(d) $xA''B''y_1B'y_2w \Rightarrow$

  i. $xB''y_1B'y_2w \Rightarrow xy_1B'y_2w$,

ii. $xA''y_1B'y_2w \Rightarrow xy_1B'y_2w$,

where $x_1Bx_2 = x$, and $y_1By_2 = y$. The substring $A''B''C''$ is erased from all of the strings produced along these paths. These strings contain at most one occurrence of the symbols A', B', and C'.

To summarize the considerations above, we can say that until the disappearance of all double primed symbols, A'', B'', and C'', only the erasing rules and the rule $(B \to B', 0, B')$ can be applied. We can see that the derivation either enters a blocking configuration, or the substring $A'B'C'$, and only this substring, is completely erased, while the resulting sentential form again contains at most one occurrence of each primed symbol.

This means that the additional conditional productions and the production $(B'' \to \varepsilon, 0, 0)$ of P' correctly simulate the application of the erasing rule $ABC \to \varepsilon$. If we note that P' contains eight conditional productions and that the degree of G' is $(3, 1)$, then the proof is complete. $\square$

3.3.2 Remarks

In Theorem 3.3.1 we have improved the result of (Meduna and Švec, 2002) by showing that simple semi-conditional grammars of degree $(2, 1)$ generate any recursively enumerable language with not more than ten conditional productions. This theorem first appeared in (Vaszil, 2005a). Later, the number of necessary conditional productions was reduced further in (Masopust, 2009a), it was shown there that nine conditional productions are sufficient. That is still the best known result at the time of writing of this dissertation.

Concerning grammars of degree $(1, 1)$, it has been known, see (Păun, 1985), that semi-conditional grammars (and thus, also simple semi-conditional grammars) of degree $(1, 1)$ without erasing rules generate only a subclass of context-sensitive languages, but the problem whether simple semi-conditional grammars of degree $(1, 1)$ with erasing rules are able to generate all recursively enumerable languages has been open for a long time, until (Masopust, 2009a) settled the problem by showing that grammars of degree $(1, 1)$ also generate any recursively enumerable language. However, the number of conditional productions can only be bounded if terminal symbols are allowed to appear as context conditions. Thus, if we allow only nonterminals in the context conditions, then the number of conditional productions can only be bounded in the case of a grammar with degree at least $(2, 1)$.

In Theorem 3.3.2 we have also shown that allowing longer words as context conditions may help to reduce the number of conditional productions, namely, simple semi-conditional grammars of degree $(3, 1)$ generate any recursively enumerable language with not more than eight conditional productions. This result still represents the best bound on the necessary number of conditional productions, although in (Okubo, 2009) a construction with eight conditional productions but less nonterminals (nine instead of eleven) is presented.

3.4 Scattered-context grammars - The number of context sensing productions and the number of nonterminals

Scattered context grammars, introduced in (Greibach and Hopcroft, 1969), are parallel rewriting devices having ordered sequences of context-free rewriting rules as productions. The rules of the sequences are applied simultaneously to nonterminals which appear in the sentential form in the same order as the order of the nonterminals on the left-hand sides of the rules in the sequence. Scattered context grammars with erasing rules are able to generate any recursively enumerable language, see for example (Meduna, 1995b). The problem whether propagating scattered context grammars, i.e., grammars without erasing rules, are able to generate all context sensitive languages, is still open.

Definition 3.4.1. A *scattered context grammar* is a 4-tuple $G = (N, T, P, S)$ where N and T are the disjoint alphabets of nonterminals and terminals, respectively, $S \in N$ is the start symbol, and P is a set of productions of the form $p : (X_1, \dots, X_n) \to (\alpha_1, \dots, \alpha_n)$ where $n \geq 1$, $X_i \in N$, $\alpha_i \in (N \cup T)^*$, $1 \leq i \leq n$. We say that p is the label (the productions are uniquely labeled), and $X_i \to \alpha_i$, $1 \leq i \leq n$ is the ith rule of the production. If $n > 1$, then p is called a *context sensing.*

A sentential form $x \in (N \cup T)^+$ directly derives $y \in (N \cup T)^*$ by applying the production $p : (X_1, \dots, X_n) \to (\alpha_1, \dots, \alpha_n) \in P$, denoted by $x \Rightarrow y$, if x can be written as $x = x_1X_1x_2 \dots x_nX_nx_{n+1}$ with $x_i \in (N \cup T)^*$, $1 \leq i \leq n+1$, and $y = x_1\alpha_1x_2 \dots x_n\alpha_nx_{n+1}$. The language generated by a scattered context grammar G is the set $L(G) = \{w \in T^* \mid S \Rightarrow^* w\}$ where $\Rightarrow^*$ denotes the reflexive and transitive closure of $\Rightarrow$.

Example 3.4.1. *Let $G = (\{S, A\}, \{a, b\}, P, S)$ be a scattered context grammar with P being the set of productions*

$$P = \{S \to AA, (A \to aA, A \to aA), (A \to bA, A \to bA), (A \to \varepsilon, A \to \varepsilon)\}.$$

The derivations of G are of the form

$$S \Rightarrow AA \Rightarrow \dots uAuA \Rightarrow \dots \Rightarrow ww,\ u, w \in \{a, b\}^*,$$

thus, the generated language is $L(G) = \{ww \mid w \in \{a, b\}^*\}$.

3.4.1 The simultaneous reduction of the number of context sensing productions and the number of nonterminals

Scattered context grammars have been shown to generate all recursively enumerable languages with four nonterminals in (Meduna, 1997), with three nonterminals in (Meduna, 2000a), and then with two context-sensing productions in (Fernau and Meduna, 2003a). In (Fernau and Meduna, 2003b), a simultaneous reduction of these measures is attempted, it is shown that any recursively enumerable language can be generated by a scattered context grammar having seven nonterminals and five context sensing productions, or six nonterminals and six context sensing productions.

In this section we improve the results of (Fernau and Meduna, 2003b) by showing that any recursively enumerable language can be generated with scattered context grammars having two context sensing productions and five nonterminals.

Theorem 3.4.1. *Every recursively enumerable language can be generated by a scattered context grammar with two context sensing productions and five nonterminals.*

Proof. Let $L \subseteq T^*$ be a recursively enumerable language generated by the grammar

$$G = (N, T, P \cup \{AB \to \varepsilon, CD \to \varepsilon\}, S),$$

in Geffert normal form (see Section 3.1), such that $N = \{S, S'A, B, C, D\}$ and P contains only context-free productions. Let us construct a scattered context grammar

$$G' = (N', T, P', S)$$

where $N' = \{S, S', 0, 1, \$\}$, and

$$\begin{aligned} P' &= \{(S) \to (h(z)Sx) \mid S \to zSx \in P,\ z \in \{A, C\}^*,\ x \in T\} \\ &\cup\ \{(S) \to (S')\} \\ &\cup\ \{(S') \to (h(u)S'h(v)) \mid S' \to uS'v \in P,\ u \in \{A, C\}^*, \\ &\qquad v \in \{B, D\}^*\} \\ &\cup\ \{(S') \to (\$\$), (\$) \to (\varepsilon)\} \\ &\cup\ \{(0, \$, \$, 0) \to (\$, \varepsilon, \varepsilon, \$), (1, \$, \$, 1) \to (\$, \varepsilon, \varepsilon, \$)\}, \end{aligned}$$

and $h : \{A, B, C, D\}^* \to \{0, 1\}^*$ is a morphism defined as $h(A) = h(B) = 101$, $h(C) = h(D) = 1001$.

The idea behind this grammar is to generate encodings of the sentential forms produced by the first two phases of the derivations in G, strings of the form $zuvw$, $zu \in \{A, C\}^*, v \in \{B, D\}^*, w \in T^*$ such that the substring over $\{A, B, C, D\}$ is encoded using the nonterminals 0 and 1 to the string $h(zu)\$\$h(v)w$, and then to simulate the third phase of the derivation by erasing $h(zu)\$\$h(v)$ if and only if $h(zu)$ is equal to the reverse of $h(v)$, that is, if and only if $w \in L$.

By observing the productions it is clear that the first two phases of the derivation of G producing $zuvw$ (z, u, v, w as above) can be simulated by G' producing the encoded string $h(zu)\$\$h(v)w$, and if $h(zu)$ is equal to the reverse of $h(v)$, then $h(zu)\$\$h(v)$ can be erased using the context sensing rules $(0, \$, \$, 0) \to (\$, \varepsilon, \varepsilon, \$)$ and $(1, \$, \$, 1) \to (\$, \varepsilon, \varepsilon, \$)$ which also check the equality of $h(zu)$ and $h(v)^R$, and then the rule $(\$) \to (\varepsilon)$ to erase the markers.

Now we show that G' can only generate terminal strings which an also be generated by G. Derivations of G' start with

$$S \Rightarrow^* h(z)Sw \Rightarrow h(z)S'w \Rightarrow^* h(zu)S'h(v)w \Rightarrow h(zu)\$\$h(v)w$$

where $zu \in \{A, C\}^*,\ v \in \{B, D\}^*,\ w \in T^*$.

Now the derivation can continue with the application of rules to delete \$, or one of the two context sensing erasing rules, $(0, \$, \$, 0) \to (\$, \varepsilon, \varepsilon, \$)$, $(1, \$, \$, 1) \to (\$, \varepsilon, \varepsilon, \$)$. Note that if a \$ marker is deleted when there are other nonterminals present, then the derivation cannot be successfully finished. Note also, that if a nonterminal 0 or 1 is placed between the two \$ markers, then it cannot be eliminated, thus, the derivation cannot be successfully finished. This means that the context sensing erasing productions can only be applied to delete those two occurrences of 0s or 1s which appear directly before and after the \$\$ markers.

By the considerations above, $h(zu)\$\$h(v)$ can only be erased with the context sensing productions if $h(zu)$ is identical to the reverse of $h(v)$, thus, if and only if the string zuv can be erased with the rules $AB \to \varepsilon$ and $CD \to \varepsilon$ of G, that is, if and only if $w \in L$. □

3.4.2 Remarks

In Theorem 3.4.1, we have improved the result of (Fernau and Meduna, 2003b) by showing that scattered context grammars with two context sensing productions and five nonterminals generate all recursively enumerable languages. Our proof used a combination of techniques from (Fernau and Meduna, 2003b) and (Meduna and Švec, 2002). This result is especially interesting since reducing the nonterminal complexity usually increases the necessary amount of "context sensitivity", but two as the number of context sensing productions is the best bound known so far, even conjectured to be optimal in (Fernau and Meduna, 2003a). Theorem 3.4.1 first appeared in (Vaszil, 2005a), and although the area has been investigated in several papers, no better bound is known at the time of writing this dissertation. In (Masopust, 2009a) the number of nonterminals is decreased to four, but this increases the number of context sensing productions to three, while in (Masopust, 2010) a construction is presented with three nonterminals, but in this case the number of context sensing productions is even higher, five.

Concerning the bound only on the number of nonterminals (with an arbitrary number of context sensing productions), we present the optimal result in the next section.

3.4.3 The optimal bound on the number of nonterminals

Now we focus on the descriptional complexity of scattered context grammars with erasing rules, namely, on the number of nonterminal symbols necessary for generating any recursively enumerable language. It is known that scattered context grammars with three nonterminals characterize the class of recursively enumerable languages, see (Meduna, 2000a). It is also shown in (Meduna, 2000b) that scattered context grammars with just one nonterminal are strictly weaker since they are not able to generate all context-sensitive languages even if erasing productions are allowed. The precise characterization of the class of languages generated by these grammars with two nonterminals, however, has been open for a long time. In the following we show how to construct scattered context grammars which are able to generate any recursively enumerable language with two nonterminals.

There are several techniques which can be used for presenting lower bounds on the number of nonterminals of computationally complete string rewriting mechanisms (grammars). Usually they are based on the simulation of some computationally complete device which has some special form or some other property enabling the reduction of the number of necessary nonterminals of the simulating grammar.

In (Masopust, 2009b; Masopust and Meduna, 2009; Meduna, 2000a) the authors limit the number of nonterminals of scattered context grammars by simulating a general phrase structure grammar in the Geffert normal form (see Section 3.1), where the number of nonterminals is bounded by a very small constant. In (Fernau et al., 2007) the number of necessary nonterminals of graph controlled, programmed, and matrix grammars to generate any recursively enumerable language is reduced. This paper demonstrates another common technique which can be used for the reduction of the number of nonterminals. The proofs are based on creating encodings of the terminal strings as numbers, that is, as words over a unary alphabet (containing one nonterminal), and then checking if the word belongs to the generated language by simulating a register machine, a computationally complete Turing machine variant which works with integers stored in counters (registers) instead of words written on tapes (see Chapter 2 for more details). Since register machines are computationally complete with two registers, two nonterminals are sufficient for keeping track of the values of the two integers. For programmed and matrix grammars, the authors use one more nonterminal, a start symbol, to produce an “initial” sentential form (the “starting configuration” of the simulation process). In the case of graph controlled grammars, the two nonterminals are sufficient to obtain computational completeness since it is possible to designate certain productions as “initial” productions (these can only be used in the first derivation step) eliminating this way the need for the additional starting nonterminal.

In the following, we also simulate a Turing machine variant working with integers (a two-counter machine, see Chapter 2 for the definition), but we use the two nonterminals differently: one of them keeps track of the contents of both counters while the other just separates the different parts of the generated sentential form. Moreover, the simulating scattered context grammar is constructed in such a way that there is no need for an additional starting nonterminal.

Theorem 3.4.2. *For any recursively enumerable language $L \subseteq T^*$, there exists a scattered context grammar $G = (N, T, P, S)$ with $|N| = 2$ such that $L(G) = L$.*

Proof. Let $L \subseteq T^*$, and let $M = (T \cup \{Z, B\}, Q, R, q_1, q_2)$ be a two-counter machine such that $L(M) = L$. Let the elements of Q be denoted as $Q = \{q_1, q_2, \ldots, q_{|Q|}\}$ with q_1 and q_2 being the start state and the accepting state, respectively. Without any loss of generality, we assume that for any $s \in Q$ and $x \in T \cup \{B, \varepsilon\}$, the fact that $(s, x, c_1, c_2) \to (q_2, e_1, e_2) \in R$ implies $c_1 = c_2 = Z$ and $e_1 = e_2 = 0$. This means that M enters the accepting state q_2 only with empty counters, and such a transition never changes the counter contents, that is, the accepting configurations of M are of the form

$$(\varepsilon, q_2, 0, 0). \tag{3.5}$$

We construct a scattered context grammar G such that $L(G) = L(M)$. Let $G = (\{S, A\}, T, P, S)$. For the sake of easier readability, we sometimes write a scattered context production $p : (A_1, \ldots, A_n) \to (\alpha_1, \ldots, \alpha_n)$ as $p : (A_1 \to \alpha_1, \ldots, A_n \to \alpha_n)$, or sometimes write some neighboring rules $A_j \to \alpha_j, \ldots, A_{j+k} \to \alpha_{j+k}$ of p as $A_j, \ldots, A_{j+k} \to \alpha_j, \ldots, \alpha_{j+k}$, $1 \le j < j + k \le n$. We might also compress the notation in the case of simultaneous deletions: If a production $p : (A_1 \to \alpha_1, \ldots, A_n \to \alpha_n)$ contains a k-tuple of neighboring deleting rules, that is, for some $1 \le j < j + k \le n$, we have $A_i \to \varepsilon$ for all i with $j \le i \le j + k$, then we may also write $p : (A_1 \to \alpha_1, \ldots, \langle w \rangle \to \langle \varepsilon \rangle, \ldots, A_n \to \alpha_n)$ where $w = A_j A_{j+1} \ldots A_{j+k}$. Moreover, a rule $A_i \to \alpha_i$ of a scattered context production where $A_i = S$ is called an S-rule, if $A_i = A$, then we call it an A-rule.

Now let

$$\begin{aligned} P \;=\; & P' \cup \{r^{(I)} : (S \to SAAAASSAASSAASSASS), \\ & \qquad r^{(F)} : (\langle SAAAAASSAASSAASSASSS \rangle \to \langle \varepsilon \rangle)\}, \end{aligned}$$

where P' is defined below. The two scattered context productions $r^{(I)}$ and $r^{(F)}$ are the initial and the final productions of any derivation. We define P in such a way that a configuration

$$(w_2, q_i, j_1, j_2),\ w_2 \in T^*,\ q_i \in Q,\ j_1, j_2 \in \mathbb{N},$$

of the two-counter machine M corresponds to a sentential form

$$w_1SAAA\underbrace{A\dots A}_{i}SSAA\underbrace{A\dots A}_{j_1}SSAA\underbrace{A\dots A}_{j_2}SSAS\ \underbrace{S\dots S}_{k=i+j_1+j_2} \tag{3.6}$$

of G. It starts with w_1S where $w_1 \in T^*$ is the part of the input string $w = w_1w_2$ which is already read, and then come three groups of As, each followed by two occurrences of the symbol S. The first group of As encodes the state of M while the second and the third groups encode the counter contents. Then, after the symbols AS, a string of k occurrences of symbols S follow where $k \in \mathbb{N}$ is such, that the sentential form contains an equal number of symbols A and S, that is, $k = i + j_1 + j_2$.

The production $r^{(I)}$ introduces the string corresponding to the initial configuration of M, the production $r^{(F)}$, on the other hand, deletes the nonterminals of the string corresponding to the accepting configuration which is of the form (3.5).

The transition changes are simulated by the productions of the set P' which is defined as follows. For each transition $t \in R$ of the two-counter machine M,

$$t : (q_m, x, c_{t,1}, c_{t,2}) \to (q_n, e_{t,1}, e_{t,2}),$$

we have a production of the form

$$\begin{aligned}&(S \to xSAAAA^nS, \langle AAAA^mS\rangle \to \langle\varepsilon\rangle,\\ &r_1^{(t,1)}, \dots, r_{t_1}^{(t,1)}, r_1^{(t,2)}, \dots, r_{t_2}^{(t,2)}, S \to S, A \to A, S \to S, r_1^{(t)}, \dots, r_l^{(t)}).\end{aligned} \tag{3.7}$$

The first rule, and the next $m + 4$ rules of the production rewrite the group of As containing $m + 3$ symbols, corresponding to the current internal state q_m of M, to a group of As containing $n + 3$ symbols, corresponding to the new internal state q_n.

Now, if $c_{t,i} = Z$ for some i, $1 \le i \le 2$, then $t_i = 4$ and

$$r_1^{(t,i)} : S \to SA\delta_{t,i}S,\ \ r_2^{(t,i)} = r_3^{(t,i)} : A \to \varepsilon,\ \ r_4^{(t,i)} : S \to \varepsilon,$$

with $\delta_{t,i} = \begin{cases} AA & \text{if } e_{t,i} = +1, \\ A & \text{if } e_{t,i} = 0. \end{cases}$

These rules make sure that the number of As corresponding to the contents of the ith counter (j_i in the sentential form of (3.6)) is zero, and $r_1^{(t,i)}$ replaces the deleted two As with two or three new ones depending on whether the number stored in the ith counter should be increased or not, that is, whether the value of $e_{t,i}$ is $+1$ or 0.

On the other hand, if $c_{t,i} = B$ for some i, $1 \leq i \leq 2$, then $t_i = 5$ and

$$r_1^{(t,i)} = r_5^{(t,i)} : S \to S, \quad r_2^{(t,i)} = r_3^{(t,i)} : A \to A, \quad r_4^{(t,i)} : A \to \delta_{t,i},$$

with $\delta_{t,i} = \begin{cases} AA & \text{if } e_{t,i} = +1, \\ A & \text{if } e_{t,i} = 0, \\ \varepsilon & \text{if } e_{t,i} = -1. \end{cases}$

In this case, the value stored in the ith counter should be non-zero, therefore, besides leaving the two Ss and the two As unchanged, one A is replaced by rule $r_4^{(t,i)}$ by two, one, or zero As depending again on the value of $e_{t,i}$.

In both of the above cases, the rest of the rules is defined as follows. For any transition t as above, let d_t denote the difference between the sums of the indices of the states and the counter contents before and after the transition t, that is, let $d_t = |\delta_{t,1}\delta_{t,2}|_A - 2 + n - m$. This is the number of symbols A added to the sentential form (or deleted, in the case when d_t is negative) by the already defined rules of the scattered context production simulating transition t. Now, if $d_t \geq 0$, then let $l = 1$ (see (3.7) for the notation), and

$$r_1^{(t)} : S \to SS^{d_t}.$$

If $d_t < 0$, then let $l = -d_t$, and

$$r_i^{(t)} : S \to \varepsilon, \text{ for all } 1 \leq i \leq l.$$

These rules make sure that the same number of symbols S are added to, or deleted from the sentential form, as the number of symbols A are added or deleted by the rest of the rules of the scattered context production. Thus, the relationship of the number of symbols S and A in the sentential form remains the same after the application of the scattered context production simulating the transition t as it was before the application of the production.

For an example of the above described construction, see Figure 3.1.

Now we show that the grammar G generates the language $L(M)$ accepted by the two-counter machine M.

$$\begin{gathered}(S \to aSAAAA^1S,\ \langle AAAA^3S\rangle \to \langle\varepsilon\rangle,\\ S \to SAAAS, \langle AAS\rangle \to \langle\varepsilon\rangle,\ S \to S, A \to A, A \to A, A \to \varepsilon, S \to S,\\ S \to S, A \to A, S \to S, \langle S^2\rangle \to \langle\varepsilon\rangle)\end{gathered}$$

(a) The scattered context production corresponding to the transition $t : (a, q_3, Z, B) \to (q_1, +1, -1)$ of the two-counter machine M. In this case, $m = 3$, $n = 1$, $t_1 = 4$, $\delta_{t_1} = AA$, $t_2 = 5$, $\delta_{t,2} = \varepsilon$, $d_t = -2$, and $l = 2$.

$$w_1SAAA\underbrace{AAA}_{3}SSAASSAA\underbrace{A\dots A}_{18}SSAS\underbrace{S\dots S}_{18+3=21} \Longrightarrow$$

$$w_1aSAAA\underbrace{A}_{1}SSAA\underbrace{A}_{1}SSAA\underbrace{A\dots A}_{17}SSAS\underbrace{S\dots S}_{21-2=19}$$

(b) The application of the above production to a sentential form corresponding to the configuration $(aw_2, q_3, 0, 18)$ results in a string corresponding to the configuration $(w_2, q_1, 1, 17)$.

Figure 3.1: An example for the construction in the proof of Theorem 3.4.2.

Starting with the initial nonterminal S, the only applicable production of G is $r^{(I)}$ resulting in the sentential form $SAAAASSAASSAASSASS$ which is of the form (3.6) and, as we mentioned before, corresponds to the initial configuration of M.

1. Note first that all productions from $P - \{r^{(I)}\}$ are defined in such a way that they insert or delete an equal number of symbols S and A, which means that if $|\alpha|_A = |\alpha|_S$ and $\alpha \Rightarrow \alpha'$ by a production other than $r^{(I)}$, then $|\alpha'|_A = |\alpha'|_S$ also holds. Since the application of the production $r^{(I)}$ adds eight S and nine A symbols to any sentential form, it follows that $r^{(I)}$ can only be applied once, in the first rewriting step of any derivation (to the sentential form S), otherwise the number of Ss and As will be different, and since $r^{(F)}$ also erases an equal number of symbols A and S, no terminal word can be obtained.

Consider now a sentential form $w\alpha$, $w \in T^*$, $\alpha \in \{S, A\}^*$ of the form (3.6) corresponding to some configuration of M. Now we show that any application of a production from $P - \{r^{(I)}\}$ during a successful derivation of G leads either to a terminal word, or to a sentential form of the same form corresponding to some other configuration of M.

2. Let $p \in P - \{r^{(I)}\}$ be a production applicable to $w\alpha$. Note first that the seven leftmost S symbols occurring in α must be rewritten by the seven leftmost S-rules of p. This follows from the fact that all productions have a rule with A on its lefthand side after the rules rewriting the seven leftmost S symbols, but no sentential form of type (3.6) has any occurrences of A to the right of the eighth S. This means that the seven leftmost S-rules in a production are either applied to the seven leftmost occurrences of S in α, or the last (rightmost) A-rule of p is not be applicable to any symbol of α.

3. Let us assume now that $\alpha = SAAAA^iSSAAA^{j_1}SSAAA^{j_2}SSASS^k$ where $k = i+j_1+j_2$, and let $left(p) = SAAAA^{\bar{i}}SSAAA^{\bar{j}_1}SSAAA^{\bar{j}_2}SSASS^{\bar{k}}$ be the word composed of the concatenation of the symbols on the lefthand sides of the rules of a production p.

3.a. First we argue that the applicability of p to α implies that $\bar{i} \leq i$, $\bar{j}_1 \leq j_1$, and $\bar{j}_2 \leq j_2$. To see this, consider that if $\bar{i} > i$, then the second and the third S-rules cannot be applied to the second and third occurrences of the symbol S in α, thus, according to point 2 above, production p is not applicable. The same holds if $\bar{j}_1 > j_1$ (or $\bar{j}_2 > j_2$). In this case, the fourth and the fifth S-rules (or the sixth and seventh S rules, respectively) cannot be applied to the fourth and fifth (or the sixth and seventh) occurrences of S in α, which means that the production p is not applicable.

3.b. We now show that $\bar{j}_1 \leq j_1$, $\bar{j}_2 \leq j_2$, and $\bar{i} = i$. If we assume that $\bar{j}_1 \leq j_1$, $\bar{j}_2 \leq j_2$, and $\bar{i} < i$, then by applying the production p to $\alpha = SAAAA^iSSAAA^{j_1}SSAAA^{j_2}SSASS^k$, we obtain a sentential form $\alpha' = w_1'SA\ldots ASA^{i-\bar{i}}SAA\ldots SSAA\ldots SSAS\ldots S$, that is, we have a substring $A^{i-\bar{i}}$ between the second and third occurrences of S in α'. Since there is no production in P which contains an A-rule between the second and third S-rules, then according to point 2, there is no way that this string of A symbols can be rewritten or erased from the sentential form. Therefore, after the application of such a production, G cannot derive a terminal word any more.

This means that having a sentential form $w_1\alpha$ with $w_1 \in T^*$, α as above, and a configuration $C = (aw_2, q_i, j_1, j_2)$ of M corresponding to this sentential form, the following holds: If there exists a transition $t : (q_{\bar{i}}, a, c_1, c_2) \to (q_l, e_1, e_2) \in R$, then the corresponding production $p : (S \to aSAAAA^lS, \langle AAAA^{\bar{i}}S\rangle \to \langle\varepsilon\rangle, \ldots) \in P$ is only applicable to $w_1\alpha$ if $\bar{i} = i$, that is, if the state on the lefthand side of transition t is the same as the state in the configuration C of M corresponding to the sentential form $w_1\alpha$ of G.

4. Consider now the sentential form $w_1\alpha$ corresponding to the configuration $C = (aw_2, q_i, j_1, j_2)$, the transition $t = (q_i, a, c_1, c_2) \to (q_l, e_1, e_2)$, as above, and the production p corresponding to t, being of the form

$$(S \to aSAAAA^lS, \langle AAAA^iS\rangle \to \langle\varepsilon\rangle, r_1^{(t,1)}, \ldots, r_{t_1}^{(t,1)}, r_1^{(t,2)}, \ldots, r_{t_2}^{(t,2)}, \\ S \to S, \ldots).$$

We now show that p can only be applied to $w_1\alpha$ in a successful derivation if the following holds: $c_m = Z$, $m \in \{1, 2\}$, if and only if $j_m = 0$; and $c_m = B$, $m \in \{1, 2\}$, if and only if $j_m > 0$.

4.a. Assume first, that $t : (q_i, a, c_1, c_2) \to (q_l, e_1, e_2)$ is such that $c_m = Z$ for some $m \in \{1, 2\}$. In this case, $t_m = 4$, $r_1^{(t,m)} : S \to SA\delta_{t,m}S$, and the neighboring productions $r_2^{(t,m)}$, $r_3^{(t,m)}$, $r_4^{(t,m)}$ can be written as $\langle AAS\rangle \to \langle\varepsilon\rangle$. Thus, if there are more than two As after the third (if $m = 1$) or the fifth (if $m = 2$) occurrences of symbols S in α, then the application of p results in a word with the substrings A^{j_1} (or A^{j_2}) between the fourth and fifth (or the sixth and seventh) occurrences of the symbol S. Then, due to a similar reasoning as in point 3.b, this substring of As can never be removed from the sentential form, and G can never produce a terminal word this way. This means that if $c_m = Z$, $m \in \{1.2\}$, then the production $p \in P$ corresponding to transition $t \in R$ can only be applied to a sentential form of G, if it corresponds to a configuration $C = (aw_2, q_i, j_1, j_2)$ of M with $j_m = 0$, that is, if t is also applicable in C.

4.b. Let us assume now, that $t : (q_i, a, c_1, c_2) \to (q_l, e_1, e_2)$ is such that $c_m = B$ for some $m \in \{1, 2\}$. In this case, $t_m = 5$, and the productions $r_j^{(t,m)}$, $1 \le j \le 5$, can be written as $S, A, A, A, S \to S, A, A, \delta_{t,m}, S$. Thus, if there are less than three As after the third (if $m = 1$) or the fifth (if $m = 2$) occurrence of the symbol S in α, then, according to the reasoning in point 2, the production p is not applicable to α. This means that if $c_m = B$, $m \in \{1, 2\}$, then the production $p \in P$ corresponding to transition $t \in R$ can only be applied to a sentential form of G which corresponds to such a configuration $C = (aw_2, q_i, j_1, j_2)$ of M, where $j_m > 0$, that is, if t is also applicable in C.

From these considerations we can see that starting from a sentential form $w\alpha$ corresponding to a configuration C of M as in (3.6), by applying a production of G we either get a sentential form $wa\alpha'$ corresponding to a configuration C' with $C \vdash_a C'$, or there are no terminal words which can be derived from $wa\alpha'$. Therefore, since the first derivation step of G introduces the sentential form $SAAAASSAASSAASSASS$ corresponding to an initial configuration $(w, q_1, 0, 0)$ of M, the word w can be generated by G if and only if it can be accepted by the two counter machine M. This concludes our proof. □

3.4.4 Remarks

We have shown that scattered context grammars with only two nonterminals are sufficient to generate any recursively enumerable language. This result first appeared in (Csuhaj-Varjú and Vaszil, 2010b). Our proof has been based on the simulation of a two-counter machine and the possibility of the elimination of a separate initial nonterminal in the simulating grammar. The initial nonterminal can be eliminated by preventing the repeated application of the initial production through maintaining the property of having an equal number of symbols A and S in the sentential forms during any terminating derivation. If the initial rule is used more than once, then the number of As becomes larger than the number of Ss and no terminal string can be produced. A similar idea already appeared in (Masopust and Meduna, 2009).

To prevent the repeated application of the initial production is a common problem in the construction of grammars with a limited number of nonterminals, see for example the constructions involving programmed and matrix grammars in (Fernau et al., 2007). The direct application of our technique, however, is not possible in these rewriting mechanisms because it also takes advantage of the fact that the rules of scattered productions have to be applied to nonterminals appearing in the same order as the rules themselves. This property makes it possible that three different integers (corresponding to the state and the counter contents of the two-counter machine) are encoded using two nonterminals: one for storing the actual values and another one for separating the regions corresponding to the different types of information. Since the number of occurrences of separator symbols carries no information about the state of the simulated two-counter machine, it can be used to "balance" the number of occurrences of the other nonterminal. In rewriting mechanisms which do not take into account the order of appearance of the rewritten nonterminals in the sentential forms, however, different integers are usually encoded by using different nonterminals for each of them, thus, the addition of a "balancing" symbol would increase the number of used nonterminals in the same way as the addition of a separate start symbol.

Our result for scattered context grammars is optimal from the point of view of the number of nonterminals, but the number of productions or the number of simultaneously rewritten nonterminals is not bounded in our construction: they depend on the structure of the language generated, or more precisely, on the structure of the two-counter machine which is simulated by the grammar.

The possibility of bounding the maximal number of rules in the scattered context productions is shown for grammars with three nonterminals in (Masopust and Meduna, 2009), just as the possibility of bounding the number of non-context-free productions (productions containing two or more rules to be applied simultaneously) for scattered context grammars with four nonterminals is shown in (Masopust, 2009b).

These measures can also be bounded in our case if we consider the simulation of a universal two-counter machine. Such a universal machine accepts different languages (any recursively enumerable language) over a given alphabet with the same state and transition sets if, instead of being started with empty counters, one of the counters is initialized with a value corresponding to the language in question. Thus, if we construct a grammar based on a universal machine U, then, for any recursively enumerable language L, we can change the initial production $r^{(I)}$ in such a way that it introduces a sentential form corresponding to the starting configuration of U where the counters are also initialized with the values necessary to accept L. This way we can obtain grammars for any language having the same number of productions, and the same number of rules in those productions. (In fact, with the exception of the production $r^{(I)}$, the grammars are identical.)

Chapter 4

Networks of Cooperating Grammars - The Number of Components and Clusters

Parallel communicating grammar systems have been the objects of a detailed study, which is confirmed by the considerable number of publications in the area. The investigations mainly concentrated on their power and on examining how this power is influenced by changes in their basic characteristics: the way of communication, the synchronization among the components, their way of functioning.

In this chapter we first examine the relationship between returning and non-returning systems, then focus our attention on the problem of reducing the number of components of non-returning parallel communicating grammar systems.

4.1 Preliminaries

First we recall the notion of parallel communicating grammar systems from (Păun and Sântean, 1989), for more detailed information see the monograph (Csuhaj-Varjú et al., 1994) or the handbook chapter (Dassow et al., 1997b).

Definition 4.1.1. A *parallel communicating grammar system* with n components, where $n \geq 1$, (a PC grammar system, in short), is an $n+3$-tuple $\Gamma = (N,\ K,\ T,\ G_1, \ldots, G_n)$, where N is a nonterminal alphabet, T is a terminal alphabet and $K = \{Q_1,\ Q_2, \ldots, Q_n\}$ is an alphabet of *query symbols.*

$N,\ T$, and K are pairwise disjoint sets, $G_i = (N \cup K,\ T,\ P_i,\ S_i),\ 1 \le i \le n$, called a *component* of Γ, is a (usually context-free) generative grammar with nonterminal alphabet $N \cup K$, terminal alphabet T, a set of rewriting rules P_i and an axiom or (a start symbol) S_i. G_1 is said to be the *master* (grammar) of Γ.

An n-tuple $(x_1, \ldots, x_n)$, where $x_i \in (N \cup T \cup K)^*,\ 1 \le i \le n$, is called a *configuration* of Γ. $(S_1, \ldots, S_n)$ is said to be the *initial configuration.* PC grammar systems change their configurations by performing direct derivation steps.

Definition 4.1.2. Let $\Gamma = (N,\ K,\ T, G_1, \ldots, G_n)$, $n \ge 1$, be a PC grammar system and let $(x_1, \ldots, x_n)$ and $(y_1, \ldots, y_n)$ be two configurations of Γ. We say that $(x_1, \ldots, x_n)$ *directly derives* $(y_1, \ldots, y_n)$, denoted by $(x_1, \ldots, x_n) \Rightarrow (y_1, \ldots, y_n)$, if one of the next two cases hold:

1. There is no x_i which contains any query symbol, that is, $x_i \in (N \cup T)^*$ for $1 \le i \le n$. In this case $x_i \Rightarrow_{G_i} y_i$. (For $x_i \in T^*$ we have $x_i = y_i$.)

2. There is some $x_i,\ 1 \le i \le n$, which contains at least one occurrence of query symbols. Let x_i be of the form $x_i = z_1 Q_{i_1} z_2 Q_{i_2}, \ldots, z_t Q_{i_t} z_{t+1}$, where $z_j \in (N \cup T)^*, 1 \le j \le t+1$ and $Q_{i_l} \in K,\ 1 \le l \le t$. In this case $y_i = z_1 x_{i_1} z_2 x_{i_2} \ldots z_t x_{i_t} z_{t+1}$ where $x_{i_l},\ 1 \le l \le t$ does not contain any query symbol. Furthermore, (a) in *returning* systems $y_{i_l} = S_{i_l}$, while (b) in *non-returning* systems $y_{i_l} = x_{i_l},\ 1 \le l \le t$.

If some x_{i_l} contains at least one occurrence of query symbols, then $y_i = x_i$. For all $i,\ 1 \le i \le n$, for which y_i is not specified above, $y_i = x_i$.

The first case is the description of a rewriting step: If no query symbols are present in any of the sentential forms, then each component grammar uses one of its rewriting rules except those which have already produced a terminal string. The derivation is blocked if a sentential form is not a terminal string, but no rule can be applied to it.

The second case describes a communication: If some query symbol, say Q_i, appears in a sentential form, then rewriting stops and a communication step must be performed. This means that Q_i must be replaced by the current sentential form of component G_i, say x_i, supposing that x_i does not contain any query symbol. (Only strings without query symbols can be communicated.) If this sentential form also contains some query symbols, then first these symbols must be replaced with the requested sentential forms. If this condition cannot be fulfilled (a circular query appeared), then the derivation is blocked. Let $\Rightarrow_{rew}$ and $\Rightarrow_{com}$ denote a rewriting and a communication step respectively.

If the sentential form of a component was communicated to an other, this component can continue its own work in two ways: In so-called *returning* systems, the component must return to its axiom and begin to generate a new string. In *non-returning* systems the components do not return to their axiom, but continue to process the current string.

A system is *centralized* if only the component G_1 is allowed to introduce query symbols, otherwise it is *non-centralized.*

The phrase *communication step* is used to denote the process of satisfying the query symbols, which can be satisfied in "parallel". For example the communication prescribed by $(Q_2,\ Q_3,\ \alpha,\ Q_3)$ takes two communication steps to realize: first we get $(Q_2,\ \alpha,\ S_3,\ \alpha)$, and then $(\alpha,\ S_2,\ S_3,\ \alpha)$. The two consecutive steps together will be referred to as a *communication sequence.*

Let $\Rightarrow^+$ and $\Rightarrow^*$ denote the transitive, and the reflexive, transitive closure of $\Rightarrow$.

Definition 4.1.3. The *language* generated by a PC grammar system Γ is $L(\Gamma) = \{w \in T^* \mid (S_1, \ldots, S_n) \Rightarrow^* (w, \alpha_2, \ldots, \alpha_n)\}$.

Thus, the generated language consists of the terminal strings appearing as sentential forms of the master grammar, G_1.

Notation. Let PC$_n$CF and NPC$_n$CF denote the classes of returning and non-returning PC grammar systems with at most n context-free components, $n \geq 1$. The classes of languages generated by such systems are denoted by $\mathcal{L}$(PC$_n$CF) and $\mathcal{L}$(NPC$_n$CF), respectively.

Example 4.1.1. *Let* $\Gamma = (N, K, T, G_1, G_2, G_3)$ *be a parallel communicating system of regular grammars with components* $G_i = (N \cup K, T, P_i, S_i)$, $1 \leq i \leq 3$, *where*

$$\begin{aligned} N &= \{S_1, S_2, S_3\}, \\ K &= \{Q_1, Q_2, Q_3\}, \\ T &= \{a, b, c\}, \end{aligned}$$

and

$$\begin{aligned} P_1 &= \{S_1 \to aS_1, S_1 \to aQ_2, S_3 \to \varepsilon\}, \\ P_2 &= \{S_2 \to bS_2, S_2 \to bQ_3\}, \\ P_3 &= \{S_3 \to cS_3\}. \end{aligned}$$

Starting from the initial configuration (S_1, S_2, S_3), Γ can reach (a^iS_1, b^iS_2, c^iS_3) after i rewriting steps for some $i > 0$. The derivation can only be finished if both G_1 and G_2 introduce their query symbols in the same step, obtaining $(a^{i+1}Q_2, b^{i+1}Q_3, c^{i+1}S_3) \Rightarrow_{com} (a^{i+1}b^{i+1}c^{i+1}S_3, \alpha, \beta)$ where $\alpha = S_2$, $\beta = S_3$ in the case of returning communication, or $\alpha = b^{i+1}c^{i+1}S_3$, $\beta = c^{i+1}S_3$ in the case of non-returning communication. In both cases, the derivation is finished by using $S_3 \to \varepsilon$ in G_1 which produces a generated string of the form $a^{i+1}b^{i+1}c^{i+1}$. Thus, Γ generates

$$L(\Gamma) = \{a^nb^nc^n \mid n \geq 1\}$$

which is a context-sensitive, but not context-free language.

4.2 The number of components of non-returning systems

Non-returning PC grammar systems are simulated in (Dumitrescu, 1996) by returning systems. This simulation uses $4n^2 - 3n + 1$ components, where n is the size (the number of components) of the simulated system, and one rewriting step of the simulated system corresponds to about $2n$ simulating rewriting steps.

Here we will briefly review a different method for the simulation of non-returning context-free PC grammar systems with returning systems which requires $(n + 1)^2$ simulating components in general, but this can be reduced to $2n$ in the best case, if the components of the simulated system satisfy certain properties which we will define later. Furthermore, one rewriting step of the simulated system corresponds to only two simulating rewriting steps.

4.2.1 Returning versus non-returning communication

After a communication is performed, not only the components which send their current strings, but also those components which receive these strings may finish the communication with different sentential forms in the returning or non-returning modes. This is due to the fact, that all query symbols occurring in one string must be rewritten in the same communication step. This requirement makes it possible, that a query symbol Q_i is replaced by some string α in the non-returning mode while in the returning mode it is replaced by S_i, since α may no longer be available when the replacement of Q_i becomes possible.

For example the query (Q_2Q_3, Q_3, a) is satisfied in the non-returning mode with the following two steps: $(Q_2Q_3, Q_3, a) \Rightarrow_{com} (Q_2Q_3, a, a) \Rightarrow_{com} (aa, a, a)$, while in the returning mode it is satisfied with $(Q_2Q_3, Q_3, a) \Rightarrow_{com} (Q_2Q_3, a, S_3) \Rightarrow_{com} (aS_3, S_2, S_3)$, producing a different sentential form in the first component.

There are special cases, when the mode of communication does not make any difference in the resulting sentential forms of those components which only receive strings during a communication (Components like G_1 in the example above.) One of them is the following: If occurrences of only one query symbol can be present in one sentential form at the same time, then all the occurrences of a certain query symbol appearing in any sentential form are replaced in the same communication step. It does not matter whether the component which sends its string returns to the axiom or not, since after the sentential form has been sent, there are no other communication symbols present requesting this same string. In other words, all components send their sentential forms at most once during a communication sequence.

Definition 4.2.1. Consider a PC grammar system $(N,\ K,\ T,\ G_1, \ldots, G_n)$. By the word *query* we refer to a sentential form containing at least one query symbol. A query is satisfied through *communication* replacing the query symbols with the requested sentential forms. This may be done in one or more *communication steps*.

We call a query *homogeneous*, if all query symbols contained in the sentential form request the same string, which means that it is of the form $\alpha_1 Q_i \alpha_2 Q_i \ldots \alpha_{t-1} Q_i \alpha_t$, where $1 \leq i \leq n$ and $2 \leq t$. Otherwise a query is *non-homogeneous*, then it is of the form $\alpha_1 Q_{i_1} \alpha_2 Q_{i_2} \ldots \alpha_{t-1} Q_{i_{t-1}} \alpha_t$ where for all $1 \leq j \leq t$ $1 \leq i_j \leq n$, $3 \leq t$ and at least two query symbols are different, there exists $1 \leq j, k \leq t$ for which $Q_{i_j} \neq Q_{i_k}$.

A *component with non-homogeneous queries* is a component grammar G_i, $1 \leq i \leq n$, which is capable of introducing non-homogeneous queries, it has at least one rule of the form $X \rightarrow \alpha Q_i \beta Q_j \delta$, where $i \neq j$.

Now we show how a non-returning communication sequence on n components can be simulated by a returning communication sequence with $n(n+2)$ components.

Lemma 4.2.1. *Consider the PC grammar system* $\Gamma = (N, K, T, G_1, ..., G_n)$ *and the non-returning communication sequence in* Γ*:*

$$(\alpha_1, .., \alpha_n) \Rightarrow^+_{com} (\beta_1, ..., \beta_n), \text{ where } \alpha_i, \beta_i \in (N \cup T \cup K)^*,\ 1 \le i \le n.$$

Let $\Gamma' = (N', K, T, G'_1, ..., G'_n, G'_{n+1}, ..., G'_{2n}, G'_{1,1}, ..., G'_{1,n}, ..., G'_{n,1}, ..., G'_{n,n})$ *and let*

$$(Q_{n+1}, ..., Q_{2n}, \alpha'_1, ..., \alpha'_n, Q_{n+1}, ..., Q_{2n}, ..., Q_{n+1}, ..., Q_{2n}) \Rightarrow^+_{com}$$
$$(\gamma_1, ..., \gamma_n, S_{n+1}, ..., S_{2n}, \delta_{1,1}, ..., \delta_{1,n}, ..., \delta_{n,1}, ..., \delta_{n,n})$$

be a returning communication sequence in Γ'*, where* $\alpha'_i = \alpha_i$ *if* $\alpha_i \in (N \cup T)^*$*, or* $\alpha'_i = \alpha_{i,1}Q_{i,j_1}\alpha_{i,2}Q_{i,j_2}...\alpha_{i,t}Q_{i,j_t}\alpha_{i,t+1}$ *if* $\alpha_i = \alpha_{i,1}Q_{j_1}\alpha_{i,2}Q_{j_2}...\alpha_{i,t}Q_{j_t}\alpha_{i,t+1}$, $\alpha_{i,k} \in (N \cup T)^*,\ 1 \le k \le t+1, 1 \le j_l \le n,\ 1 \le l \le t$*. Then* $\gamma_i = \beta_i,\ 1 \le i \le n$.

Proof. If $|\ \alpha_i\ |_K = 0$ then $\beta_i = \alpha_i$. In this case $\alpha'_i = \alpha_i,\ \gamma_i = \alpha'_i$, so $\gamma_i = \beta_i$. If $\alpha_i = \alpha_{i,1}Q_{j_1}\alpha_{i,2}Q_{j_2}...\alpha_{i,t}Q_{j_t}\alpha_{i,t+1},\ \alpha_{i,k} \in (N \cup T)^*,\ 1 \le k \le t+1$ then $\beta_i = \alpha_{i,1}\beta_{j_1}\alpha_{i,2}\beta_{j_2}...\alpha_{i,t}\beta_{j_t}\alpha_{i,t+1}$. In this case $\alpha'_i = \alpha_{i,1}Q_{i,j_1}\alpha_{i,2}Q_{i,j_2}...\alpha_{i,t}Q_{i,j_t}\alpha_{i,t+1}$. All queries of α_i are redirected in α'_i to the i-th n-tuple of assistant grammars $G'_{i,1}, ..., G'_{i,n}$. When the sentential form of a component G'_{n+i} is available for communication which means it contains no query symbols, it is sent to G'_i and n copies of it appear as sentential forms of all $G'_{j,i}$ $1 \le j \le n$. These copies are available for further requests by $G'_{n+1}, ..., G'_{2n}$. From these considerations it follows that $\gamma_i = \alpha_{i,1}\beta_{j_1}\alpha_{i,2}\beta_{j_2}...\alpha_{i,t}\beta_{j_t}\alpha_{i,t+1} = \beta_i$. □

Now we show how to decrease the number of components in the simulating returning communication sequence.

Lemma 4.2.2. *Consider the PC grammar system* $\Gamma = (N, K, T, G_1, ..., G_n)$ *and the non-returning communication sequence in* Γ*:*

$$(\alpha_1, .., \alpha_n) \Rightarrow^+_{com} (\beta_1, ..., \beta_n), \text{ where } \alpha_i, \beta_i \in (N \cup T \cup K)^*,\ 1 \le i \le n,$$

and let $i_1, i_2, \ldots, i_k,\ k \le n$ *denote the indices of those sentential forms which introduce non-homogeneous queries.*

Let $\Gamma' = (N', K, T, G'_1, ..., G'_n, G'_{n+1}, ..., G'_{2n}, G'_{i_1,1}, ..., G'_{i_1,n}, ..., G'_{i_k,1}, ..., G'_{i_k,n})$ *and let*

$$(Q_{n+1}, ..., Q_{2n}, \alpha'_1, ..., \alpha'_n, Q_{n+1}, ..., Q_{2n}, ..., Q_{n+1}, ..., Q_{2n}) \Rightarrow^+_{com}$$
$$(\gamma_1, ..., \gamma_n, S_{n+1}, ..., S_{2n}, \delta_{i_1,1}, ..., \delta_{i_1,n}, ..., \delta_{i_k,1}, ..., \delta_{i_k,n})$$

be a returning communication sequence in Γ'*, where* $\alpha'_i = \alpha_i$ *if* $\alpha_i \in (n \cup T)^*$*, or* $\alpha'_i = \alpha_{i,1}Q_{n+j}\alpha_{i,2}Q_{n+j} \ldots \alpha_{i,t}Q_{n+j}\alpha_{i,t+1}$ *if* $\alpha_i = \alpha_{i,1}Q_j\alpha_{i,2}Q_j \ldots \alpha_{i,t}Q_j\alpha_{i,t+1}$

$(i \notin \{i_1, i_2, \ldots, i_k\})$ and $\alpha'_{i_j} = \alpha_{i_j,1}Q_{i_j,l_1}\alpha_{i_j,2}Q_{i_j,l_2}...\alpha_{i_j,t}Q_{i_j,l_t}\alpha_{i_j,t+1}$ if $\alpha_{i_j} = \alpha_{i_j,1}Q_{l_1}\alpha_{i_j,2}Q_{l_2}...\alpha_{i_j,t}Q_{l_t}\alpha_{i_j,t+1}$, $\alpha_{i_j,m} \in (N \cup T)^*, 1 \leq m \leq t+1$, $1 \leq l_r \leq n, 1 \leq r \leq t$ for all $i_j \in \{i_1, i_2, ..., i_k\}, j \leq k$. Then $\gamma_i = \beta_i$, $1 \leq i \leq n$.

Proof. If $i \notin \{i_1, i_2, \ldots, i_k\}$ then there are two possible cases: Either α_i does not contain a query at all or $\alpha_i = \alpha_{i,1}Q_j\alpha_{i,2}Q_j...\alpha_{i,t}Q_j\alpha_{i,t+1}, \alpha_{i,m} \in (N \cup T)^*$, $1 \leq m \leq t+1$, the query of α_i is homogeneous.

The statement now follows from Lemma 4.2.1 and from the fact that in a homogeneous query all query symbols occurring in one sentential form are replaced in one same step. $\square$

Now we recall a statement from (Vaszil, 1998). It shows that based on the above described ideas, non-returning PC grammar systems with n context-free components can be simulated by returning systems with $(n+1)^2$ components but this number can be reduced if at least one of the simulated components is not capable of introducing non-homogeneous queries. If none of them introduces non-homogeneous queries, then $2n$ simulating components are enough.

Theorem 4.2.3. *If $L \in \mathcal{L}(\mathrm{NPC}_n\mathrm{CF})$ and $\Gamma \in$ npcclass generating L, then $L = L(\Gamma')$, where $\Gamma' \in \mathrm{PC}_{\mathrm{n(k+2)+1}}\mathrm{CF}$ and $k \leq n$ is the number of components with non-homogeneous queries in Γ.*

Notice, that $2n$ simulating components are enough also for centralized systems, even if the queries are non-homogeneous. This is easy to see, if we consider that communications in a centralized system can only consist of one communication step, since every requested sentential form is always available (only the master grammar can introduce queries), there is no need to save the intermediate results of communications.

4.2.2 Reducing the number of components of non-returning systems

In (Csuhaj-Varjú and Vaszil, 1999), recursively enumerable languages were generated by returning PC grammar systems with 11 component grammars. The necessary number of components was later improved in (Csuhaj-Varjú et al., 2003) to five which is still the best known bound.

The above results inspired further investigations of the descriptional complexity of returning context-free PC grammar systems. In (Csuhaj-Varjú and

Vaszil, 2002) a trade-off between the number of rules or nonterminals and the number of components is demonstrated: With no bound on the number of components, seven rules and eight nonterminals in each of the component grammars are sufficient to generate any recursively enumerable language, while if the number of rules and nonterminals can be arbitrarily high, then the number of components can be bounded by a constant. Possible restrictions on the size of the rules have also been observed, in (Csuhaj-Varjú and Vaszil, 2001) normal form theorems were presented showing that all languages that can be generated by context-free returning parallel communicating grammar systems can also be generated by systems using rules of the form $X \to \alpha$, where X is a nonterminal and α consists of at most two symbols. In addition, if in a rule $X \to \alpha$, the string α contains a query symbol, then α is the query symbol itself.

The result showing that arbitrary recursively enumerable languages can also be generated by non-returning PC grammar systems first appeared in (Mandache, 2000). The number of components, however, in the construction of (Mandache, 2000) depends on the language to be generated, and in general, it can be arbitrary high. The first construction to bound the number of necessary components in the non-returning case was presented in (Vaszil, 2007) where all recursively enumerable languages were shown to be generated with eight components.

In the following we show that any n-counter machine can be simulated with non-returning PC grammar systems of $n+4$ components. As n-counter machines are known to be computationally complete for $n = 2$, this also implies that six components are sufficient to generate any recursively enumerable language with non-returning systems.

Theorem 4.2.4. *For any n-counter machine M ($n \geq 1$) over an alphabet T, a context-free non-returning PC grammar system Γ with $n+4$ components can be constructed such that the language $L \subseteq T^*$ accepted by M is equal to the language generated by Γ.*

Proof. Let $M = (T \cup \{Z, B\}, E, R, q_0, q_F)$ be an n-counter machine, and let $L \subseteq T^*$ be the language accepted by M. Without any loss of generality we may assume that M always enters the final state with empty counters and leaves them unchanged, i. e., for any $q \in E$ with $(q, x, c_1, c_2, \ldots, c_n) \to (q_F, e_1, e_2, \ldots, e_n) \in R$, it holds that $c_i = Z$ and $e_i = 0$, $1 \leq i \leq n$.

To prove the statement, we construct a context-free non-returning PC grammar system Γ having $n+4$ components generating L and simulating the transitions of M. For the sake of easier readability, we will present the simulating construction for $n = 2$, and describe the modifications for the general case afterwards.

Let

$$\Gamma = (N, K, T, G_{sel}, G_{gen}, G_{c_1}, G_{c_2}, G_{ch_1}, G_{ch_2}),$$

where G_{gen} is the master grammar and $G_\gamma = (N \cup K, T, P_\gamma, S)$ is a component grammar for $\gamma \in \{gen, sel, c_1, c_2, ch_1, ch_2\}$.

Let $\mathcal{I} = \{[q, x, c_1, c_2, q', e_1, e_2] \mid (q, x, c_1, c_2) \to (q', e_1, e_2) \in R\}$ and let us introduce for any $\alpha = [q, x, c_1, c_2, q', e_1, e_2] \in \mathcal{I}$ the following notations: $\mathit{State}(\alpha) = q$, $\mathit{Read}(\alpha) = x$, $\mathit{NextState}(\alpha) = q'$, and $\mathit{Store}(\alpha, i) = c_i$, $\mathit{Action}(\alpha, i) = e_i$, where $i = 1, 2$. For technical reasons, we also define $\mathcal{I}' \subseteq \mathcal{I}$, the set of symbols corresponding to transitions which require the zero check of both counters simultaneously, that is, $\mathcal{I}' = \{[q, x, Z, Z, q', e_1, e_2] \mid (q, x, Z, Z) \to (q', e_1, e_2) \in R\}$.

The simulation is based on representing the states and the transitions of M with nonterminals from $\mathcal{I}$ and the values of the counters by strings of nonterminals containing as many symbols A as the value stored in the given counter. Every component is dedicated to simulating a certain type of activity of the two-counter machine: G_{sel} selects the transition to be simulated, G_{c_i}, where $1 \leq i \leq 2$, simulates the respective counter and the update of its contents, G_{ch_j}, where $1 \leq j \leq 2$, assists the work of G_{c_i}, and G_{gen} generates the word read (and possibly accepted) by M.

Let

$$\begin{aligned} N \;=\; & \{S, A, F, F', F'', F''', C_1, C_2, C_3, M_0, M_1, M_2, M_3, V\} \cup \\ & \{D_{i,\alpha}, E_{i,\alpha}, H_{i,\alpha}, H_{i,\alpha,j}, X_\alpha \mid \alpha \in \mathcal{I}, 1 \leq i \leq 3, 1 \leq j \leq 2\} \cup \mathcal{I} \end{aligned}$$

and let the rules of the components be defined as follows.

Let

$$\begin{aligned} P_{sel} \;=\; & \{S \to \alpha \mid \alpha \in \mathcal{I}, \mathit{State}(\alpha) = q_0\} \cup \\ & \{\alpha \to D_{1,\alpha}, D_{1,\alpha} \to D_{2,\alpha}, D_{2,\alpha} \to D_{3,\alpha} \mid \alpha \in \mathcal{I}\} \cup \\ & \{D_{3,\alpha} \to \beta \mid \alpha, \beta \in \mathcal{I},\ \mathit{NextState}(\alpha) = \mathit{State}(\beta)\} \cup \\ & \{D_{3,\alpha} \to F \mid \alpha \in \mathcal{I},\ \mathit{NextState}(\alpha) = q_F\} \cup \\ & \{F \to V, V \to V\}. \end{aligned}$$

This component selects the transition of the two-counter machine to be simulated. The axiom S is used to initialize the system by introducing one of the symbols from $\mathcal{I}$ denoting an initial transition, i. e., a symbol of the form $[q_0, x, c_1, c_2, q', e_1, e_2]$ where q_0 is the initial state. The other productions are used for changing the transition into the next one to be performed. The appearance of symbol F indicates that the simulation of the last transition has been finished and the rules $F \to V, V \to V$ can be used to continue rewriting until the master component also finishes its work.

Let

$$\begin{aligned} P_{gen} &= \{S \to Q_{sel}, C_1 \to C_2, C_2 \to C_3, C_3 \to Q_{sel}\} \cup \\ &\quad \{\alpha \to xC_1 \mid \alpha \in \mathcal{I}, \mathit{Read}(\alpha) = x\} \cup \\ &\quad \{H_{3,\alpha} \to \varepsilon \mid \alpha \in \mathcal{I}\} \cup \{H_{2,\alpha,1} \to \varepsilon \mid \alpha \in \mathcal{I}'\} \cup \\ &\quad \{M_1 \to \varepsilon, F' \to \varepsilon, F'' \to \varepsilon, F \to Q_{ch_1}Q_{c_1}\}. \end{aligned}$$

This component generates the string accepted by the counter machine by adding the symbol $x = \mathit{Read}(\alpha)$ for each $\alpha \in \mathcal{I}$ (chosen by the selector component G_{sel}) using the rule $\alpha \to xC_1$. The productions rewriting C_1 to C_2, C_2 to C_3, and then C_3 to Q_{sel} are used for maintaining the synchronization. The result of the computation is produced after the symbol F appears. Using the rule $F \to Q_{ch_1}Q_{c_1}$, the sentential forms of components G_{ch_1}, G_{c_1}, and G_{c_2} are transferred to G_{gen} (G_{c_2} is queried by G_{ch_1} in the same step) and it makes sure that these strings do not contain any nonterminal letter which is different from $H_{3,\alpha}$, for $\alpha \in \mathcal{I}$, or $H_{2,\alpha,1}$, for $\alpha \in \mathcal{I}'$, or from any of M_1, F', F'', since these are the only symbols which can be erased. (The presence of symbols $H_{3,\alpha}$, or the symbols $H_{2,\alpha,1}$ and $H_{3,\alpha}$ together, depending on $\alpha \in \mathcal{I}$, and M_1 indicate that the simulation of the checks and the updates of the contents of the counters of the two counter machine were correct, F' and F'' are different variants of the symbol denoting the final transition.) If the work of the component stops with a terminal word, then this string was also accepted by M and the simulation was correct.

The following two components are for representing the contents of the counters of M and for simulating the changes in the stored numbers. Let for $i \in \{1, 2\}$,

$$\begin{aligned} P_{c_i} &= \{S \to Q_{sel}, A \to Q_{ch_2}, F \to F', F' \to F'\} \cup \\ &\quad \{\alpha \to X_\alpha, X_\alpha \to y_{i,\alpha,1}Q_{sel}, D_{3,\alpha} \to y_{i,\alpha,2}Q_{sel} \mid \alpha \in \mathcal{I}, \\ &\quad\ \ \mathit{Store}(\alpha, i) = B, y_{i,\alpha,j} = \sigma(\mathit{Action}(\alpha, i), B, j),\ 1 \le j \le 2\} \cup \\ &\quad \{\alpha \to H_{1,\alpha}, H_{1,\alpha} \to H_{2,\alpha,i}, H_{2,\alpha,i} \to H_{3,\alpha}, H_{3,\alpha} \to y_{i,\alpha,1}Q_{sel} \mid \\ &\quad\ \ \alpha \in \mathcal{I}, \mathit{Store}(\alpha, i) = Z, y_{i,\alpha,1} = \sigma(\mathit{Action}(\alpha, i), Z, 1)\}, \end{aligned}$$

where $\sigma : \{1,0,-1\}\times\{B,Z\}\times\{1,2\} \to \{A,\varepsilon\}$ is a partial mapping defined as $\sigma(1,B,1) = A$, $\sigma(1,B,2) = A$, $\sigma(0,B,1) = A$, $\sigma(0,B,2) = \varepsilon$, $\sigma(1,Z,1) = A$, $\sigma(1,Z,2) = \varepsilon$, and $\sigma(-1,B,j) = \sigma(0,Z,j) = \varepsilon$ for $1 \le j \le 2$.

These components are responsible for simulating the change in the contents of the counters, which is represented by a string u consisting of as many letters A as the actual stored number in the counter. By performing rule $A \to Q_{ch_2}$ and the rules $\alpha \to X_\alpha$, $X_\alpha \to y_{i,\alpha,1}Q_{sel}, D_{3,\alpha} \to y_{i,\alpha,2}Q_{sel}$, the components are able to check whether the string representing the respective counter contents contains at least one occurrence of the letter A (if it is required by the transition represented by α), and then modify the contents of the counters in the prescribed manner by introducing the necessary number of new As as $y_{i,\alpha,1}$ and $y_{i,\alpha,2}$. If $Store(\alpha,i) = B$, then the simulation is correct if and only if one occurrence of A is rewritten first and then productions $\alpha \to X_\alpha$, $X_\alpha \to y_{i,\alpha,1}Q_{sel}, D_{3,\alpha} \to y_{i,\alpha,2}Q_{sel}$ are applied in the given order, i. e., after three rewriting steps the new string will contain one more occurrence of M_1. Any other order of rule application results in introducing a letter which cannot be erased from the sentential form anymore ($D_{2,\alpha}$ if no A was rewritten in the first two steps, or M_2 if an A was rewritten in the second step, or M_3 if an A was rewritten in the third step). If $Store(\alpha,i) = Z$, then the rules $\alpha \to H_{1,\alpha}, H_{1,\alpha} \to H_{2,\alpha,i}$, $H_{2,\alpha,i} \to H_{3,\alpha}$, and $H_{3,\alpha} \to y_{i,\alpha,1}Q_{sel}$ are used. The simulation is successful if after applying the productions, $H_{3,\alpha}$ appears in the third step in the new sentential form and it has no occurrence of the symbol A. The absence of A will be checked later by components G_{ch_1} and G_{gen}.

Let

$$\begin{aligned}
P_{ch_1} \;=\; & \{S \to Q_{sel}, \alpha \to E_{1,\alpha}, E_{3,\alpha} \to Q_{sel}, E_{1,\beta} \to E_{2,\beta} \mid \alpha \in \mathcal{I},\\
& \quad \beta \in \mathcal{I} - \mathcal{I}'\} \cup\\
& \{E_{2,\alpha} \to E_{3,\alpha} \mid \alpha \in \mathcal{I}, Store(\alpha,j) = B, 1 \le j \le 2\} \cup\\
& \{E_{2,\alpha} \to Q_{c_2}E_{3,\alpha} \mid \alpha \in \mathcal{I}, Store(\alpha,1) = B, Store(\alpha,2) = Z\} \cup\\
& \{E_{2,\alpha} \to Q_{c_1}E_{3,\alpha} \mid \alpha \in \mathcal{I}, Store(\alpha,1) = Z, Store(\alpha,2) = B\} \cup\\
& \{E_{1,\alpha} \to Q_{c_1}E_{2,\alpha}, E_{2,\alpha} \to Q_{c_2}E_{3,\alpha} \mid \alpha \in \mathcal{I}'\} \cup\\
& \{F \to Q_{c_2}F'', F'' \to F''', F''' \to F'''\}.
\end{aligned}$$

This component assists in checking whether the contents of the respective counter is zero if it is required by the transition to be performed. This is done by requesting the string of the component G_{c_1} and/or G_{c_2}. If the string (or strings) communicated to this component contains (contain) an occurrence of A, then this letter will never be removed since P_{ch_1} has no rule for deleting A and the component G_{gen} which will later issue a query to G_{ch_1}, has no erasing rule for A either. This means that the simulation is correct if the string or strings communicated to G_{ch_1} are free from A, but contains (contain) an occurrence of $H_{3,\alpha}$, if $\alpha \in \mathcal{I} - \mathcal{I}'$, or both $H_{2,\alpha,1}$ and $H_{3,\alpha}$, if $\alpha \in \mathcal{I}'$.

Finally, let

$$P_{ch_2} \;=\; \{S \to M_0, M_0 \to M_1, M_1 \to M_2, M_2 \to M_3, M_3 \to M_0\}.$$

This component assists G_{c_1} and G_{c_2} in checking whether or not the string representing the counter contents contains an occurrence of A. The simulated counter stores a non-negative integer and the simulation is correct if and only if P_{ch_2} is queried in a step when the symbol M_1 is communicated to the respective component G_{c_1} or G_{c_2}.

In what follows, we discuss the work of Γ in details. After the first rewriting step, we obtain a configuration $(S, S, S, S, S, S) \Rightarrow (\alpha_0, Q_{sel}, Q_{sel}, Q_{sel}, Q_{sel}, M_0) \Rightarrow (\alpha_0, \alpha_0, \alpha_0, \alpha_0, \alpha_0, M_0)$ where α_0 is a nonterminal denoting one of the initial transitions of the two-counter machine, i. e., $State(\alpha_0) = q_0$. Notice that since both counters store zero at the beginning, the sentential forms of components G_{c_1} and G_{c_2} do not contain any occurrence of A.

Now we demonstrate how the simulation works. Depending on c_1 and c_2 in $\alpha = [q, x, c_1, c_2, q', e_1, e_2]$, we discuss the cases separately.

Let $\alpha = [q, x, B, Z, q', e_1, e_2] \in \mathcal{I}$, where $x \in T \cup \{\varepsilon\}$, $q, q' \in E$, and we do not specify e_1, e_2 at this moment. Furthermore, let $\beta \in \mathcal{I}$ with $NextState(\alpha) = State(\beta)$. Suppose that up to transition α the simulation of the work of M was correct. Then the configuration of Γ is of the form $(\alpha, w\alpha, u\alpha, v\alpha, \bar{w}\alpha, M_0)$ where $w \in T^*$, $u, v \in \{A, M_1\}^*$, and $\bar{w} \in (\{M_1\} \cup \{H_{3,\alpha} \mid \alpha \in \mathcal{I}\} \cup \{H_{2,\alpha,1} \mid \alpha \in \mathcal{I}'\})^*$.

By the next rewriting step, α changes to $D_{1,\alpha}$ at the first component, P_{sel}, then by the second rewriting step to $D_{2,\alpha}$, and by the third rewriting step to $D_{3,\alpha}$. Similarly, $w\alpha$ at P_{gen} changes to wxC_1, then to wxC_2, and finally to wxC_3, where $x = Read(\alpha)$.

Let us examine now $u\alpha$ which represents the contents of the first counter. Since, by the requirements of the simulated transition, the counter must store a non-negative integer, u should have at least one occurrence of A. If this is not the case, then the only rule which can be applied is $\alpha \to X_\alpha$, and then in the second step $X_\alpha \to y_{1,\alpha,1}Q_{sel}$, which introduces $D_{2,\alpha}$ in the string. Then there are two cases: If $y_{1,\alpha,1} = \varepsilon$, then the derivation gets blocked since there is no rule for rewriting $D_{2,\alpha}$. If $y_{1,\alpha,1} = A$, then the derivation can continue, but it will not produce any terminal word, since this sentential form as a subword of some string will be sent to the master component in a latter phase of the derivation where P_{gen} should remove all nonterminals, but the rules of P_{gen} cannot erase $D_{2,\alpha}$.

If we suppose that u has at least one occurrence of A, then after three rewriting steps performed on $u\alpha$ and the communication following them, the next cases may hold: The new string contains M_1 and $D_{3,\alpha}$ (first an occurrence of A, then α, and then X_α was rewritten) which corresponds to the correct simulation. If an occurrence of A is rewritten in the second and/or in the third step, then M_2 and/or M_3 are introduced and these symbols cannot be erased in the further derivation steps from the string sent at the end of the derivation to the master component, G_{gen}.

Therefore, after the fourth rewriting step from $u\alpha$, we must have a string of the form $u_1M_1u_2y_{1,\alpha,1}y_{1,\alpha,2}Q_{sel}$ where $u = u_1Au_2$ and $y_{1,\alpha,j}$, $1 \leq j \leq 2$ corresponds to $e_1 = \mathit{Action}(\alpha, 1)$ for $\alpha = [q, x, B, Z, q', e_1, e_2]$ as follows: Since one A was removed from u, if $e_1 = -1$ then $y_{1,\alpha,1} = y_{1,\alpha,2} = \varepsilon$, if $e_1 = 0$ then $y_{1,\alpha,1} = A$ and $y_{1,\alpha,2} = \varepsilon$, if $e_1 = +1$ then $y_{1,\alpha,1} = y_{1,\alpha,2} = A$.

Let us consider now $v\alpha$, i. e., the string representing the contents of the second counter. In this case v must not have an appearance of A (according to $\alpha = [q, x, B, Z, q', e_1, e_2]$). If this is the case, that is, if $|v|_A = 0$, then the only rule which can be applied is $\alpha \to H_{1,\alpha}$, and the derivation continues with applying $H_{1,\alpha} \to H_{2,\alpha,i}$ and $H_{2,\alpha,i} \to H_{3,\alpha}$. After the third rewriting step the new string will be of the form $vH_{3,\alpha}$ which will be forwarded by request to component G_{ch_1} and stored there until the end of the derivation when it is sent to the master component G_{gen}. The grammar G_{gen} is not able to erase the nonterminal A, thus, terminal words can only be generated if G_{ch_1} received a string representing an empty counter.

Assume that v contains at least one copy of A. Then, after three rewriting steps the following cases may appear: No occurrence of A is rewritten, thus the obtained string, $vH_{3,\alpha}$, contains at least one A. In this case no terminal word can be generated since, as we discussed before, the string is sent to G_{ch_1} and from now on A cannot be removed in the following phases of the derivation. If at least one occurrence of A is rewritten, then at least one of the symbols M_2, M_3, or $H_{2,\alpha,2}$ is introduced in the string. These letters can never be removed, since the sentential form is forwarded to G_{ch_1} and stored there until the end of the derivation when it is sent to the master component G_{gen}, but neither G_{ch_1}, nor G_{gen} have rules for erasing these symbols. Notice that $H_{2,\alpha,1}$ for $\alpha \in \mathcal{I}'$ can be deleted at G_{gen}, but no other symbol $H_{2,\alpha,j}$ for $j = 1, 2$ and $\alpha \in (\mathcal{I} - \mathcal{I}')$ can be erased.

This means that to have a correct simulation, the new string obtained from $v\alpha$ after the fourth rewriting step must be of the form $vy_{2,\alpha,1}Q_{sel}$, where v contains no occurrence of A and $y_{2,\alpha,1}$ is the string corresponding to $e_2 = Action(\alpha, 2)$. Since, in the case of a correct simulation, no A was deleted, $y_{2,\alpha,1} = \varepsilon$ if $e_2 = 0$, and $y_{2,\alpha,1} = A$ if $e_2 = +1$ (the case $e_2 = -1$ is not applicable, since the counter stores zero, $Store(\alpha, 2) = Z$).

Continuing the derivation, the prescribed communication step results in the configuration $(\beta, wx\beta, u'\beta, v'\beta, \bar{w}'\beta, M_0)$ where $\beta \in \mathcal{I}$ is a transition with $NextState(\alpha) = State(\beta)$, u', v' are strings representing the counters of M following the transition described by $\alpha \in \mathcal{I}$, and $\bar{w}'$ is a string over $\{M_1\} \cup \{H_{3,\alpha} \mid \alpha \in \mathcal{I}\} \cup \{H_{2,\alpha,1} \mid \alpha \in \mathcal{I}'\}$. Thus, we obtain a configuration of the form we started from. Now, similarly as above, the simulation of the transition corresponding to the symbol $\beta \in \mathcal{I}$ can be performed.

The reader may immediately notice that the case $\alpha = [q, x, Z, B, q', e_1, e_2]$ can be treated in the same way, with changing the discussion concerning components G_{c_1} and G_{c_2}. If $\alpha = [q, x, B, B, q', e_1, e_2]$, then the proof is also based on analogous considerations: For the case of the component representing the second counter of M, P_{c_2}, we use the same reasoning as we did in the case of P_{c_1} above.

It remains to discuss the case when $\alpha = [q, x, Z, Z, q', e_1, e_2]$. In this case, the string obtained from $u\alpha$ after the second rewriting step and the string obtained from $v\alpha$ after the third rewriting step are forwarded to component G_{ch_1}. The simulation is correct if and only if the strings obtained by G_{ch_1} are of the form $uH_{2,\alpha,1}$ and $vH_{3,\alpha}$ (notice that $\alpha \in \mathcal{I}'$) with no occurrence of A in u or in v. If the strings transferred to G_{ch_1} have at least one occurrence of A, then no terminal word can be derived in Γ by this derivation, based on the arguments given above (G_{gen} has no rule for deleting A).

Suppose now that u has at least one occurrence of A which is rewritten during the three rewriting steps. Then, at least one of the symbols M_2, M_3, or $H_{1,\alpha}$ is introduced in the string which implies an unsuccessful termination, since these symbols will appear in the sentential form at the end of the derivation at component G_{gen} but they cannot be deleted. Analogously, if v contains at least one occurrence of A and it is rewritten in these three rewriting steps, then $H_{3,\alpha}$ cannot be introduced, which also results in a string containing at least one symbol which cannot be deleted in the latter phases of the derivation.

Now we discuss the terminating derivation phase. Suppose now that $NextState(\alpha) = q_F$ and G_{sel} decides to end the simulation of M, that is, the nonterminal $D_{3,\alpha}$ is changed to F. Then the obtained configuration is $(F, wxF, u'F, v'F, \bar{w}'F, M_0)$. Then we get $(V, wxQ_{ch_1}Q_{c_1}, u'F', v'F', \bar{w}'Q_{c_2}F''$, $M_1)$, and after then $(V, wx\bar{w}'v'F'F''u'F', u'F', v'F', \bar{w}'v'F'F'', M_1)$.

Since in the case of a correct simulation $|\bar{w}'|_A = 0$, therefore by applying the erasing rules of P_{gen} to delete $\{H_{3,\alpha} \mid \alpha \in (\mathcal{I}-\mathcal{I}')$, $\{H_{2,\alpha,1}, H_{3,\alpha} \mid \alpha \in \mathcal{I}'\}$, M_1, F', and F'', we either obtain a terminal word $w' = wx$ also accepted by the two-counter machine M, or there are nonterminals in the sentential form of G_{gen} which cannot be deleted. By the explanations above, it can be seen that any word of L can be generated by Γ, and it does not generate any other words, thus Γ characterizes the same language as M accepts.

Now we describe how to modify the above construction for any $n \geq 2$. Let $\Gamma' = (N, K', T, G'_{sel}, G'_{gen}, G'_{c_1}, \ldots, G'_{c_n}, G'_{ch_1}, G'_{ch_2})$, where G'_{gen} is the master grammar and $G'_{\gamma} = (N \cup K', T, P'_{\gamma}, S)$ is a component grammar for $\gamma \in \{gen, sel, c_1, \ldots, c_n, ch_1, ch_2\}$.

Let us define $P'_{sel} = P_{sel}$ and $P'_{ch_2} = P_{ch_2}$ where P_{sel}, P_{ch_2} are the components given above, and let also P'_{c_i} be defined for all i, $1 \leq i \leq n$, based on $Store(\alpha, i)$ and $Action(\alpha, i)$ as above.

The other components are modified as follows. Let

$$\begin{aligned} P'_{gen} = P_{gen} - (\{H_{2,\alpha,1} \to \varepsilon \mid \alpha \in \mathcal{I}'\} \cup \{F \to Q_{ch_1}Q_{c_1}\}) \cup \\ \{H_{3,\alpha} \to \varepsilon \mid \alpha \in \mathcal{I}\} \cup \{F \to Q_{ch_1}Q_{c_1} \ldots Q_{c_n}\}, \end{aligned}$$

and let

$$\begin{aligned} P'_{ch_1} \quad = \quad & \{S \to Q_{sel}, \alpha \to E_{1,\alpha}, E_{1,\alpha} \to E_{2,\alpha}, E_{3,\alpha} \to Q_{sel} \mid \alpha \in \mathcal{I}\} \cup \\ & \{E_{2,\alpha} \to E_{3,\alpha}K_{c_1} \cdots K_{c_n} \mid \alpha \in \mathcal{I}, K_{c_i} = \varepsilon \text{ for } Store(\alpha, i) = B, \\ & \ K_{c_i} = Q_{c_i} \text{ for } Store(\alpha, i) = Z\} \cup \\ & \{F \to F'', F'' \to F''', F''' \to F'''\}. \end{aligned}$$

The rules of P_{ch_1} are modified in P'_{ch_1} in such a way that even if the emptiness of more than one counters is checked, that is, if more then one components from G_{c_i}, $1 \leq i \leq n$, are queried, then all the query symbols are introduced in the same step. This allows simplifications also in P'_{gen} compared to P_{gen} where, in addition, also the rule $F \to Q_{ch_1}Q_{c_1}$ is modified to $F \to Q_{ch_1}Q_{c_1} \ldots Q_{c_n}$. □

Observing the rules of Γ in the proof above, the reader may notice that Γ has no rule with right-hand side longer than two. This fact implies that not only returning but non-returning context-free PC grammar systems preserve the well-known property of context-free grammars, i.e., that binary normal form grammars obtain the generative power of the whole grammar class without any restriction.

Definition 4.2.2. A context-free non-returning parallel communicating grammar system $\Gamma = (N, K, T, G_1, \ldots, G_n)$, where $n \geq 1$, is said to be in *binary normal form*, if any rule in any G_i, $1 \leq i \leq n$, is of the form $A \to u$ where $|u| \leq 2$.

Moreover, as n-counter machines characterize the class of recursively enumerable languages already for $n = 2$, the above theorem implies that non-returning PC grammar systems generate any recursively enumerable language with six components.

Combining these two observations, we obtain the following corollary.

Corollary 4.2.5. *Any recursively enumerable language can be generated by a context-free non-returning parallel communicating grammar system in binary normal form with six components.*

4.2.3 Remarks

In the previous sections we have first examined the differences between the nature of returning and non-returning communication in context-free PC grammar systems, and also have shown how non-returning systems can be directly simulated by returning ones. These results first appeared in (Vaszil, 1998). Next, we have shown that the computation of n-counter machines can be simulated by context-free non-returning PC grammar systems having $n+4$ components, thus, any recursively enumerable language can be generated by these system having six component grammars (even if these are in binary normal form). These results first appeared in (Csuhaj-Varjú and Vaszil, 2010a). In the following we examine how these parameters can be further reduced if we consider a kind of clustering of the components of PC grammar systems.

4.3 Clustering the components, further reduction of size parameters

Answering the natural question, whether or not the clustered organization implies changes in the properties of PC grammar systems, we studied the generative power of non-returning clustered PC grammar systems. In (Csuhaj-Varjú et al., 2011) it was shown that these constructs with only three clusters and seven components are as powerful as Turing machines. Thus, although the number of components does not increase significantly when compared to the non-clustered case, the maximal power can also be obtained with queries to groups of agents instead of queries to precisely identified individuals.

In this section we show that any recursively enumerable language can be generated by a non-returning clustered PC grammar system with four (predefined) clusters and five components, thus, there seems to be a trade-off between the number of components and the number of clusters. Although the start point of our argument is the same as in (Csuhaj-Varjú and Vaszil, 2009) (and (Csuhaj-Varjú et al., 2011)), namely, we simulate two-counter machines, there is no direct tool for distinguishing the members of the clusters during the work of the system, therefore, to select the configurations which correspond steps of the correct simulation extra efforts are needed.

The results on clustered systems also imply the improvement of the bound of the number of components necessary to obtain computational completeness for standard nonreturning PC grammar systems from six to five, to the same bound as needed in the case of returning systems.

In the second part of this section we follow another natural idea, namely, we examine the case when the clusters are not fixed in advance, as in (Csuhaj-Varjú et al., 2011), but formed dynamically in each configuration when query symbols appear. This approach is also motivated by clustered problem solving systems where the clusters are formed according to the (actual) competence levels of the agents. Furthermore, this concept is closely related to the observation that the number of different query symbols which appear in the actual configurations of the system is a significant parameter. If this number is one, then the query may represent a broadcast (any agent who knows the answer may reply), if it is, say k, then k different questions are distributed. Based on these considerations, we introduce the concept of a dynamically clustered PC grammar system, where if the number of query symbols in the actual configuration is k, then at least $k+1$ clusters are formed non-deterministically (at least the querying component should belong to a cluster which is not queried).

We show that dynamically clustered non-returning PC grammar systems are as powerful as Turing machines with six components and three clusters. This means that the use of only two different query symbols is sufficient to generate any recursively enumerable language. It is an open question whether systems with only one query symbol are enough to obtain this computational power.

We also deal with unary languages generated by non-returning clustered PC grammar systems. In this case only the length of the string, no other information on the structure of the word is given. We show that to generate a recursively enumerable language over a unary alphabet, non-returning (clustered) PC grammar systems with four components (and four predefined clusters) are sufficient. If dynamical clustering is considered, then five components and three dynamical clusters, i.e., two query symbols are enough.

Now we present the notion of a parallel communicating grammar system with (predefined) clusters of components (Csuhaj-Varjú et al., 2011), and then we introduce the concept of a dynamically clustered PC grammar system. Note that the original notion of a PC grammar system (see Section 4.1) can be obtained as a special case of the variant with predefined clusters where each component belongs to a different cluster having just one element.

Definition 4.3.1. A *parallel communicating grammar system with* m *(predefined) clusters* and n components (a clustered PC grammar system) is an $(m + n + 3)$-tuple $\Gamma = (N, K, T, G_1, \dots, G_n, \mathcal{C}_1, \dots, \mathcal{C}_m),\ n, m \geq 1$, where N and T is defined as usual, $K = \{Q_1, \dots, Q_m\}$ is the alphabet of *query symbols*, and the components are $G_i = (N \cup K, T, P_i, S_i),\ 1 \leq i \leq n$. The set $\mathcal{C}_j \subseteq \{G_i \mid 1 \leq i \leq n\}$, $1 \leq j \leq m$, is a *cluster* of components. One of the clusters, $\mathcal{C}_k$, $1 \leq k \leq m$, is distinguished and called the *master cluster* of Γ.

An n-tuple $(x_1, \dots, x_n)$, where $x_i \in (N \cup T \cup K)^*,\ 1 \leq i \leq n$, is called a *configuration* of Γ; $(S_1, \dots, S_n)$ is said to be the *initial configuration.*

Definition 4.3.2. We say that $(x_1, \dots, x_n)$ *directly derives* $(y_1, \dots, y_n)$, denoted by $(x_1, \dots, x_n) \Rightarrow (y_1, \dots, y_n)$, if one of the following two cases holds:

1. There is no x_i which contains any query symbol, that is, $x_i \in (N \cup T)^*$ for all $1 \leq i \leq n$. Then, for each i, $1 \leq i \leq n$, $x_i \Rightarrow_{G_i} y_i$ (y_i is obtained from x_i by a direct derivation step in G_i) for $x_i \notin T^*$ and $x_i = y_i$ for $x_i \in T^*$.

2. There is some x_i, $1 \leq i \leq n$, which contains at least one occurrence of a query symbol. Then, for each x_i, $1 \leq i \leq n$, with $|x_i|_K \neq 0$ we write $x_i = z_1 Q_{i_1} z_2 Q_{i_2} \dots z_t Q_{i_t} z_{t+1}$, where $z_j \in (N \cup T)^*$, $1 \leq j \leq t+1$, and $Q_{i_l} \in K$, $1 \leq l \leq t$. If for all l, $1 \leq l \leq t$, and for every sentential form x_k of G_k, $1 \leq k \leq n$, $G_k \in \mathcal{C}_{i_l}$, it holds that $|x_k|_K = 0$, then $y_i = z_1 x_{i_1} z_2 x_{i_2} \dots z_t x_{i_t} z_{t+1}$ where $x_{i_l} \in \{x_k \mid G_k \in \mathcal{C}_{i_l}\}$, that is, any one of the components in the queried cluster is allowed to reply if none of the current sentential forms of the components in the cluster contains a query symbol. Furthermore, (a) in *returning* systems $y_{i_l} = S_{i_l}$, while (b) in *non-returning* systems $y_{i_l} = x_{i_l}$, $1 \leq l \leq t$. If on the other hand, for some l, $1 \leq l \leq t$, there is a $G_k \in \mathcal{C}_{i_l}$, $1 \leq k \leq n$, such that $|x_k|_K \neq 0$ then $y_{i_l} = x_{i_l}$. For all i', $1 \leq i' \leq n$, for which y'_i is not specified above, $y'_i = x'_i$.

Moreover, a clustered PC grammar system works in such a way that the same (non-deterministically chosen) component in $\mathcal{C}_j$ replies to all queries $Q_j \in K$, $1 \leq j \leq m$ appearing in a given configuration, i.e., for $Q_{i_h} = Q_{i_g}$, $1 \leq g, h \leq t$, above, both Q_{i_h} and Q_{i_g} are replaced with x_{i_h}, where $|x_{i_h}|_K = 0$ and x_{i_h} is the sentential form of $G_k \in \mathcal{C}_{i_h}$ for some k, $1 \leq k \leq n$.

For a fixed i, $1 \leq i \leq n$, let the language $L(G_i)$ generated by the component G_i be defined as $L(G_i) = \{x \in T^* \mid (S_1, \dots, S_i, \dots, S_n) \Rightarrow^* (x_1, \dots, x_i, \dots, x_n)$ for some $x_1, \dots, x_n \in (N \cup T \cup K)^*$ such that $x = x_i\}$ where $\Rightarrow^*$ denote the reflexive and transitive closure of $\Rightarrow$. The *language generated* by Γ is $\bigcup_{G_i \in C_j} L(G_i)$ where C_j is the master cluster of Γ for some j, $1 \leq j \leq m$.

Definition 4.3.3. A *parallel communicating grammar system with dynamical clusters* and n components (a dynamically clustered PC grammar system) is an $(n+3)$-tuple $\Gamma = (N, K, T, G_1, \dots, G_n)$, $n \geq 1$, where N, T, and $G_i, 1 \leq i \leq n$ are defined as above, the set of query symbols, however, is $K \subseteq \{Q_1, \dots, Q_n\}$, and instead of a master cluster, a master component is given.

The rewriting steps are also defined in the same way as above, but the communication steps are performed differently. If k different query symbols appear in a configuration, $Q_1, \dots, Q_k$, $k \geq 1$, then the components are grouped into l arbitrary clusters, $\mathcal{C}_1, \dots, \mathcal{C}_l$, in such a way that $l \geq k$, and there is no component in $\mathcal{C}_i$ with Q_i occurring in its sentential form for any $1 \leq i \leq k$. Then each cluster $\mathcal{C}_i$ corresponds to the query symbol Q_i

in the sense that Q_i can be replaced by the sentential forms of one of the components from the cluster $\mathcal{C}_i$.

More formally, we say that a configuration $c = (x_1, \ldots, x_n)$ *directly derives* the configuration $c' = (y_1, \ldots, y_n)$ *by a communication step*, if there is some x_i, $1 \leq i \leq n$, which contains at least one occurrence of a query symbol.

Definition 4.3.4. Let $K_c = \{Q \in K \mid |x_i|_Q \geq 1,\ 1 \leq i \leq n\}$, and let $k = |K_c|$, that is, let k be the number of different query symbols occurring in the configuration c. Let also $\mathcal{C}_j \subseteq \{G_1, \ldots, G_n\}$, $1 \leq j \leq l$ for some $l \geq k$, such that $\bigcup_{1 \leq j \leq l} \mathcal{C}_j = \{G_1, \ldots, G_n\}$ and for $j_1 \neq j_2$, $\mathcal{C}_{j_1} \cap \mathcal{C}_{j_2} = \emptyset$, and moreover, for all $1 \leq s \leq n$ and $G_s \in \mathcal{C}_j$ it holds that $|x_s|_{Q_j} = 0$.

Then, we write $x_i = z_1 Q_{i_1} z_2 Q_{i_2} \ldots z_t Q_{i_t} z_{t+1}$ for each x_i, $1 \leq i \leq n$, with $|x_i|_K \neq 0$ where $z_j \in (N \cup T)^*$, $1 \leq j \leq t+1$, and $Q_{i_m} \in K_c$, $1 \leq m \leq t$. If for all i_m, $1 \leq m \leq t$, and for every sentential form x_s of $G_s \in \mathcal{C}_{i_m}$, $1 \leq s \leq n$, it holds that $|x_s|_K = 0$, then $y_i = z_1 x_{i_1} z_2 x_{i_2} \ldots z_t x_{i_t} z_{t+1}$ where $x_{i_l} \in \{x_s \mid G_s \in \mathcal{C}_{i_m}\}$, that is, each cluster is assigned to the corresponding query symbol and any of the components in the assigned cluster is allowed to reply if none of the current sentential forms of the components of the cluster contains a query symbol.

Furthermore, (a) in *returning* systems $y_{i_m} = S_{i_m}$, while (b) in *non-returning* systems $y_{i_m} = x_{i_m}$, $1 \leq m \leq t$. If on the other hand, for some m, $1 \leq m \leq t$, there is a $G_s \in \mathcal{C}_{i_m}$, $1 \leq s \leq n$, such that $|x_s|_K \neq 0$ then $y_{i_m} = x_{i_m}$. For all i', $1 \leq i' \leq n$, for which y'_i is not specified above, $y'_i = x'_i$. The h (*homogeneous*) communication mode is defined in the same way as in the case of predefined clusters.

Again, the same (non-deterministically chosen) component in $\mathcal{C}_j$ replies to all queries $Q_j \in K$, $1 \leq j \leq n$ appearing in a given configuration, i.e., for $Q_{i_h} = Q_{i_g}$, $1 \leq g, h \leq t$, above, both Q_{i_h} and Q_{i_g} are replaced with x_{i_h}, where $|x_{i_h}|_K = 0$ and x_{i_h} is the sentential form of $G_k \in \mathcal{C}_{i_h}$ for some k, $1 \leq k \leq n$.

Unlike in the case of predefined (static) clustering, we require here that the clusters are disjoint, otherwise any query symbol could always be replaced by any query free sentential form.

The *language generated* by Γ is $L(G_i)$ where G_i is the master component.

Notation. Let the class of languages generated by context-free returning and non-returning PC grammar systems having at most m predefined clusters and n components, $n \geq m \geq 1$, be denoted by $\mathcal{L}(\mathrm{PC}_{m/n}\mathrm{CF})$ and $\mathcal{L}(\mathrm{NPC}_{m/n}\mathrm{CF})$, respectively. If $m = n$ then we put n in the subscript instead of n/n. Let also $\mathcal{L}(X_*\mathrm{CF}) = \bigcup_{i,j\geq 1}\mathcal{L}(X_{i/j}\mathrm{CF})$, $X \in \{\mathrm{PC}, \mathrm{NPC}\}$. If the clusters are formed dynamically, then we use DPC and DNPC instead of PC and NPC, respectively.

Now we show that every recursively enumerable language can be generated by a context-free clustered non-returning PC grammar system with five components and four predefined (static) clusters or six components with three dynamical clusters, that is, with two different query symbols.

Theorem 4.3.1. *$\mathcal{L}(\mathrm{NPC}_{4/5}\mathrm{CF}) = \mathcal{L}(\mathrm{RE})$ and $\mathcal{L}(D\mathrm{NPC}_{3/6}\mathrm{CF}) = \mathcal{L}(\mathrm{RE})$.*

Proof. Let $L \in \mathcal{L}(\mathrm{RE})$ over an alphabet T, and let $M = (T\cup\{Z, B\}, Q, R, q_0, q_F)$ be a two-counter machine with $L(M) = L$.

Without the loss of generality, we may assume for any $(q, x, c_1, c_2) \to (q_F, e_1, e_2) \in R$ that $c_1 = c_2 = Z$ and $e_1 = e_2 = 0$. We also may assume the following: For any transition rule $(q, x, c_1, c_2) \to (r, e_1, e_2) \in R$, if for some $i \in \{1, 2\}$ it holds that $c_i = B$ and $e_i \in \{0, +1\}$, then there exists another transition $(q, x, c'_1, c'_2) \to (r, e_1, e_2) \in R$ such that $c'_i = Z$. To see this consider the following. If there is a transition rule $(q, x, B, c_2) \to (r, e_1, e_2) \in R$ with $e_1 \in \{0, +1\}$, such that there is no $(q, x, Z, c_2) \to (r, e_1, e_2) \in R$, then we can add the new states q', q'' to the state set, modify the rule as $(q, x, B, c_2) \to (q', -1, 0)$, and add the new transition rules $(q', \varepsilon, B, c_2) \to (q'', +1, 0)$, $(q', \varepsilon, Z, c_2) \to (q'', +1, 0)$, $(q'', \varepsilon, B, c_2) \to (r, e_1, e_2)$, $(q'', \varepsilon, Z, c_2) \to (r, e_1, e_2)$ to the set of transitions.

Let us define $\mathcal{I} = \{[q, x, c_1, c_2, q', e_1, e_2] \mid (q, x, c_1, c_2) \to (q', e_1, e_2) \in R\}$, and let us introduce for any $\alpha = [q, x, c_1, c_2, q', e_1, e_2] \in \mathcal{I}$ the following notation: $State(\alpha) = q$, $Read(\alpha) = x$, $NextState(\alpha) = q'$, and $Store(\alpha, i) = c_i$, $Action(\alpha, i) = e_i$, where $i = 1, 2$. We also define for $c', c'' \in \{B, Z\}$ subsets of $\mathcal{I}$ as $\mathcal{I}_{(c',c'')} = \{\alpha \in \mathcal{I} \mid \alpha = [q, x, c', c'', q', e_1, e_2]\}$. We construct clustered NPC grammar systems generating L by simulating the transitions of M.

For the first equality, let $\Gamma = (N, K, T, G_{sel}, G_{c_1}, G_{c_2}, G_{ch}, G_{gen}, \mathcal{C}_1, \ldots, \mathcal{C}_4)$, where $G_\gamma = (N \cup K, T, P_\gamma, S)$, $\gamma \in \{sel, c_1, c_2, gen, ch\}$ is a component grammar, and $\mathcal{C}_4$ is the master cluster.

Let $N = \mathcal{I} \cup \{D_{i,\alpha}, E_{i,\alpha}, H_{i,\alpha}, H_{i,\alpha,j}, X_{i,\alpha,j} \mid \alpha \in \mathcal{I}, 1 \leq i \leq 3, 1 \leq j \leq 2\} \cup \{F_{c,i,j}, F_{gen,j}, F_{ch,j} \mid 1 \leq i \leq 2, 1 \leq j \leq 5\} \cup \{S, A, F, C_1, C_2, C_3, W_1, W_2\}$.

The simulation is based on representing the states and the transitions of M with nonterminals from $\mathcal{I}$ and the values of the counters by strings of nonterminals containing as many symbols A as the value stored in the given counter.

Let the clusters and the rule sets of the components be defined as follows. Let $\mathcal{C}_1 = \{G_{sel}\}$, and let

$$\begin{aligned} P_{sel} &= \{S \to \alpha \mid \alpha \in \mathcal{I}, \mathit{State}(\alpha) = q_0\} \cup \{F \to F_{sel}, F_{sel} \to F_{sel}\} \cup \\ &\quad \{\alpha \to D_{1,\alpha}, D_{1,\alpha} \to D_{2,\alpha}, D_{2,\alpha} \to D_{3,\alpha} \mid \alpha \in \mathcal{I}\} \cup \\ &\quad \{D_{3,\alpha} \to \beta \mid \alpha, \beta \in \mathcal{I},\ \mathit{NextState}(\alpha) = \mathit{State}(\beta)\} \cup \\ &\quad \{D_{3,\alpha} \to F \mid \alpha \in \mathcal{I},\ \mathit{NextState}(\alpha) = q_F\}. \end{aligned}$$

This component selects the transition of M to be simulated. The axiom, S, introduces a symbol denoting an initial transition, when symbol F appears, then the simulation of the last transition has been finished.

Next we define $\mathcal{C}_2 = \{G_{c_1}, G_{c_2}\}$. Let for $i \in \{1, 2\}$,

$$\begin{aligned} P_{c_i} &= \{S \to Q_1W_i, F \to F_{c,i,1}, F_{c,i,3} \to F_{c,i,3}, F_{c,i,j} \to F_{c,i,j+1} \mid \\ &\qquad 1 \leq j \leq 2\} \cup \\ (1,i) &\quad \{\alpha \to X_{1,\alpha,i}, X_{1,\alpha,i} \to X_{2,\alpha,i}, X_{2,\alpha,i} \to y_{\alpha,i}, A \to Q_1 \mid \alpha \in \mathcal{I}, \\ &\qquad \mathit{Store}(\alpha, i) = B, y_{\alpha,i} = \sigma(\mathit{Action}(\alpha, i), B)\} \cup \\ (2,i) &\quad \{\alpha \to H_{1,\alpha,i}, H_{1,\alpha,i} \to H_{2,\alpha,i}, H_{2,\alpha,i} \to H_{3,\alpha,i}, H_{3,\alpha,i} \to y_{\alpha,i}Q_1 \mid \\ &\qquad \alpha \in \mathcal{I}, \mathit{Store}(\alpha, i) = Z, y_{\alpha,i} = \sigma(\mathit{Action}(\alpha, i), Z)\}, \end{aligned}$$

where $\sigma : \{1, 0, -1\} \times \{B, Z\} \to \{\varepsilon, A, AA\}$ is a partial mapping defined as $\sigma(1, B) = AA$, $\sigma(0, B) = \sigma(1, Z) = A$, and $\sigma(-1, B) = \sigma(0, Z) = \varepsilon$.

These components simulate the change in the contents of the counters which is represented by a string consisting of as many letters A as the actual stored number in the counter. Let $C_3 = \{G_{ch}\}$, and let

$$\begin{aligned} P_{ch} &= \{S \to Q_1\} \cup \\ (3) &\quad \{\alpha \to E_{1,\alpha}, E_{1,\alpha} \to Q_2, H_{2,\alpha,1} \to Q_2, H_{3,\alpha,2} \to Q_1 \mid \\ &\qquad \alpha \in \mathcal{I}_{(Z,Z)}\} \cup \\ (4) &\quad \{\alpha \to E_{1,\alpha}, E_{1,\alpha} \to E_{2,\alpha}, E_{2,\alpha} \to Q_2, H_{3,\alpha,1} \to Q_1 \mid \\ &\qquad \alpha \in \mathcal{I}_{(Z,B)}\} \cup \\ (5) &\quad \{\alpha \to E_{1,\alpha}, E_{1,\alpha} \to E_{2,\alpha}, E_{2,\alpha} \to Q_2, H_{3,\alpha,2} \to Q_1 \mid \\ &\qquad \alpha \in \mathcal{I}_{(B,Z)}\} \cup \\ (6) &\quad \{\alpha \to E_{1,\alpha}, E_{1,\alpha} \to E_{2,\alpha}, E_{2,\alpha} \to E_{3,\alpha}, E_{3,\alpha} \to Q_1 \mid \\ &\qquad \alpha \in \mathcal{I}_{(B,B)}\} \cup \\ (7) &\quad \{F \to F_{ch,1}, F_{ch,4} \to F_{ch,4}\} \cup \{F_{ch,j} \to F_{ch,j+1} \mid 1 \leq j \leq 3\}. \end{aligned}$$

This component assists in checking whether the respective counter is zero if it is required by the transition to be performed. This is done by requesting the string of component G_{c_1} and/or G_{c_2} from $\mathcal{C}_2$. If the obtained string represents an empty counter, then it must not have an occurrence of A.

Finally, we define $\mathcal{C}_4 = \{G_{gen}\}$, and

$$\begin{aligned} P_{gen} &= \{S \to Q_1, C_1 \to C_2, C_2 \to C_3, C_3 \to Q_1\} \cup \\ &\quad \{\alpha \to xC_1 \mid \alpha \in \mathcal{I}, \mathit{Read}(\alpha) = x\} \cup \\ &\quad \{F \to Q_2, F_{c,1,1} \to Q_2, F_{c,2,2} \to Q_3, F_{ch,3} \to \varepsilon, W_1 \to \varepsilon, \\ &\quad\ W_2 \to \varepsilon\}. \end{aligned} \tag{8}$$

This component generates the string accepted by the counter machine by adding the symbol $x = \mathit{Read}(\alpha)$ for each $\alpha \in \mathcal{I}$ (chosen by the selector component G_{sel}) using the rule $\alpha \to xC_1$.

Next we discuss the work of Γ in more detail. After the first rewriting step and the following communication, we obtain a configuration $(\alpha, \alpha W_1, \alpha W_2, \alpha, \alpha)$, where $\mathit{State}(\alpha) = q_0$, and then Γ simulates the transition corresponding to α.

To see how the simulation is done, let us consider a configuration of the form $(\alpha, u_1\alpha u_2 W_1, v_1 \alpha v_2 W_2, \bar{z}_1 \alpha \bar{z}_2, w\alpha)$ where $u_1u_2, v_1v_2 \in \{A\}^*$, $\bar{z}_1\bar{z}_2 \in \{W_1, W_2\}^*$, and $w \in T^*$. Suppose that up to transition α the simulation of the work of M was correct, that is, $\mathit{State}(\alpha)$ corresponds to the state of M, w corresponds to the string read by M, and u_1u_2, v_1v_2 contain the same number of As as stored on the counter tapes of M. Depending on c_1 and c_2 in $\alpha = [q, x, c_1, c_2, q', e_1, e_2]$, we discuss the next steps of Γ separately.

Let $\alpha = [q, x, B, Z, q', e_1, e_2] \in \mathcal{I}$, where $x \in T \cup \{\varepsilon\}$, $q, q' \in Q$, and $e_1 \in \{-1, 0, +1\}$, $e_2 \in \{0, +1\}$. Furthermore, let $\beta \in \mathcal{I}$ with $\mathit{NextState}(\alpha) = \mathit{State}(\beta)$.

In the next rewriting step, α changes to $D_{1,\alpha}$ at component G_{sel}, then in the second and third rewriting steps, $D_{1,\alpha}$ to $D_{2,\alpha}$, $D_{2,\alpha}$ to $D_{3,\alpha}$. Similarly, $w\alpha$ at G_{gen} changes to wxC_1, then to wxC_2, and finally wxC_3, where $x = \mathit{Read}(\alpha)$.

Let us examine now $u_1\alpha u_2 W_1$ which represents the contents of the first counter. If $e_1 = -1$, then due to the simulated transition, the counter must store a non-negative integer, u_1u_2 should have at least one occurrence of A. If this is not the case, then the rule $A \to Q_1$ in $(1, 1)$ cannot be applied since $y_{\alpha,1} = \varepsilon$. Thus, after the third step, the derivation is blocked since there is no rule to be applied.

If $e_1 \in \{0, +1\}$, then $y_{\alpha,1} \in \{A, AA\}$, which means that the derivation can also continue in the case when u does not contain any symbol A. The simulation remains correct, however, as we have assumed that $\alpha = [q, x, B, Z, q', e_1, e_2]$ for $e_1 \in \{0, +1\}$ implies that $\alpha' = [q, x, Z, Z, q', e_1, e_2]$ is also a possible transition.

If the derivation is not blocked, then after three rewriting steps performed on $u_1 \alpha u_2 W_1$ and the communication following them, the next cases may hold: The only nonterminal in the new sentential form is W_1 (rules $\alpha \rightarrow X_{1,\alpha,1}$, $X_{1,\alpha,1} \rightarrow X_{2,\alpha,1}$, and $X_{2,\alpha,1} \rightarrow y_{\alpha,1}$ were applied) or an occurrence of $D_{j,\alpha}$, $1 \leq j \leq 3$, is in the new string (rule $A \rightarrow Q_1$ was applied). Since symbols $D_{j,\alpha}$ cannot be erased in the further derivation steps from the string sent at the end of the derivation to the component G_{gen}, Γ can generate a terminal word only if no occurrence of A is rewritten in the first three steps. Therefore, after the fourth rewriting step starting from $u_1 \alpha u_2 W_1$, we must have a string of the form $u_1' Q_1 u_2' W_1$ where $u_1' A u_2' = u_1 y_{\alpha,1} u_2$ and $y_{\alpha,1}$ corresponds to $e_1 = \mathit{Action}(\alpha, 1)$ for $\alpha = [q, x, B, Z, q', e_1, e_2]$ as follows: if $e_1 = -1$, then $y_{\alpha,1} = \varepsilon$, if $e_1 = 0$, then $y_{\alpha,1} = A$, if $e_1 = +1$, then $y_{\alpha,1} = AA$.

Let us consider now $v_1 \alpha v_2 W_2$, i. e., the string representing the contents of the second counter. In this case, $v_1 v_2$ must not have any occurrence of A. If this is the case, then only the rules in $(2, 2)$ can be applied. After the third rewriting step, the new string will be of the form $v_1 H_{3,\alpha,2} v_2 W_2$ which will be forwarded to component G_{ch} (this is guaranteed by the rules of G_{ch}) and stored there until the end of the derivation when it is sent to G_{gen} in the master cluster $\mathcal{C}_4$. Since G_{gen} does not have rules for erasing As, terminal words can only be generated if G_{ch} received a string representing an empty counter. Notice that G_{c_1} and G_{c_2} are in the same cluster, so G_{ch} might receive the sentential form of any of them. However, if a string not containing $H_{3,\alpha,2}$, i.e., the string of G_{c_1} is forwarded to G_{ch}, then the derivation is blocked, since G_{ch} cannot continue its work.

Assume now that $v_1 v_2$ contains at least one copy of A. Then, after three rewriting steps the following cases may appear: No occurrence of A is rewritten, thus the obtained string, $v_1 H_{3,\alpha,2} v_2 W_2$, contains at least one A. In this case no terminal word can be generated, as we discussed before. If at least one occurrence of A is rewritten, then at least one of the symbols $D_{j,\alpha}$, $1 \leq j \leq 3$, is introduced in the string. This case leads to a blocking situation, since the string is sent to G_{ch} which has no rule for continuing the rewriting.

Thus, the string obtained from $v_1 \alpha v_2 W_2$ after the fourth rewriting step must be $y_{\alpha,2} Q_1 W_2$, where $y_{\alpha,2}$ corresponds to $e_2 = \mathit{Action}(\alpha, 2)$. Since no

A could be deleted, $y_{\alpha,2} = \varepsilon$ if $e_2 = 0$, and $y_{\alpha,2} = A$ if $e_2 = +1$ (the case $e_2 = -1$ is not applicable, since the counter stores zero, $Store(\alpha, 2) = Z$).

Continuing the derivation, the prescribed communication step results in the configuration $(\beta, u_1'\beta u_2'W_1, v'\beta W_2, \bar{z}_1'\beta\bar{z}_2', w'\beta)$ where $u_1'u_2', v', \in \{A\}^*$, $\bar{z}_1'\bar{z}_2' \in \{W_1, W_2\}^*$, $w' \in T^*$, and $\beta \in \mathcal{I}$ is a transition with $NextState(\alpha) = State(\beta)$. Now, similarly as above, the simulation of the transition corresponding to the symbol $\beta \in \mathcal{I}$ can be performed.

It is easy to see that the case $\alpha = [q, x, Z, B, q', e_1, e_2]$ can be treated in the same way, with changing the discussion concerning components G_{c_1} and G_{c_2}. If $\alpha = [q, x, B, B, q', e_1, e_2]$, then we use the same reasoning for G_{c_2} as we did for G_{c_1} above. The case $\alpha = [q, x, Z, Z, q', e_1, e_2]$ can be obtained similarly: The simulation can proceed if the sentential form of G_{ch} contains $H_{2,\alpha,1}$ after the second, and $H_{3,\alpha,2}$ after the third derivation step, and has no occurrence of A.

Now we discuss how the derivation terminates. Suppose, $NextState(\alpha) = q_F$ and G_{sel} changes the nonterminal $D_{3,\alpha}$ to F. Then, in the next four derivation steps the sentential forms of G_{c_1}, G_{c_2}, and G_{ch} are forwarded to component G_{gen} in this order, or the derivation is blocked. The simulation is successful, if by erasing $F_{ch,3}$ and applying rules $W_1 \to \varepsilon$ and $W_2 \to \varepsilon$ as many times as necessary, G_{gen} obtains a terminal word.

Thus, we have shown that $\mathcal{L}(\mathrm{NPC}_{4/5}\mathrm{CF}) \supseteq \mathcal{L}(\mathrm{RE})$. Since due to the Church thesis, $\mathcal{L}(\mathrm{NPC}_{4/5}\mathrm{CF}) \subseteq \mathcal{L}(\mathrm{RE})$ holds as well, we proved the result.

For the second equality, let $\Gamma = (N, K, T, G_{sel}, G_{c_1}, G_{c_2}, G_{ch}, G_{ass}, G_{gen})$, where the components are $G_\gamma = (N \cup K, T, P_\gamma, S)$, $\gamma \in \{sel, c_1, c_2, ch, ass, gen\}$, and G_{gen} is the master grammar. Let $N = N' \cup \{F_{ass}\}$ where N' contains the same symbols as the nonterminals of the system above. Let $K = \{Q_1, Q_2\}$ and let us define the sets of productions $P_{sel}, P_{c_1}, P_{c_2}, P_{ch}$, P_{gen} exactly as in the previous case. The rules of the newly added assistant component are as follows.

$$\begin{aligned} P_{ass} &= \{S \to Q_1, \alpha \to Q_1, D_{1,\alpha} \to Q_1, D_{2,\alpha} \to Q_1, D_{3,\alpha} \to Q_1 \mid \\ &\qquad \alpha \in \mathcal{I}\} \cup \{F \to F_{ass}, F_{ass} \to F_{ass}\}. \end{aligned}$$

The dynamically clustered non-returning PC grammar system Γ works very similarly to the system with the predefined clusters. The newly added component G_{ass} has rewriting rules only for the sentential forms of G_{sel}, thus, when it introduces in each rewriting step the query symbol Q_1, it makes sure that all occurring symbols Q_1 have to be assigned to a dynamically formed cluster $\mathcal{C}_1$ which contains G_{sel}, and moreover, all occurring symbols Q_1 have to be replaced by the sentential form of G_{sel}. (Note that Γ works in the homogeneous communication mode.)

It is left to the reader to check that this is sufficient to see that Γ also simulates the work of the two-counter machine M and also to show that the reverse inclusion, for the equality, holds. $\square$

As (static) clustered non-returning PC grammar systems are special cases of non-clustered systems, that is, $\mathcal{L}(\mathrm{NPC}_n\mathrm{CF}) = \mathcal{L}(\mathrm{NPC}_{n/n}\mathrm{CF})$ by definition, Theorem 4.3.1 also implies that the necessary number of components of standard (non-clustered) non-returning PC grammar systems to generate any recursively enumerable language is five.

Corollary 4.3.2. $\mathcal{L}(\mathrm{NPC}_5\mathrm{CF}) = \mathcal{L}(\mathrm{RE})$.

Proof. Notice that for any $n \geq k$, $\mathcal{L}(\mathrm{NPC}_{k/n}\mathrm{CF}) \subseteq \mathcal{L}(\mathrm{NPC}_{n/n}\mathrm{CF})$. To see this, consider a clustered non-returning PC grammar system with n components and k clusters where $n > k$, and consider a modified system with n clusters which are formed from the n components and with each rule introducing queries being replaced by several rules in the following way: if $\mathcal{C}_i = \{G_{i_1}, \ldots, G_{i_k}\}$ is a cluster, then any rule $X \to uQ_iv$ is replaced with the rules $X \to uQ_{i_1}v, \ldots, X \to uQ_{i_k}v$. (If more than one query symbols are present, then all possible combinations are added.) $\square$

We can also show that considering sets of integers (languages over the unary alphabet), the bound on the necessary number of components can be further decreased.

Let $\mathcal{L}(\mathbb{N}\mathrm{RE})$ denote the class of recursively enumerable languages over unary alphabets.

Theorem 4.3.3. $\mathcal{L}(\mathbb{N}\mathrm{RE}) \subseteq \mathcal{L}(\mathrm{NPC}_4\mathrm{CF})$ *and* $\mathcal{L}(\mathbb{N}\mathrm{RE}) \subseteq \mathcal{L}(\mathrm{DNPC}_{3/5}\mathrm{CF})$.

Proof. Let $L \in \mathcal{L}(\mathbb{N}\mathrm{RE})$ over a unary alphabet $T = \{a\}$, and consider the register machine $M = (2, H, q_0, q_F, R)$ (see Chapter 2 for the definition) with two registers and with $L(M) = L$. Let us define the set of symbols $\mathcal{I}$ as follows.

$$\begin{aligned}\mathcal{I} \;=\; & \{[q,c_1,c_2;0,+1,q'] \mid q:(\texttt{ADD}(2),q'), q:(\texttt{nADD}(2),q',r)\in R, \text{ or} \\ & \quad q:(\texttt{nADD}(2),r,q')\in R,\ c_1,c_2\in\{B,Z\}\}\cup \\ & \{[q,c_1,c_2;+1,0,q'] \mid q:(\texttt{ADD}(1),q'), q:(\texttt{nADD}(1),q',r)\in R, \text{ or} \\ & \quad q:(\texttt{nADD}(1),r,q')\in R,\ c_1,c_2\in\{B,Z\}\}\cup \\ & \{[q,c_1,B;0,-1,q'] \mid q:(\texttt{SUB}(2),q')\in R,\ c_1\in\{B,Z\}\}\cup \\ & \{[q,B,c_2;-1,0,q'] \mid q:(\texttt{SUB}(1),q')\in R,\ c_2\in\{B,Z\}\}\cup \\ & \{[q,Z,c_2;0,0,q'] \mid q:(\texttt{CHECK}(1),q')\in R,\ c_2\in\{B,Z\}\}\cup \\ & \{[q,c_1,Z;0,0,q'] \mid q:(\texttt{CHECK}(2),q')\in R,\ c_1\in\{B,Z\}\}.\end{aligned}$$

Analogously to the previous proof, we also introduce for any $\alpha\in\mathcal{I}$ with $\alpha=[q,c_1,c_2;e_1,e_2,q']$, the notations: $State(\alpha)=q$, $NextState(\alpha)=q'$, $Store(\alpha,i)=c_i$, $Action(\alpha,i)=e_i$, where $i=1,2$, and we also define for $c',c''\in\{B,Z\}$ subsets of $\mathcal{I}$ as $\mathcal{I}_{(c',c'')}=\{\alpha\in\mathcal{I}\mid\alpha=[q,c',c'';,e_1,e_2,q']\}$. To prove the first inclusion, we construct a clustered non-returning PC grammar system generating L by simulating the transitions of M.

Let $\Gamma=(N,K,T,G_{sel},G_{c_1},G_{c_2},G_{ch},\mathcal{C}_{sel},\mathcal{C}_{c_1},\mathcal{C}_{c_2},\mathcal{C}_{ch})$, where $G_\gamma=(N\cup K,T,P_\gamma,S)$, $\gamma\in\{sel,c_1,c_2,ch\}$ are the component grammars, and $\mathcal{C}_\gamma=\{G_\gamma\}$ are the clusters, $\mathcal{C}_{sel}$ is the master cluster. The construction of Γ is similar to the one found in the proof of Theorem 4.3.1. Let $N=\mathcal{I}\cup\{D_{i,\alpha},E_{i,\alpha},H_{i,\alpha},H_{i,\alpha,j},X_{i,\alpha,j}\mid\alpha\in\mathcal{I},1\le i\le 3,1\le j\le 2\}\cup\{S,A_1,A_2,F,F_{sel},F_{c,1},F_{ch},W_1,W_2\}$.

The states and the transitions of M are represented with nonterminals from $\mathcal{I}$, the values of the counters by strings of nonterminals containing as many symbols A_i, as the value stored in the counter c_i, $1\le i\le 2$.

Let the components of Γ be defined as follows.

$$P_{sel} \;=\; P'_{sel}\cup\{F_{sel}\to Q_{c_2}Q_{ch},W_1\to\varepsilon,W_2\to\varepsilon,F_{ch}\to\varepsilon,F_{c_1}\to\varepsilon\}$$

where P'_{sel} contains the same rules as the set P_{sel} in the proof of Theorem 4.3.1. Let

$$P_{c_1}=P'_{c_1}\cup\{F\to F_{c,1},F_{c,1}\to F_{c,1}\},$$

$$P_{c_2}=P'_{c_2}\cup\{F\to Q_{c_1},A_1\to a,F_{c,1}\to F_{c,1}\}$$

where for $i \in \{1,2\}$,

$$\begin{aligned} P'_{c_i} \;=\; & \{\alpha \to X_{1,\alpha,i}, X_{1,\alpha,i} \to X_{2,\alpha,i}, X_{2,\alpha,i} \to y_{\alpha,i}, A_i \to Q_{sel} \mid \alpha \in \mathcal{I}, \\ & \quad Store(\alpha,i)=B, y_{\alpha,i}=\sigma(Action(\alpha,i),B,i)\} \cup \\ & \{\alpha \to H_{1,\alpha,i}, H_{1,\alpha,i} \to H_{2,\alpha,i}, H_{2,\alpha,i} \to H_{3,\alpha,i}, H_{3,\alpha,i} \to y_{\alpha,i}Q_{sel} \mid \\ & \quad \alpha \in \mathcal{I}, Store(\alpha,i) = Z, y_{\alpha,i} = \sigma(Action(\alpha,i),Z,i)\} \cup \\ & \{S \to Q_{sel}W_i\}, \end{aligned}$$

with $\sigma : \{1,0,-1\} \times \{B,Z\} \times \{1,2\} \to \{\varepsilon, A_1, A_2, A_1A_1, A_2A_2\}$ is a partial mapping defined as $\sigma(1,B,i) = A_iA_i$, $\sigma(0,B,i) = \sigma(1,Z,i) = A_i$, and $\sigma(-1,B,i) = \sigma(0,Z,i) = \varepsilon$, $i \in \{1,2\}$.

These components simulate the change in the contents of the counters as in the case of the proof of Theorem 4.3.1, the only difference being that for the representation of the two registers A_1 and A_2, they use two different symbols, A_1 and A_2, respectively.

Let also $P_{ch} = P'_{ch} \cup \{F \to F_{ch}, F_{ch} \to F_{ch}\}$ where P'_{ch} contains the rules of P_{ch} from the proof of Theorem 4.3.1 with the exception of the rules (7).

The simulation of M is also done in a similar way. After the initial rewriting step and the following communication, a configuration $(\alpha_0, \alpha_0W_1, \alpha_0W_2, \alpha_0)$ is obtained where $\alpha_0 \in \mathcal{I}$ corresponds to a transition with $State(\alpha_0) = q_0$.

Then a computation is simulated by executing the transitions corresponding to the chosen symbols from $\mathcal{I}$, and eventually $(F, u_1Fu_2W_1, v_1Fv_2W_2, zF)$ is reached, a configuration which corresponds to a final configuration of M, that is, if the simulation was correct, then $u_1u_2 = A_1^j$ for some $j \geq 0$ stores the value j computed by M, $v_1v_2 = \varepsilon$, and $z \in \{W_1, W_2\}^*$. The checking of the correctness of the simulation and the generation of the terminal string a^j is done by entering the configuration $(F_{sel}, u_1F_{c,1}u_2W_1, v_1Q_{c_1}v_2W_2, zF_{ch})$, and after performing the communication $(F_{sel}, u_1F_{c,1}u_2W_1, v_1u_1F_{c,1}u_2W_1v_2W_2, zF_{ch})$. Now, the symbols A_1 in u_1u_2 are changed to a by the rule of P_{c_2}, and then the resulting string is transferred to G_{sel} together with the sentential form of G_{ch}. If erasing the nonterminals $W_1, W_2, F_{ch}, F_{c,1}$ results in a terminal string a^j, then the simulation of M computing the value j was correct.

For the second inclusion, we modify this construction similarly to the second part of the proof of Theorem 4.3.1. Let $\Gamma = (N, K, T, G_{sel}, G_{c_1}, G_{c_2}, G_{ch}, G_{ass})$, where $G_\gamma = (N \cup K, T, P_\gamma, S)$, $\gamma \in \{sel, c_1, c_2, ch, ass\}$ are the component grammars, and G_{sel} is the master grammar. Let $N = N' \cup \{F_t\}$ where N' contains the same symbols as the nonterminals of the system in the previous part of the proof, and let $K = \{Q_1, Q_2\}$. The sets of productions $P_{sel}, P_{c_1}, P_{c_2}$, and P_{ch} are defined as above, but replacing $Q_{sel}, Q_{c_1}, Q_{c_2}$ in the rules by Q_1, and Q_{ch} by Q_2. The rule $F_{c_1} \to F_{c_1}$ of P_{c_1} is replaced by $F_{c_1} \to F_t$ and $F_t \to F_t$. The rules of the newly added assistant component are as follows.

$$\begin{aligned} P_{ass} &= \{S \to Q_1, \alpha \to Q_1, D_{1,\alpha} \to Q_1, D_{2,\alpha} \to Q_1, D_{3,\alpha} \to Q_1 \mid \\ &\qquad \alpha \in \mathcal{I}\} \cup \{F \to Q_1, F_{c_1} \to F_t, F_t \to F_t\}. \end{aligned}$$

This dynamically clustered PC grammar system works very similarly to the one in the second part of the proof of Theorem 4.3.1. The newly added component G_{ass} makes sure that all occurring symbols Q_1 have to be assigned to a dynamically formed cluster $\mathcal{C}_1$ which contains G_{sel} during the instruction simulating phase, or G_{c_1} in the final phase of the simulation. $\square$

4.3.1 Remarks

In this section we have studied the bound for the number of components needed to generate any recursively enumerable language by clustered and standard non-returning PC grammar systems. We have shown that five components and four clusters are sufficient for non-returning systems for the generation of recursively enumerable languages. This result implies that five components are also sufficient in the standard, non-clustered case, which shows that non-returning systems are as efficient from this point of view as returning ones (see (Csuhaj-Varjú et al., 2003) for the best result on the returning case). We have also shown that this bound is even less if we consider unary languages: four components are enough in that case. Then we have introduced the concept of dynamical clustering, and shown that three dynamical clusters, that is, two different query symbols and six component grammars are sufficient to obtain all recursively enumerable languages with dynamical clusters. The results of this section first appeared in (Csuhaj-Varjú and Vaszil, 2011).

Chapter 5

Membrane Systems - The Breadth of Rules and the Number of Regions

In this chapter we discuss descriptional complexity issues of two membrane system variants: P systems with symport/antiport rules and P colonies. A symport/antiport system uses communication rules only, thus, the direct change of the objects is not possible, the rules only specify how the objects are exchanged between the different regions. P colonies consist of a collection of cell-like agents placed in an environment which is, similarly to symport/antiport systems, represented by a multiset of objects processed by the agents during the computation. The processing rules associated to the cells are very simple: they can change one object which is inside the cell, or exchange one object which is inside with another one which is outside. These rules are grouped into "programs" which are executed by the cell on all objects inside of it in parallel.

In the following, we first deal with the descriptional complexity measures of symport/antiport P systems, namely the complexity of the communication rules (this is measured by their breadth, the size of the multisets handled by the rule) and the complexity of the membrane structure (measured by the number of membranes contained by the system). Then we also examine some of the descriptional complexity aspects of P colonies: the number of cells in the system, the number of programs associated to the cells, and the number of objects inside the cells.

5.1 Preliminaries

In the next section we will use the notion of an n-counter automaton. It is a modification of what we called n-counter machine in Chapter 2. It can be seen as the generative version of n-counter machines for unary alphabets, where in addition, only one of the counters are changed in any computational step. We present the definition in the form given, for example, in (Frisco, 2004).

Definition 5.1.1. A *counter automaton* is a construct $M = (Q, C, R, q_0, f)$ where Q is a set of states, $C = \{c_0, c_1, \ldots, c_n\}$ is a set of counters, c_0 being the output counter, R is a set of transitions of the form $(r \to s, X)$, for two states $r, s \in Q$, and an instruction $X \in \{i+, i-, e, (i = 0) \mid 0 \leq i \leq n\}$, $q_0 \in Q$ is the initial state, and $f \in Q$ is the final state.

If a transition $(r \to s, X)$ is an element of R, then the machine can pass from state r to state s executing X. If X is $i+$ or $i-$, then it instructs the machine to increase or decrease, respectively, the value of counter c_i by one, if it is e, then it instructs the machine to leave the counter values unchanged, if it is $i = 0$, then the transition from r to s is only possible by having zero as the contents of counter c_i.

Definition 5.1.2. The configuration of a counter automaton M is given by the $(n+2)$-tuple $(q, c_0, c_1, \ldots, c_n)$ where $q \in Q$ is a state, and $c_i \in \mathbb{N}$, $0 \leq i \leq n$, are the values stored in the counters, c_0 being the value of the output counter. The initial configuration is $(q_0, 0, 0, \ldots, 0)$, a final configuration is of the form $(f, c_0, c_1, \ldots, c_n)$ where f is the final state of M.

Given a configuration $(q, c_0, c_1, \ldots, c_n)$, the machine can pass to configuration $(q', c'_0, c'_1, \ldots, c'_n)$, denoted as $(q, c_0, c_1, \ldots, c_n) \Rightarrow (q', c'_0, c'_1, \ldots, c'_n)$, if $(q \to q', X) \in R$, and

- if $X = j+$, then $c'_j = c_j + 1$ and $c'_i = c_i,\ 0 \leq i \leq n,\ i \neq j$,
- if $X = j-$, then $c'_j = c_j - 1$ and $c'_i = c_i,\ 0 \leq i \leq n,\ i \neq j$,
- if $X = e$, then $c'_i = c_i,\ 0 \leq i \leq n$,
- if $X = (j = 0)$, then $c'_i = c_i,\ 0 \leq i \leq n$, and $c_j = 0$.

Let $\Rightarrow^*$ denote the reflexive and transitive closure of $\Rightarrow$.

A computation is a sequence of such transitions leading from the initial configuration to a final configuration, its result can be read from the output counter.

Definition 5.1.3. The set of nonnegative integers generated by a counter automaton M as above is the following.

$$\begin{aligned} N(M) \;=\; & \{c_0 \in \mathbb{N} \mid (q_0, 0, 0, \ldots, 0) \Rightarrow^* (f, c_0, c_1, \ldots, c_n),\ \text{where} \\ & q_0 \text{ and } f \text{ are the initial and the final states, respectively}\}. \end{aligned}$$

It is clear that counter automata are able to generate any recursively enumerable set of numbers if they have two or more counters beside the output counter.

Notation. In the following, with $\mathbb{N}_k$RE for some $k \in \mathbb{N}$, we denote the class $\{k + L \mid L \in \mathbb{N}\text{RE}\}$ where $k + L = \{x + k \mid x \in L\}$ and $\mathbb{N}$RE denotes the class of recursively enumerable sets of nonnegative integers.

5.2 P system with symport/antiport rules

Now we recall the definition of symport/antiport P systems from (Păun and Păun, 2002). A symport rule consists of the description of a finite multiset and a direction ("in" or "out") specifying that in a certain computational step the given multiset may enter from the parent region (in case of the direction "in") or leave to the parent region (in case of "out"). An antiport rule specifies two multisets, one with the direction "in" and the other with the direction "out", which means that the objects of the corresponding multisets may enter and leave the given region simultaneously in one computational step. (Of course, such rules can only be applied if the necessary objects are present in the corresponding regions in a sufficient amount.)

A P system is a structure of hierarchically embedded membranes, each having a label and enclosing a region containing a multiset of objects and possibly other membranes. The out-most membrane which is unique and usually labeled with 1, is called the skin membrane. The membrane structure is denoted by a sequence of matching parentheses where the matching pairs have the same label as the membranes they represent. If $x \in \{[_i,]_i \mid 1 \leq i \leq n\}^*$ is such a string of matching parentheses of length $2n$, denoting a structure where membrane i contains membrane j, then $x = x_1\ [_i\ x_2\ [_j\ x_3\ ]_j\ x_4\ ]_i\ x_5$ for some $x_k \in \{[_l,]_l \mid 1 \leq l \leq n,\ l \neq i, j\}^*,\ 1 \leq k \leq 5$. If membrane i contains membrane j, and there is no other membrane, k, such that k contains j and i contains k (x_2 and x_4 above are strings of matching parentheses themselves), then we say that membrane i is the parent membrane of j. A membrane m is called elementary if it contains no membrane, in this case $x = x_1\ [_m\ ]_m\ x_2$.

The evolution of the contents of the regions of a P system is described by rules associated to the regions. Applying the rules synchronously in each region, the system performs a computation by passing from one configuration to another one. In the following we concentrate on communication rules called symport or antiport rules.

A symport rule is of the form (x, in) or $(x, out), x \in V^*$. If such a rule is present in a region i, then the objects of the multiset x must enter from the parent region or must leave to the parent region, respectively. An antiport rule is of the form $(x, in; y, out), x, y \in V^*$, in this case, objects of x enter from the parent region and in the same step, objects of y leave to the parent region.

The rules are applied in the maximal parallel manner, that is, as many rules are applied in each region as possible. The end of the computation is defined by halting: A P system halts when no more rules can be applied in any of the regions, the result is the number of objects in an elementary membrane labeled as output.

Definition 5.2.1. A P system with symport/antiport of degree $n \geq 1$ is a construct

$$\Pi = (V, \mu, E, w_1, \ldots, w_n, R_1, \ldots, R_n, out)$$

where

- V is an alphabet of objects,
- μ is a membrane structure of n membranes,
- $E \subseteq V$ is a set of objects (the ones which can be found in the environment in an arbitrary number of copies),
- $w_i \in V^*,\ 1 \leq i \leq n$, are the initial contents of the n regions,
- $R_i,\ 1 \leq i \leq n$, are the sets of symport/antiport rules associated to the regions,

- $out \in \{1, \dots, n\}$ is the label of an elementary membrane, the output membrane.

To simplify the notations we denote both symport and antiport rules as $(x, in; y, out)$, $x, y \in V^*$ where we also allow at most one of x, y to be the empty multiset. If $y = \varepsilon$ or $x = \varepsilon$, then the notation above denotes the symport rule (x, in) or (y, out), respectively. Moreover, when no confusion arises, we will enumerate the elements $a_1, \dots, a_n$ of a multiset as $M = \{a_1, \dots, a_n\}$, similarly to the usual set notation.

The $n+1$-tuple of finite multisets of objects present in finite number of copies in the environment and in the n regions of the P system Π describes a *configuration* of Π; $(\varepsilon, w_1, \dots, w_n) \in (V^*)^{n+1}$ is the initial configuration.

Definition 5.2.2. For a configuration $(u_0, \dots, u_n)$, the P system may enter the new configuration $(u'_0, \dots, u'_n)$, denoted as $(u_0, \dots, u_n) \Rightarrow (u'_0, \dots, u'_n)$, if there exist rules as follows.

For all $i, 1 \leq i \leq n$, there is a multiset of rules $P_i = \{\{r_{i,1}, \dots, r_{i,m_i}\}\}$, where $r_{i,j} = (x_{i,j}, in; y_{i,j}, out) \in R_i$ satisfying the conditions below where x_i, y_i denote the multisets $\bigcup_{1 \leq j \leq m_i} x_{i,j}$ and $\bigcup_{1 \leq j \leq m_i} y_{i,j}$, respectively. Furthermore, there is no $r \in R_j$, for any j, $1 \leq j \leq n$, such that the rule multisets P'_i with $P'_i = P_i$ for $i \neq j$ and $P'_j = \{\{r\}\} \cup P_j$, also satisfy the conditions which are given as

$$x_1 = x'_1 \cup x''_1 \text{ with } supp(x'_1) \subseteq E, \; x''_1 \subseteq u_0, \text{ and}$$

$$\bigcup_{parent(j)=i} x_j \cup y_i \subseteq u_i, \text{ for } 1 \leq i \leq n.$$

Then the new configuration is obtained by

$$u'_0 = u_0 - x''_1 \cup y_1, \text{ and}$$

$$u'_i = u_i \cup x_i - y_i \cup \bigcup_{parent(j)=i} y_j - \bigcup_{parent(j)=i} x_j, \text{ for } 1 \leq i \leq n.$$

The P system generates numbers as follows.

Definition 5.2.3. The set of numbers generated by a symport/antiport P system as above, $N(\Pi) \in \mathbb{N}RE$, is the following set.

$$\begin{aligned} N(\Pi) \;=\; & \{x = card(u_{out}) \mid (\varepsilon, w_1, \dots, w_n) \Rightarrow^* (u_0, \dots, u_n), \\ & \text{where } (u_0, \dots, u_n) \text{ is a halting configuration}\} \end{aligned}$$

and $\Rightarrow^*$ denotes the reflexive and transitive closure of $\Rightarrow$.

Notation. Let $\mathrm{NOP}_n(sym_r, anti_s)$ denote the class of sets of numbers generated by symport/antiport P systems of degree n where for all (x, in), (y, out), $(v, in; z, out) \in R_i$, $1 \leq i \leq n$, $card(x) \leq r$, $card(y) \leq r$, and $card(v) \leq s$, $card(z) \leq s$.

5.2.1 Symport/antiport P systems with minimal cooperation

P systems with symport/antiport were shown to be able to generate any recursively enumerable set of numbers already in (Păun and Păun, 2002). This result was improved from the point of view of the number of necessary membranes and the complexity of communication rules in (Frisco and Hoogeboom, 2004; Martín-Vide et al., 2002a; Martín-Vide et al., 2002b). The study of minimal symport/antiport, when the multisets in the rules contain at most one object, started in (Bernardini and Gheorghe, 2003) where it was shown that such systems with nine membranes generate any recursively enumerable set of numbers. Then the number of necessary membranes were decreased to six in (Kari et al., 2004), to five in (Bernardini and Păun, 2004), and then to four in (Frisco, 2004).

In this section we present an improvement of the last result by showing that three membranes are sufficient to generate any recursively enumerable set of numbers with minimal symport/antiport.

Theorem 5.2.1. $\mathrm{NOP}_3(sym_1, anti_1) = \mathbb{N}_5\mathrm{RE}$.

Proof. Consider the counter automaton $M = (Q, C, R, q_0, f)$ with counters $C = \{c_0, c_1, \ldots, c_n\}$, c_0 being the output counter, and let the transitions be uniquely labeled by elements of the set $lab(R)$. We construct a P system Π generating the language $L(\Pi) = \{x + 5 \mid x \in L(M)\}$ as follows. Let

$$\Pi = (V, \mu, E, w_1, w_2, w_3, R_1, R_2, R_3, 3)$$

where $\mu = [_1\ [_2\ [_3\ ]_3\ ]_2\ ]_1$, and

$$V = \{I_1, \bar{I}_1, I_2, I_3, I_4, I_5, \infty_1, \infty_2, \infty_3, \infty_4, C, f_1, f_1', f_2, f_2', \bar{f}_2, f_3, f_3', f_4\} \cup$$

$$\{t, t' \mid t \in lab(R)\} \cup \{c_i, 0_i \mid 0 \leq i \leq n\},$$

$$E \;=\; \{t, t' \mid t \in lab(R)\} \cup \{I_4, f_1, f_2, \bar{f}_2, f_3, f_4\} \cup \{c_i \mid 0 \leq i \leq n\}.$$

The initial region contents are

$$\begin{aligned} w_1 &= \{\{I_1, \bar{I}_1, I_2, I_3, \infty_1, \infty_2, \infty_3, \infty_4, \infty_4, C\}\}, \\ w_2 &= \{\{\infty_1, \infty_2\}\} \cup \{\{0_i \mid 0 \leq i \leq n\}\}, \text{ and} \\ w_3 &= \{\{I_5, f_1', f_2', f_3', \infty_3\}\}. \end{aligned}$$

The work of the P system can be divided into three phases:

- Initialization,
- simulation of the counter automaton, and
- termination.

In the *initialization phase* an arbitrary number of counter symbols $c_i, 0 \leq i \leq n$, are moved into region 1 and an arbitrary number of transition symbols t' for some $t \in lab(R)$ are moved into region 3.

In the *simulation phase* Π simulates M by modifying the number of counter symbols present in region 2 according to the counter contents of M as follows. First Π imports a transition symbol t_1 for a possible transition $t_1 : (q \to r, X) \in R$ of M into region 1. Then this symbol travels to region 2 and then to region 3 where it remains until the termination phase, and from where the primed version, t_1', is released to region 2 moving to region 1 where it is sent out to the environment and at the same time an other transition symbol $t_2 \in lab(R)$ is imported for an other valid transition $t_2 : (r \to s, Y)$ of M. While these transition symbols travel through the system to region 3 and back, the modifications on the number of counter symbols in region 2 are realized. If $X = i-$, then a copy of c_i is removed from region 2 when t_1 enters this region from region 1. If $X = i+$, then a copy of c_i is imported from region 1 to region 2 when t_1' moves from region 2 to region 1. For $X = (i = 0)$, through the aid of maximal parallel rule application, the system allows the above described movement of the transition symbols only in the case when region 2 does not contain any symbol c_i.

In the *termination phase*, the counter symbols corresponding to the output counter are moved to region three, then the possibly still present transition symbols of the form t, t' for some $t \in lab(R)$ are moved from region 3 to region 2. In case of an unsuccessful simulation, Π may never stop which is ensured by an infinite loop: A pair of ∞_1 symbols are present in region 1 and 2, together with the rule $(\infty_1, in; \infty_1, out)$ in region 2. This loop is "destroyed" only in the termination phase after a successful simulation allowing the computation to stop.

For the sake of easier readability we present the rules of Π in groups corresponding to these phases $R_i = R_i^{ini} \cup R_i^{sim} \cup R_i^{ter},\ 1 \le i \le 3$.

For $j,\ 0 \le j \le n$, and $t \in lab(R)$,

$$\begin{array}{rcl}(t', in; I_1, out), (I_1, in), (I_4, in; I_1, out), (c_j, in; \bar{I}_1, out), (\bar{I}_1, in) & \in & R_1^{ini},\\ (I_2, in), (t', in; I_2, out), (\infty_3, in; I_2, out), (I_3, in; \infty_2, out) & \in & R_2^{ini},\\ (I_3, in), (t', in; I_3, out) & \in & R_3^{ini}.\end{array}$$

With the help of the initialization symbols $I_1, \bar{I}_1, I_2, I_3 \in w_1$, these rules import an arbitrary number of transition symbols t' with $t \in lab(R)$ into region 3 and counter symbols $c_i,\ 0 \le i \le n$, into region 1. In the first step, I_1 and $\bar{I}_1$ are moved out of the system, I_2 and I_3 are moved to region 2. Since there will be other rules in region 1 for I_3, it is necessary to make sure that it is moved to region 2 by sending out the symbol ∞_2. If ∞_2 is not sent out, an infinite loop is formed which can not be later destroyed. By applying these rules in succession, the imported transition symbols are moved to region 3, the counter symbols remain in region 1. If for some reason, because of the application of some other rule, a transition symbol can not be moved to region 2 from region 1, then ∞_3 is moved into region 2 instead, creating an infinite loop. Another infinite loop involving the two ∞_1 objects keeps the system working until a correct simulation of a successful computation is finished, then it is removed, otherwise if the simulation does not follow the right track, the system will produce no result. These infinite loops need rules

$$\begin{array}{rcl}(\infty_1, in; \infty_1, out), (\infty_2, in; \infty_2, out) & \in & R_2^{ini}\\ (\infty_3, in; \infty_3, out) & \in & R_3^{ini}\end{array}$$

If once in region 1, instead of a transition symbol, I_4 is imported, then the initialization phase is finished using the following rules.

$$\begin{array}{rcl}(I_3, out) & \in & R_1^{ini}\\ (I_4, in; I_3, out), (I_1, in; I_4, out), (\infty_4, in; I_4, out), (\bar{I}_1, in; I_5, out) & \in & R_2^{ini}\\ (I_1, in; I_5, out), (I_2, in; I_1, out) & \in & R_3^{ini}\end{array}$$

First, the symbol I_3 is moved out from region 2 to region 1 and at the same time I_4 is moved from region 1 to region 2. Then I_3 leaves the system, and I_4 is sent back to region 1 while moving I_1 to region 2. Since there are other rules for I_1 in region 1, an infinite loop is created with the ∞_4 symbols if in this step I_1 is not moved to region 2. Then I_1 is transferred to region 3, where it brings also I_2 to region 3, and releases I_5 which moves to region 1 while bringing $\bar{I}_1$ to region 2. The infinite loop needs the rule

$$(\infty_4, in; \infty_4, out) \quad \in \quad R_2^{ini}.$$

Thus, at the end of a successful initialization phase, the system ends up in a configuration where u_1, u_2, u_3 are the multisets contained by the three regions as

$$\begin{aligned} u_1 &= \{\{c_{i_1}, \dots, c_{i_k} \mid i_j \in \{0, \dots, n\},\ 1 \le j \le k\}\} \cup \\ &\quad \{\{I_4, I_5, \infty_2, \infty_2, \infty_3, \infty_4, \infty_4, \infty_1, C\}\}, \\ \\ u_2 &= \{\{I_1, \bar{I}_1, \infty_1, 0_0, 0_1, \dots, 0_n\}\},\ \text{and} \\ \\ u_3 &= \{\{t_1', \dots, t_m' \mid t_j \in lab(R),\ 1 \le j \le m\}\} \cup \\ &\quad \{\{I_2, f_1', f_2', f_3', \infty_3\}\}. \end{aligned}$$

The simulation of the counter automaton is realized with the following rules.

$$\begin{aligned} R_1^{sim} &= \{(t_0, in; I_5, out), (t_2, in; t_1', out) \mid t_0, t_1, t_2 \in lab(R) \text{ with} \\ &\quad t_0 : (q_0 \to q, X), t_1 : (q \to r, Y), t_2 : (r \to s, Z) \text{ for} \\ &\quad \text{some } X, Y, Z\}, \\ \\ R_2^{sim} &= \{(t, in), (c_i, in; t', out) \mid t : (r \to s, i+) \in R\} \cup \\ &\quad \{(t, in; c_i, out), (t', out) \mid t : (r \to s, i-) \in R\} \cup \\ &\quad \{(t, in), (t', out) \mid t : (r \to s, e) \in R\} \cup \\ &\quad \{(t, in; 0_i, out), (0_i, in; c_i, out), (0_i, in; t', out) \mid \\ &\quad\ t : (r \to s, i = 0) \in R\}, \\ \\ R_3^{sim} &= \{(t, in; t', out) \mid t \in lab(R)\}. \end{aligned}$$

First I_5 is sent out of the system and one transition symbol, denoting a transition from the initial state q_0 is imported, then the transition corresponding to the symbol is simulated. This is done by moving the transition symbol to the third region, exchanging it to its primed version, and moving the primed version back t region 1. While the transition symbol travels through the regions, it adds or subtracts a counter symbol to or from region 2 when necessary. If the instruction corresponding to the simulated transition is $i = 0$, then the above described movement of the transition symbol is only possible if there are no c_i counter symbols present in region 2.

If the system simulates a transition $t_f : (q \rightarrow f, X)$ to the final state f, it may enter the terminating phase. In this phase the following rules are used.

$$\begin{aligned} R_1^{ter} &= \{(f_1, in; t_f', out), (\bar{f}_2, in; f_1', out), (\bar{f}_2, in; \bar{f}_2, out), \\ &\quad (f_2, in; \bar{f}_2, out), (f_3, in; f_2', out), (f_4, in; f_3', out)\}, \end{aligned}$$

$$\begin{aligned} R_2^{ter} &= \{(f_1, in), (C, in; f_1', out), (f_2, in; 0_0, out), (0_0, in; c_0, out), \\ &\quad (0_0, in; f_2', out), (f_3, in; C, out), (f_3', out), (f_4, in; \infty_1, out)\}, \end{aligned}$$

$$\begin{aligned} R_3^{ter} &= \{(f_1, in; f_1', out), (C, in), (C, out), (c_0, in; C, out), \\ &\quad (f_2, in; f_2', out), (f_3, in; f_3', out), (f_4, in; rs' out), \\ &\quad (f_4, in; rs, out), (f_4, out)\}. \end{aligned}$$

During the terminating phase, symbols f_1, f_2, f_3, and f_4 travel through the system, each performing a specific task. First, after a transition symbol t_f' is present in region 1, f_1 is imported into the system. It moves to region 3 where f_1' is released which moves to region 1 bringing C to region 2. The task of C is to move all c_0 counter symbols corresponding to the output counter to region 3. This may take several steps, so f_1' is exchanged with $\bar{f}_2$ in region 1, and $\bar{f}_2$ can move in and out of the system for an arbitrary amount of time. When $\bar{f}_2$ is exchanged with f_2, the termination process continues. The travel of f_2 is only possible if there is no c_0 present in region 2. In this case, after f_2' leaves the system, f_3 is imported. While f_3 moves through the system it removes C from region 2, so no further movement of possibly newly appearing c_0 will be allowed to region 3, then, when f_3' leaves the system, f_4 is introduced. When f_4 moves to region 2, it removes ∞_1, thus removes the infinite loop, and then it also removes the remaining transition symbols from region 3. When all of these symbols are out of region 3, the system stops working, having only the counter symbols plus five other symbols, f_1, f_2, f_3, ∞_3, and I_2 in region 3, the output region, thus producing a result $x \in \mathbb{N}$ for some $(x - 5) \in L(M)$. □

5.2.2 Remarks

We have shown how to simulate counter automata using P systems with minimal symport/antiport and three membranes which improved the previously known best result of (Frisco, 2004) stating that four membranes are sufficient to reach this power. The above described simulation first appeared in (Vaszil, 2005b), but the research in this area continued intensively also after its publication. In (Alhazov et al., 2005a) the need for the five superfluous symbols was eliminated (thus, $\mathbb{N}RE$ was characterized instead of $\mathbb{N}_5RE$), then in (Alhazov et al., 2006) two membranes were shown to be sufficient. This last result is optimal, since one membrane with minimal symport/antiport only generates finite sets (see (Alhazov et al., 2005b)).

5.3 P colonies

P colonies were introduced in (Kelemen et al., 2004) as a class of very simple membrane systems similar to the so-called colonies of simple formal grammars, (Kelemen and Kelemenová, 1992).

Due to the inspiration from the cell structure and functioning, and from the cell cooperation in tissues, P colonies model a population of cells, which "live" together in a common shared environment, and process symbolic objects appearing in this environment. The objects correspond to (or in certain extent model the) chemical compounds – ions, molecules, macromolecules – dealt with in biochemistry.

In order to have as simple agents as possible, the complexity and the capabilities of the cells are very restricted. Each cell is associated with a multiset of *objects* ("chemicals") present inside it, and with a set of *rules* for processing these objects ("reactions"). In each moment, only a fixed number of objects (two or three objects usually) are allowed to be inside any cell. Moreover, the rules available to any cell are either of the form $a \to b$, specifying that an internal object a is transformed into an internal object b, or of the form $c \leftrightarrow d$, specifying the fact that an internal object c is sent out of the cell, to the environment, in exchange of the object d, which was present in the environment and is now brought inside the cell (rules of the latter type also correspond to the phenomenon of *antiport* already seen in the previous section.

With each cell, we associate a set of programs composed of rules as above.

In the case of systems consisting of cells with only two objects inside, each program has two rules; when considering cells with three objects inside, then the programs have three rules. The rules of the program must be applied in parallel to the objects in the cell. Thus, a cell containing the objects a, c will contain the objects b, d after applying these rules.

The choice of these kinds of rules points to the observation that alive cells/beings are continuously exchanging objects with their environment and, at the same time, they evolve internally. P colonies use the simplest forms of exchanges (object-to-object) and the simplest form of internal evolution (one-object-transformation). The cells of a P colony execute a computation by synchronously applying their programs to objects inside the cells and outside in the environment. Communication between the cells is only possible indirectly through the environment which is common to all of them. When a halting configuration is reached, that is, when no more rules can be applied, the result of the computation is read as the number of certain types of objects present in the environment.

Now we recall the definition of a P colony from (Kelemen et al., 2004).

Definition 5.3.1. A *P colony* is a construct

$$\Pi = (V, e, o_f, I_E, C_1, \dots, C_n, k),\ n \geq 1,$$

where V is an alphabet (its elements are called *objects*), e (the environmental object) and o_f (the final object) are two distinguished objects of V, I_E is a multiset over V, containing the objects initially present in the environment, $C_1, \dots, C_n$ are the *cells* of the colony, and k is the number of objects which are allowed to be present in the cells.

Each cell $C_i,\ 1 \leq i \leq n$, is a pair $C_i = (O_i, P_i)$, where O_i is a multiset over $\{e\}$ having the same cardinality $card(O_i) = k$ for all $i,\ 1 \leq i \leq n$ (the initial state of the cell), and P_i is a finite set of *programs*; each program being a set of rules of the forms $a \to b$ (internal point mutation), $c \leftrightarrow d$ (one object exchange with the environment), $c \leftrightarrow d/c' \leftrightarrow d'$ (checking rule for one object exchange with the environment), or $c \leftrightarrow d/a \to b$ (checking rule for one object exchange with the environment or internal point mutation), where $a, b, c, d, c', d' \in V$. The programs contain one rule for each element of O_i, thus, the number of rules of a program coincides with k, the cardinality of O_i, $1 \leq i \leq n$.

Usually (as in (Csuhaj-Varjú et al., 2006a) and in (Kelemen et al., 2004)), the multiset I_E only contains an infinite supply of e objects which are present in the environment, but sometimes (as in (Csuhaj-Varjú et al., 2006b) and in the next section), we allow I_E to contain other symbols as well, because we would like to allow the initialization of the colony by also placing objects different from e in the environment.

The programs of the cells are used in the non-deterministic maximally parallel way usual in membrane computing: in each time unit, each cell which can use one of its programs should use one. When using a program, each of its rules must be applied to distinct objects of the cell. In this way, we get transitions among the configurations of the colony. A sequence of transitions is a *computation*. A computation is halting if it reaches a configuration where no cell can use any program. The result of a halting computation is the number of copies of the object o_f present in the environment in the halting configuration. Initially, the environment contains I_E, and the cells also contain k copies of e inside.

Because of the non-determinism in choosing the programs, several computations can be obtained from a given initial configuration, hence with a P colony Π we can associate a set of numbers computed by all possible halting computations of Π.

For a P colony $\Pi = (V, e, o_f, I_E, C_1, \dots, C_n, k)$ as above, a configuration can be formally written as an $(n+1)$-tuple

$$(w_1, \dots, w_n; w_E),$$

where w_i represents the multiset of objects from cell C_i, $1 \leq i \leq n$ ($w_i \in V^k$), and $w_E \in (V - \{e\})^*$ represents the multiset of those objects in the environment which are different from the "background" object e.

The initial configuration is $(e^k, \dots, e^k; \bar{I}_E)$ where $\bar{I}_E \in (V - \{e\})^*$ is a finite multiset containing the objects from I_E which are different from e.

Let the programs of each P_i be labeled in a one-to-one manner by labels in the set $lab(P_i)$ in such a way that $lab(P_i) \cap lab(P_j) = \emptyset$ for $i \neq j$, $1 \leq i, j \leq n$. For a rule r and a multiset $w \in V^*$, let $left(r, w) = a$ and $right(r, w) = b$ if $a \in w$ and r is a point mutation rule $r = (a \to b)$ or a checking rule $r = (c \leftrightarrow d/a \to b)$ and $d \notin w$, and let $left(r, w) = right(r, w) = \varepsilon$ otherwise. Let also, for a rule r and a multiset $w \in V^*$ $export(r, w) = c$ and $import(r, w) = d$ if $d \in w$ and r is an exchange rule $r = (c \leftrightarrow d)$ or a checking rule $r = (c \leftrightarrow d/c' \leftrightarrow d')$. If r is a checking rule as above with $d \notin w$ but $d' \in w$, then let $export(r, w) = c'$, $import(r, w) = d'$. Let $export(r, w) = import(r, w) = \varepsilon$ in all other cases. For a program p and any $\alpha \in \{left, right, export, import\}$, let $\alpha(p, w) = \bigcup_{r \in p} \alpha(r)$.

Definition 5.3.2. A *transition* from a configuration to another is denoted as

$$(w_1, \dots, w_n; w_E) \Rightarrow (w'_1, \dots, w'_n; w'_E)$$

where the following conditions are satisfied: There is a set of program labels P with $|P| \leq n$, such that $p, p' \in P$, $p \neq p'$, $p \in lab(P_j)$ implies $p' \notin lab(P_j)$, and for each $p \in P$, $p \in lab(P_j)$, $left(p, w_E) \cup export(p, w_E) = w_j$, and $\bigcup_{p \in P} import(p, w_E) \subseteq w_E$. Furthermore, the chosen set P is maximal, that is, if any other program $r \in \bigcup_{1 \leq i \leq n} lab(P_i)$, $r \notin P$, is added to P, then the conditions above are not satisfied.

Now, for each j, $1 \leq j \leq n$, for which there exists a $p \in P$ with $p \in lab(P_j)$, let

$$w'_j = right(p, w_E) \cup import(p, w_E).$$

If there is no $p \in P$ with $p \in lab(P_j)$ for some j, $1 \leq j \leq n$, then let

$$w'_j = w_j,$$

and moreover, let

$$w'_E = w_E - \bigcup_{p \in P} import(p, w_E) \cup \bigcup_{p \in P} export(p, w_E).$$

A configuration is halting if the set of program labels P satisfying the conditions above cannot be chosen to be other than the empty set, $\emptyset$.

Definition 5.3.3. The set of nonnegative integers *computed* by a P colony Π is defined as

$$N(\Pi) = \{|v_E|_{o_f} \mid (w_1, \dots, w_n, I_E) \Rightarrow^* (v_1, \dots, v_n, v_E)\}$$

where $(w_1, \dots, w_n, I_E)$ is the initial configuration, $(v_1, \dots, v_n, v_E)$ is a halting configuration, and $\Rightarrow^*$ denotes the reflexive and transitive closure of $\Rightarrow$.

Notation. The family of all sets of numbers computed as above by P colonies with k-objects ($k = 2, 3$) of degree at most $n \geq 1$ having at most $h \geq 1$ programs in the cells without using checking rules, and j programs altogether, is denoted by PCOL$(k, n, h, \textit{no-check})$. If checking rules are allowed, then we write *check* instead of *no-check*; thus, for instance, PCOL$(2, n, h, \textit{check})$ will be the family of numbers computed by two-objects P colonies with at most n cells, each having at most h programs where the use of checking rules is allowed. If we need to express that the number of cells or the number of programs is unbounded, we use $*$ instead of the respective parameter, n or h.

5.3.1 The number of cells and programs

It was shown in (Kelemen et al., 2004) and (Csuhaj-Varjú et al., 2006a) that P colonies are able to compute any recursively enumerable set of numbers, even in the situation when the starting configuration contains one certain object only: an infinite number of copies of it in the environment, and two or three copies of the same object inside the cells.

From (Csuhaj-Varjú et al., 2006a), we have that

$$\mathrm{PCOL}(2, *, 4, \mathit{check}) = \mathrm{PCOL}(3, *, 3, \mathit{check}) = \mathbb{N}\mathrm{RE}.$$

Even one cell is enough, if it may have an arbitrarily large number of programs, that is,

$$\mathrm{PCOL}(2, 1, *, \mathit{check}) = \mathbb{N}\mathrm{RE}.$$

Similar results were also obtained without the use of checking rules. In this case we have

$$\mathrm{PCOL}(2, *, 8, \mathit{no\text{-}check}) = \mathrm{PCOL}(3, *, 7, \mathit{no\text{-}check}) = \mathbb{N}\mathrm{RE}.$$

In this section we show that if the environment is initialized by a finite multiset of objects before the computation begins, then P colonies generate any recursively enumerable set of numbers in such a way that the number of cells and the number of programs in each cell are simultaneously bounded. The values of the bounds depend on the type of rules used and seem to suggest a trade-off between the number of necessary cells and the number of necessary programs in each cell. The results demonstrate that one cell with a bounded, but fairly large amount of programs might possess the power of Turing machines, which power can also be reached by several cells and a significantly lower number of programs in each cell.

To achieve our goal, we will show how to simulate a universal register machine, that is, a fixed machine which is able to mimic the computations of any other register machine if it is given an appropriate program.

In (Korec, 1996) several results on small universal register machines are presented. Those machines all have a small number of registers and a small number of instructions, the exact numbers depending on the chosen set of instructions and the chosen notion of universality. They compute functions of non-negative integers by having the argument of the function in one of the registers before the computation starts, and obtaining the result of the function in an other register after a halting computation. The universal machines have eight registers, and they can simulate the computation of any register machine M with the help of a "program", an integer $code(M) \in \mathbb{N}$ coding the particular machine M. If $code(M)$ is placed in the second register and an argument $x \in \mathbb{N}$ is placed in the third register, then the universal machine simulates the computation of M by halting if and only if M halts, and by producing the same result in its first register as M produces in its output register after a halting computation.

Since these machines are defined to compute functions of non-negative integers and work in such a way that the argument of the function is initially present in the third register, we need to modify them in order to conform to the number generating definition of register machines we have given in Chapter 2. Thus, we add a new start label l'_0 and a separate non-deterministic add instruction, $l'_0 : (\texttt{nADD}(r_3), l'_0, l_0)$, to produce an argument $x \in \mathbb{N}$ in the third register before the actual computation begins, that is, to make the resulting universal machine generate any value from the range of the function computed by the simulated register machine. We summarize the results we use from (Korec, 1996) in the following theorem.

Theorem 5.3.1. (Korec, 1996) *Let $\mathbb{M}$ be the set of register machines. Then, there is a register machine U_{32} with eight registers and a recursive function $g : \mathbb{M} \to \mathbb{N}$ such that for each $M \in \mathbb{M}$, $N(M) = N(U_{32}(g(M)))$, where $N(U_{32}(g(M)))$ denotes the set of numbers computed by U_{32} with initially containing $g(M)$ in the second register.*

The machine U_{32} has one HALT *instruction labeled by l_h, one instruction of the type* nADD *labeled by l_0, and $8 + 11 + 13 = 32$ instructions of the types* ADD*,* SUB*, and* CHECK*, respectively.*

Moreover, the machine either halts using the HALT *instruction and having the result of the computation in the first register, or its computation goes on infinitely.*

In the following, we will simulate U_{32} to obtain bounds on the size parameters of universal P colonies. First, we present new register machine instructions which can be obtained by combining elements of the basic instruction set defined above and which make the presentation of U_{32} more efficient from our point of view.

- $l_i : (\texttt{DADD}(r, s), l_j)$ - *double add*: Add 1 to register r and register s, then go to the instruction with label l_j.

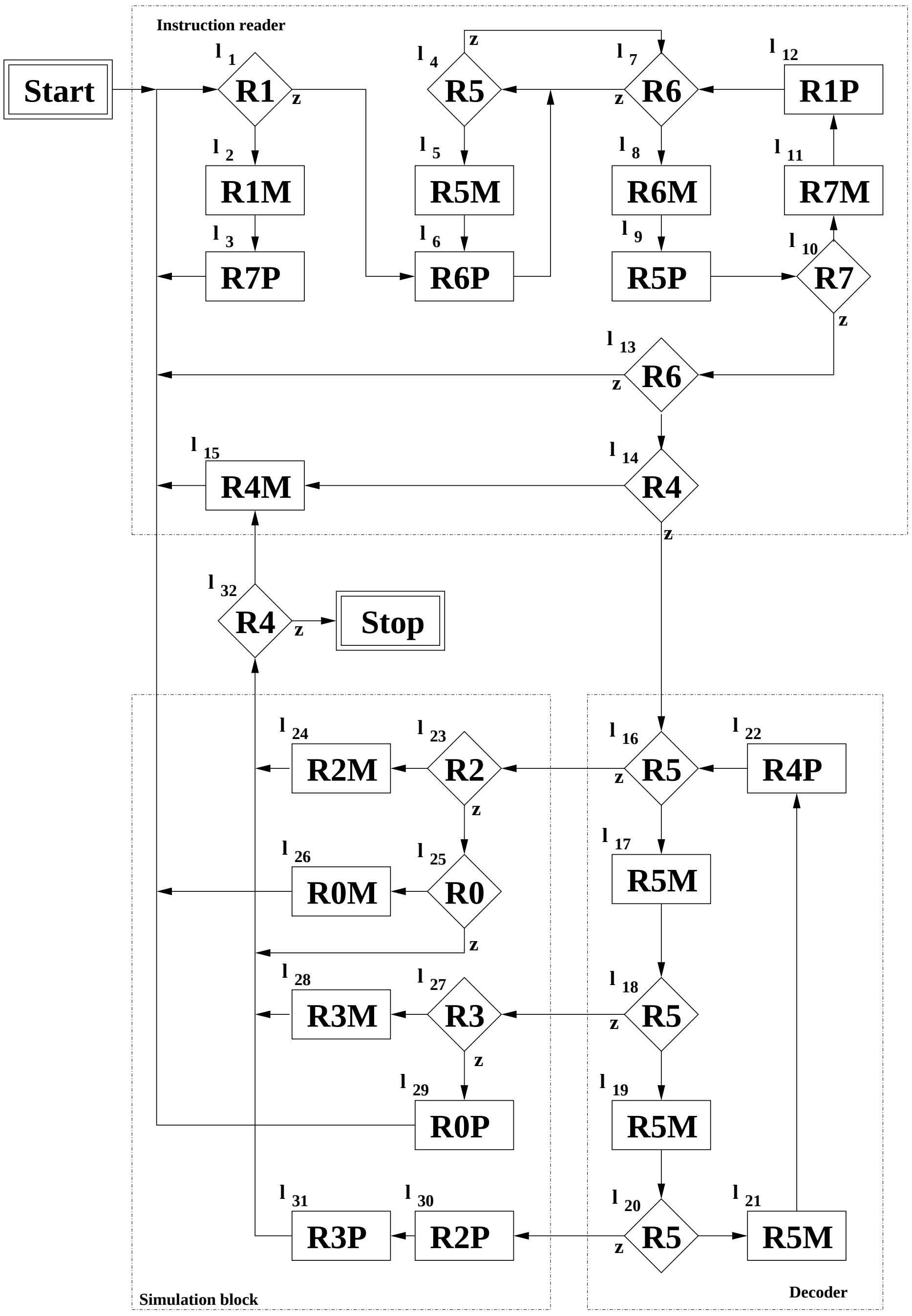

Figure 5.1: The flowchart of the universal machine U_{32} from (Korec, 1996). A rectangle shaped box containing RjP or RjM corresponds to the instruction $l_i : (\mathtt{ADD}(j), l_k)$ or $l_i : (\mathtt{SUB}(j), l_k)$, respectively, where l_i is the label of the box and the arrow leads to the instruction labeled with l_k. A rhombus shaped box containing Rj corresponds to $l_i : (\mathtt{CHECK}(j), l_k, l_l)$ where the rhombus is labeled with l_i, the arrow marked with "z" leads to instruction l_k, and the unmarked arrow leads to the box l_l.

- l_i : ($\texttt{CHECKSUB}(r), l_j, l_k$) - *zero check and subtract*: If register r is non-empty, then subtract 1 from it and go to the instruction with label l_j, otherwise go to the instruction with label l_k.
- l_i : ($\texttt{CHECKSUBADD}_1(r, s), l_j, l_k$) - *zero check, subtract, add, first variant*: If register r is non-empty, then subtract 1 from it, add 1 to register s, and go to the instruction with label l_j, otherwise go to the instruction with label l_k.
- l_i : ($\texttt{CHECKSUBADD}_2(r, s), l_j, l_k$) - *zero check, subtract, add, second variant*: If register r is non-empty, then subtract 1 from it and go to the instruction with label l_j, otherwise add 1 to register s and go to the instruction with label l_k.

By looking at the flowchart on Figure 5.1, we might observe how the instructions above can be used to obtain the same functioning as U_{32}. Thus, as a direct consequence of Theorem 5.3.1, we present the register machine U_{15} as follows. We use the name U_{15} to emphasize the fact that this machine has fifteen instructions (besides the nondeterministic add and the halt instruction).

Observation 5.3.2. *Let $\mathbb{M}$ be the set of register machines. Then, there is a register machine U_{15} with eight registers and a recursive function $g : \mathbb{M} \to \mathbb{N}$ such that for each $M \in \mathbb{M}$, $N(M) = N(U_{15}(g(M)))$, where $N(U_{15}(g(M)))$ denotes the set of numbers computed by U_{15} with initially containing $g(M)$ in the second register.*

The machine U_{15} has one HALT *instruction labeled by l_h, one instruction of the type* nADD *labeled by l_0, and furthermore, one* ADD *instruction labeled by l_1, one* CHECK *instruction labeled by l_2, six* CHECKSUB *instructions labeled by l_i, $3 \le i \le 8$, one* DADD *instruction labeled by l_9, five* $\texttt{CHECKSUBADD}_1$ *instructions labeled by l_i, $10 \le i \le 14$, and one* $\texttt{CHECKSUBADD}_2$ *instruction labeled by l_{15}.*

Moreover, the machine either halts using the HALT *instruction and having the result of the computation in the first register, or its computation goes on infinitely.*

To see how the instructions of U_{32} can be combined to obtain U_{15}, consider the three instructions q_1 : ($\texttt{CHECK}(r_1), q_6, q_2$), q_2 : ($\texttt{SUB}(r_1), q_3$), and q_3 : ($\texttt{ADD}(r_7), q_1$) from the flowchart in Figure 5.1. These are equivalent with the combined instruction q_1 : ($\texttt{CHECKSUBADD}_1(r_1, r_7), q_6, q_1$).

Definition 5.3.4. We call a two-object P colony restricted, if the programs contain exactly one point mutation rule of the form $a \to b$, and either one exchange rule of the form $c \leftrightarrow d$, or one checking rule of the form $c \leftrightarrow d/c' \leftrightarrow d'$.

Notation. The class of sets generated by two-objects P colonies with restricted programs is denoted by $\mathrm{PCOL}(2\mathrm{R}, n, h, X)$ where n and h denotes the number of cells and programs and $X \in \{check, no\text{-}check\}$, as before.

First we consider restricted and non-restricted two object P colonies with checking rules.

Theorem 5.3.3.

$$\mathrm{PCOL}(2\mathrm{R}, 16, 6, check) = \mathrm{PCOL}(2, 16, 5, check) = \mathbb{N}\mathrm{RE}.$$

Proof. Let $L \in \mathbb{N}\mathrm{RE}$ and let M be a register machine with $L = N(M)$. Consider the universal register machine $U_{15} = (8, H, l_0, l_h, R)$ from Observation 5.3.2. By placing the code of M (which is the value of $g(M)$) in the second register, it computes $N(M)$, producing the result in its first register. We show how to construct P colonies which simulate the computation of U_{15}.

Consider a P colony $(V, e, 1, I_E, C_1, \dots, C_n)$ where V contains the special object e, two symbols l and l' for each instruction label $l \in H$ of the universal machine, and one symbol j, $1 \le j \le 8$, for each register. (For the sake of simplicity, we denote the number $j \in \mathbb{N}$ and the object $j \in V$ in the same way.) The number of j symbols in the environment corresponds to the value of register j. Thus, the initial contents of the environment, I_E is $g(M)$ copies of the object 2 plus the label of the initial instruction l_0, and the result of the computation can be read as the number of 1 objects corresponding to the value of the first register in a halting configuration.

The P colonies we are going to construct contains one cell for each instruction.

A nondeterministic add instruction $l_i : (\mathtt{nADD}(r_i), l_{j_i}, l_{k_i})$ is simulated by increasing the number of objects corresponding to the value of register r_i by one and by changing the instruction label l_i present in the environment to l_{j_i} or l_{k_i}. This can be done with five programs as follows.

$$\begin{aligned} P_i = \{ & \langle e \to r_i; e \leftrightarrow l_i \rangle,\ \langle l_i \to l_{j_i}; r_i \leftrightarrow e \rangle,\ \langle e \to e; l_{j_i} \leftrightarrow e \rangle, \\ & \langle l_i \to l_{k_i}; r_i \leftrightarrow e \rangle,\ \langle e \to e; l_{k_i} \leftrightarrow e \rangle \}. \end{aligned}$$

An add instruction $l_i : (\texttt{ADD}(r_i), l_{j_i})$ is simulated by increasing the number of objects corresponding to the value of register r_i by one and by changing the instruction label to l_{k_i}.

$$P_i = \{\langle e \to r_i; e \leftrightarrow l_i\rangle,\ \langle l_i \to l_{j_i}; r_i \leftrightarrow e\rangle,\ \langle e \to e; l_{j_i} \leftrightarrow e\rangle\}.$$

An add instruction $l_i : (\texttt{DADD}(r_i, s_i), l_{j_i})$ is simulated by increasing the number of objects corresponding to the values of registers s_i and s_i by one, and by changing the instruction label to l_{j_i}.

$$\begin{aligned} P_i = \{&\langle e \to r_i; e \leftrightarrow l_i\rangle,\ \langle l_i \to s_i; r_i \leftrightarrow e\rangle,\ \langle e \to l_{j_i}; s_i \leftrightarrow e\rangle, \\ &\langle e \to e; l_{j_i} \leftrightarrow e\rangle\}. \end{aligned}$$

An instruction of type $l_i : (\texttt{CHECK}(r_i), l_{j_i}, l_{k_i})$ is simulated by six restricted programs as follows.

$$\begin{aligned} P_i = \{&\langle e \to e; e \leftrightarrow l_i\rangle,\ \langle l_i \to l'_{j_i}; e \leftrightarrow r_i/e \leftrightarrow e\rangle,\ \langle l'_{j_i} \to l_{j_i}; r_i \leftrightarrow e\rangle, \\ &\langle e \to e; l_{j_i} \leftrightarrow e\rangle,\ \langle l'_{j_i} \to l_{k_i}; e \leftrightarrow e\rangle,\ \langle e \to e; l_{k_i} \leftrightarrow e\rangle\}. \end{aligned}$$

If we allow non-restricted programs, we can replace P_i with

$$\begin{aligned} P'_i = \{&\langle e \to e; e \leftrightarrow l_i\rangle,\ \langle l_i \to l_{j_i}; e \leftrightarrow r_i/e \to l_{k_i}\rangle,\ \langle l_{j_i} \leftrightarrow e; r_i \leftrightarrow e\rangle, \\ &\langle l_{j_i} \to e; l_{k_i} \leftrightarrow e\rangle\} \end{aligned}$$

which achieve the same effect as P_i with four programs.

An instruction $l_i : (\texttt{CHEKSUB}(r_i), l_{j_i}, l_{k_i})$ is simulated with five restricted programs

$$\begin{aligned} P_i = \{&\langle e \to e; e \leftrightarrow l_i\rangle,\ \langle l_i \to l_{j_i}; e \leftrightarrow r_i/e \leftrightarrow e\rangle,\ \langle r_i \to e; l_{j_i} \leftrightarrow e\rangle, \\ &\langle l_{j_i} \to l_{k_i}; e \leftrightarrow e\rangle,\ \langle e \to e; l_{k_i} \leftrightarrow e\rangle\}, \end{aligned}$$

or with four non-restricted programs

$$\begin{aligned} P'_i = \{&\langle e \to e; e \leftrightarrow l_i\rangle,\ \langle l_i \to l_{j_i}; e \leftrightarrow r_i/e \to l_{k_i}\rangle,\ \langle r_i \to e; l_{j_i} \leftrightarrow e\rangle, \\ &\langle l_{j_i} \to e; l_{k_i}; \leftrightarrow e\rangle\rangle\}, \end{aligned}$$

by exchanging the label l_i to l_{j_i} and decreasing the number of r_i objects by one, or if the number of r_i objects is zero, then exchanging l_i to l_{k_i}.

An instruction of type $l_i : (\texttt{CHEKSUBADD}_1(r_i, s_i), l_{j_i}, l_{k_i})$ is simulated with six restricted programs

$$\begin{aligned} P_i = \{&\langle e \to e; e \leftrightarrow l_i\rangle,\ \langle l_i \to l_{j_i}; e \leftrightarrow r_i/e \leftrightarrow e\rangle,\ \langle r_i \to s_i; l_{j_i} \leftrightarrow e\rangle, \\ &\langle e \to e; s_i \leftrightarrow e\rangle,\ \langle l_{j_i} \to l_{k_i}; e \leftrightarrow e\rangle,\ \langle e \to e; l_{k_i} \leftrightarrow e\rangle\}, \end{aligned}$$

or with five non-restricted programs

$$\begin{aligned} P'_i &= \{\langle e \to s_i; e \leftrightarrow l_i\rangle,\ \langle l_i \to l_{j_i}; s_i \leftrightarrow r_i/s_i \to e\rangle,\ \langle r_i \to e; l_{j_i} \leftrightarrow e\rangle, \\ &\quad \langle l_{j_i} \to l_{k_i}; e \leftrightarrow e\rangle,\ \langle e \to e; l_{k_i} \leftrightarrow e\rangle\}, \end{aligned}$$

by exchanging the label l_i to l_{j_i} and an r_i object to s_i, or if the number of r_i objects is zero, then exchanging l_i to l_{k_i}.

An instruction of type $l_i : (\texttt{CHEKSUBADD}_2(r_i, s_i), l_{j_i}, l_{k_i})$ is simulated with six restricted programs

$$\begin{aligned} P_i &= \{\langle e \to e; e \leftrightarrow l_i\rangle,\ \langle l_i \to l_{j_i}; e \leftrightarrow r_i/e \leftrightarrow e\rangle,\ \langle r_i \to e; l_{j_i} \leftrightarrow e\rangle, \\ &\quad \langle l_{j_i} \to s_i; e \leftrightarrow e\rangle,\ \langle e \to l_{k_i}; s_i \leftrightarrow e\rangle,\ \langle e \to e; l_{k_i} \leftrightarrow e\rangle, \}, \end{aligned}$$

or with five non-restricted programs

$$\begin{aligned} P'_i &= \{\langle e \to s_i; e \leftrightarrow l_i\rangle,\ \langle l_i \to l_{j_i}; s_i \leftrightarrow r_i/s_i \leftrightarrow e\rangle,\ \langle r_i \to e; l_{j_i} \leftrightarrow s_i\rangle, \\ &\quad \langle l_{j_i} \to l_{k_i}; e \leftrightarrow e\rangle,\ \langle e \to e; l_{k_i} \leftrightarrow e\rangle\}, \end{aligned}$$

by exchanging the label l_i to l_{j_i} and removing an object r_i, or if the number of r_i objects is zero, then increasing the number of s_i objects and exchanging l_i to l_{k_i}.

There are no programs for the halting label l_h, thus, its appearance ends the computation which otherwise never stops.

For the simulation of U_{15} with restricted programs, consider the P colony $\Pi = (V, e, 1, I_E, C_0, \ldots, C_{15})$ with V and I_E as above, and one cell with the programs P_0 for the initial `nADD` instruction labeled with l_0 which fills the input register, one cell with P_1, for the simulation of the `ADD` instruction, one cell with P_2 for the simulation of the `CHECK` instruction, six cells with P_i, $3 \leq i \leq 8$, for the simulation of the `CHECKSUB` instructions, one cell with P_9 for the simulation of the double add `DADD`, five cells with P_i, $10 \leq i \leq 14$, for the simulation of the $\texttt{CHECKSUBADD}_1$, and one cell with P_{15} for the simulation of the $\texttt{CHECKSUBADD}_2$ instruction. This gives us $1+1+1+6+1+5+1 = 16$ cells with at most 6 programs, thus the first part of our statement is proved.

To simulate U_{15} with non-restricted programs, we can take the modified P colony $\Pi' = (V, e, 1, I_E, C_0, C_1, C'_2, C'_3, \ldots, C'_8, C_9, C'_{10}, \ldots, C'_{15})$, where $C'_i = (O_i, P'_i)$, $i \in \{2, 3, \ldots, 8, 10, \ldots, 15\}$, which gives us 16 cells with at most 5 programs. □

Now we show that by increasing the number of programs, one cell is sufficient to generate any set in $\mathbb{N}RE$, even with restricted programs.

Theorem 5.3.4.

$$\mathrm{PCOL}(2\mathrm{R}, 1, 114, \mathit{check}) = \mathbb{N}\mathrm{RE}.$$

Proof. Similarly to the proof of Theorem 5.3.3, we show how to construct a P colony which simulates the computation of the universal register machine U_{15} from Observation 5.3.2.

Let the `nADD` instruction of U_{15} be labeled by l_0, the `ADD` instruction by l_1, the `CHECK` instruction by l_2, the six `CHECKSUB` instructions by $l_3, \dots, l_8$, the `DADD` instruction by l_9, the five $\mathtt{CHECKSUBADD}_1$ instructions by $l_{10}, \dots, l_{14}$, and the $\mathtt{CHECKSUBADD}_2$ instruction by l_{15}.

Let $\Pi = (V, e, 1, I_E, C)$ be a P colony where V contains the special objects e, t; the symbol l_i for each instruction of U_{15}, that is, for $0 \leq i \leq 15$; the symbols l'_i for the `nADD`, `ADD` and `DADD` instructions, that is, for $i \in \{0, 1, 9\}$; the symbols l''_i for the `CHECK`, `CHECKSUB`, $\mathtt{CHECKSUBADD}_1$, and $\mathtt{CHECKSUBADD}_2$ instructions, that is, for $i \in \{3, \dots, 8, 10, \dots, 15\}$; and one symbol j for each register j, $1 \leq j \leq 8$. The initial contents of the environment, I_E is a number of copies of the object 2 for initializing the contents of the second register, plus one copy of the symbols l'_i for $i \in \{0, 1, 9\}$. The result of a computation can be read in a halting configuration as the number of 1 objects in the environment.

The computation starts by producing an arbitrary number of copies of the double primed instruction labels l''_i, $i \in \{3, \dots, 8, 10, \dots, 15\}$, and the introduction of the initial instruction label l_0. This is achieved with the following 25 programs.

$$\begin{aligned} P_{ini} &= \{\langle e \to l''_3; e \leftrightarrow e\rangle,\ \langle e \to l''_{15}; l''_{15} \leftrightarrow e\rangle,\ \langle e \to l_0; l''_{15} \leftrightarrow e\rangle\} \cup \\ &\quad \{\langle e \to l''_i; l''_i \leftrightarrow e\rangle,\ \langle e \to l''_{i+1}; l''_i \leftrightarrow e\rangle, \mid \\ &\quad\ i \in \{3, \dots, 7, 10, \dots, 14\}\} \cup \\ &\quad \{\langle e \to l''_8; l''_8 \leftrightarrow e\rangle,\ \langle e \to l''_{10}; l''_8 \leftrightarrow e\rangle\}. \end{aligned}$$

Now, for the instruction $l_0 : (\mathtt{nADD}(r_0), l_{i_0}, l_{j_0})$, we have the following four programs

$$\begin{aligned} P_0 &= \{\langle l_0 \to l'_0; e \leftrightarrow e\rangle,\ \langle e \to r_0; l'_0 \leftrightarrow l'_0\rangle,\ \langle l'_0 \to l_{i_0}; r_0 \leftrightarrow e\rangle, \\ &\quad \langle l'_0 \to l_{j_0}; r_0 \leftrightarrow e\rangle\}. \end{aligned}$$

For the instruction $l_1 : (\mathtt{ADD}(r_1), l_{i_1})$, we have

$$P_1 = \{\langle l_1 \to l'_1; e \leftrightarrow e\rangle,\ \langle e \to r_1; l'_1 \leftrightarrow l'_1\rangle,\ \langle l'_1 \to l_{i_1}; r_1 \leftrightarrow e\rangle\}.$$

For the instruction $l_2 : (\mathtt{CHECK}(r_2), l_{i_2}, l_{j_2})$ we have

$$\begin{aligned} P_2 &= \{\langle l_2 \to l_2'; e \leftrightarrow r_2/e \leftrightarrow e\rangle, \langle r_2 \to r_2; l_2' \leftrightarrow l_2''\rangle, \langle l_2'' \to l_{i_2}; r_2 \leftrightarrow e\rangle, \\ &\quad \langle l_2' \to l_{j_2}; e \leftrightarrow e\rangle\}. \end{aligned}$$

For the instructions $l_i : (\mathtt{CHECKSUB}(r_i), l_{j_i}, l_{k_i}),\ 3 \le i \le 8$, we have the programs

$$\begin{aligned} P_i &= \{\langle l_i \to l_i'; e \leftrightarrow r_i/e \leftrightarrow e\rangle, \langle r_i \to e; l_i' \leftrightarrow l_i''\rangle, \langle l_i'' \to l_{j_i}; e \leftrightarrow e\rangle, \\ &\quad \langle l_i' \to l_{k_i}; e \leftrightarrow e\rangle\}. \end{aligned}$$

For the instruction $l_9 : (\mathtt{DADD}(r_i, s_i), l_{i_9})$, we have

$$\begin{aligned} P_9 &= \{\langle l_9 \to l_9'; e \leftrightarrow e\rangle,\ \langle e \to r_i; l_9' \leftrightarrow l_9'\rangle,\ \langle l_9' \to l_9'; r_i \leftrightarrow e\rangle, \\ &\quad \langle e \to s_i; l_9' \leftrightarrow l_9'\rangle,\ \langle l_9' \to l_{i_9}; s_i \leftrightarrow e\rangle\}. \end{aligned}$$

For the instructions $l_i : (\mathtt{CHECKSUBADD}_1(r_i, s_i), l_{j_i}, l_{k_i}),\ 10 \le i \le 14$, we have the programs

$$\begin{aligned} P_i &= \{\langle l_i \to l_i'; e \leftrightarrow r_i/e \leftrightarrow e\rangle, \langle r_i \to s_i; l_i' \leftrightarrow l_i''\rangle, \langle l_i'' \to l_{j_i}; s_i \leftrightarrow e\rangle, \\ &\quad \langle l_i' \to l_{k_i}; e \leftrightarrow e\rangle\}, \end{aligned}$$

and for the instruction $l_{15} : (\mathtt{CHECKSUBADD}_2(r_{15}, s_{15}), l_{i_{15}}, l_{j_{15}})$, we have the programs

$$\begin{aligned} P_{15} &= \{\langle l_{15} \to l_{15}'; e \leftrightarrow r_{15}/e \leftrightarrow e\rangle, \langle r_{15} \to e; l_{15}' \leftrightarrow l_{15}''\rangle, \\ &\quad \langle l_{15}'' \to l_{i_{15}}; e \leftrightarrow e\rangle, \langle e \to s_{15}; l_{15}' \leftrightarrow l_{15}'\rangle, \langle l_{15}' \to l_{j_{15}}; s_{15} \leftrightarrow e\rangle\}. \end{aligned}$$

Moreover, in order to ensure that the cell exchanges primed objects from inside with double primed objects from the environment, we also consider the programs

$$P_{trap} = \{\langle l_i' \to t; r_i \leftrightarrow e\rangle,\ \langle t \to t; e \leftrightarrow e\rangle \mid i \in \{3, \ldots, 8, 10, \ldots, 15\}\ \}$$

where t is a trap-object and r_i corresponds, for $3 \le i \le 8$, to the register used by the CHECKSUB instructions labeled with l_i, and for $10 \le i \le 15$, to the register which is checked for zero by the $\mathtt{CHECKSUBADD}_1$ or $\mathtt{CHECKSUBADD}_2$ instructions labeled with l_i. If an exchange of the primed and double primed labels cannot be realized during the simulation of these instructions because the number of double primed symbols produced in the initial phase is insufficient, then the trap-object is introduced and the program $\langle t \to t; e \leftrightarrow e\rangle$ provides an "infinite loop", preventing the halting of the computation.

Considering the P colony above with $C = (O, P)$ where $P = \bigcup_{i=0}^{15} P_i \cup P_{ini} \cup P_{trap}$, we need 25 programs for the initial phase, four programs for the nADD instruction, three for the ADD instruction, four for the CHECK instruction, $6{\cdot}4 = 24$ for the CHECKSUB instructions, five for the DADD instruction, $5{\cdot}4 = 20$ for the $\mathtt{CHECKSUBADD}_1$ instructions, five for the $\mathtt{CHECKSUBADD}_2$ instruction, and we have $2 \cdot 12 = 24$ programs in P_{trap}. This gives us $25 + 4 + 3 + 4 + 24 + 5 + 20 + 5 + 24 = 114$ programs in total, so our statement is proved. □

Now we show how the use of checking rules can be avoided. First we consider the case of two objects P colonies with restricted and with non-restricted programs.

Theorem 5.3.5.

$$\mathrm{PCOL}(2\mathrm{R}, 16, 10, \textit{no-check}) = \mathrm{PCOL}(2, 16, 9, \textit{no-check}) = \mathbb{N}\mathrm{RE}.$$

Proof. We show how the universal register machine U_{15} from Observation 5.3.2 can be simulated.

Consider $\Pi = (V, e, 1, I_E, C_0, \ldots, C_{15})$ from the proof of Theorem 5.3.3 and let $\Pi' = (V', e, 1, I_E, C'_0, \ldots C'_{16})$ where $V' = V \cup \{l', l''\}$. Now we construct the cells C'_i, $0 \leq i \leq 16$, which achieve the same effect as the 16 cells of Π. The nADD, ADD, and DADD instructions are simulated in Π without checking rules, thus we can take

$$C'_i = C_i, \text{ for } i \in \{0, 1, 9\},$$

without any change at all.

For the simulation of the other instructions, we need an additional cell, C'_{16}, with two programs

$$P'_{16} = \{\langle e \to l''; e \leftrightarrow l' \rangle,\ \langle l' \to e; l'' \leftrightarrow e \rangle\}.$$

This cell produces the symbol l'' from the symbol l' in two computational steps.

For the instruction $l_2 : (\mathtt{CHECK}(r_2), l_{i_2}, l_{j_2})$ we need the programs

$$\begin{aligned} P'_2 \ = \ & \{\langle e \to l'; e \leftrightarrow l_2 \rangle,\ \langle l_2 \to l'_2; l' \leftrightarrow e \rangle,\ \langle l'_2 \to l''_2; e \leftrightarrow r_2 \rangle, \\ & \langle l''_2 \to l_{i_2}; r_2 \leftrightarrow e \rangle,\ \langle l_{i_2} \to l_{i_2}; e \leftrightarrow l'' \rangle,\ \langle l'_2 \to l_{j_2}; e \leftrightarrow l'' \rangle, \\ & \langle l'' \to e; l_{i_2} \leftrightarrow e \rangle,\ \langle l'' \to e; l_{j_2} \leftrightarrow e \rangle\}, \end{aligned}$$

having at most eight restricted programs. The interplay of these programs and the programs in P'_{16} produces the desired checking effect, the symbol

l_2 is exchanged to l_{i_2} if there is at least one r_2 symbol in the environment, otherwise l_{j_2} is released.

For the instructions $l_i : (\texttt{CHECKSUB}(r_i), l_{j_i}, l_{k_i}),\ 3 \le i \le 8$, we also need P'_{16}, plus cells with at most nine programs as follows, for each $i,\ 3 \le i \le 8$,

$$\begin{aligned} P'_i \;=\; & \{\langle e \to l'; e \leftrightarrow l_i\rangle,\ \langle l_i \to l'_i; l' \leftrightarrow e\rangle,\ \langle l'_i \to l''_i; e \leftrightarrow r_i\rangle, \\ & \langle l''_i \to l_{j_i}; r_i \leftrightarrow e\rangle,\ \langle l_{j_i} \to l_{j_i}; e \leftrightarrow l''\rangle,\ \langle l'_i \to l_{k_i}; e \leftrightarrow l''\rangle, \\ & \langle l'' \to e; l_{j_i} \leftrightarrow r_i\rangle,\ \langle r_i \to e, e \leftrightarrow e\rangle,\ \langle l'' \to e; l_{k_i} \leftrightarrow e\rangle\}. \end{aligned}$$

These programs work together with P'_{16} very similarly to the ones above. However, they not only check, but if possible, also decrease the number of register symbols in the environment.

For the instructions $l_i : (\texttt{CHECKSUBADD}_1(r_i, s_i), l_{j_i}, l_{k_i}),\ 10 \le i \le 14$, we also have cells with nine programs,

$$\begin{aligned} P'_i \;=\; & \{\langle e \to l'; e \leftrightarrow l_i\rangle,\ \langle l_i \to l'_i; l' \leftrightarrow e\rangle,\ \langle l'_i \to l''_i; e \leftrightarrow r_i\rangle, \\ & \langle l''_i \to l_{j_i}; r_i \leftrightarrow e\rangle,\ \langle l_{j_i} \to l_{j_i}; e \leftrightarrow l''\rangle,\ \langle l'_i \to l_{k_i}; e \leftrightarrow l''\rangle, \\ & \langle l'' \to s_i; l_{j_i} \leftrightarrow r_i\rangle,\ \langle r_i \to e, s_i \leftrightarrow e\rangle,\ \langle l'' \to e; l_{k_i} \leftrightarrow e\rangle\}, \end{aligned}$$

and for the $l_{15} : (\texttt{CHECKSUBADD}_2(r_{15}, s_{15}), l_{i_{15}}, l_{j_{15}})$, instruction we have

$$\begin{aligned} P'_{15} \;=\; & \{\langle e \to l'; e \leftrightarrow l_{15}\rangle,\ \langle l_{15} \to l'_{15}; l' \leftrightarrow e\rangle,\ \langle l'_{15} \to l''_{15}; e \leftrightarrow r_{15}\rangle, \\ & \langle l''_{15} \to l_{i_{15}}; r_{15} \leftrightarrow e\rangle,\ \langle l_{i_{15}} \to l_{i_{15}}; e \leftrightarrow l''\rangle,\ \langle l'_{15} \to s_{15}; e \leftrightarrow l''\rangle, \\ & \langle l'' \to e; l_{i_{15}} \leftrightarrow r_{15}\rangle,\ \langle r_{15} \to e; e \leftrightarrow e\rangle,\ \langle l'' \to l_{j_{15}}; s_{15} \leftrightarrow e\rangle, \\ & \langle e \to e; l_{j_{15}} \leftrightarrow e\rangle\}, \end{aligned}$$

which is a set with ten programs.

Thus, if we have

$$C'_i = (ee, P'_i),\ i \in \{2, \ldots, 8, 10, \ldots, 16\}$$

in Π', then we can simulate U_{15} with 16 cells, each having at most 10 restricted programs without checking rules, and altogether 131 programs.

If the use of non-restricted programs is allowed, we can decrease the number of programs of certain cells.

The simulation of the $l_i : (\texttt{CHECKSUB}(r_i), l_{j_i}, l_{k_i}),\ 3 \le i \le 8$, instructions can be done with eight programs as follows.

$$\begin{aligned} P''_i \;=\; & \{\langle e \to l'; e \leftrightarrow l_i\rangle,\ \langle l_i \to l'_i; l' \leftrightarrow e\rangle,\ \langle l'_i \to l''_i; e \leftrightarrow r_i\rangle, \\ & \langle r_i \to e; l''_i \to l_{j_i}\rangle,\ \langle l_{j_i} \to l_{j_i}; e \leftrightarrow l''\rangle,\ \langle l'_i \to l_{k_i}; e \leftrightarrow l''\rangle, \\ & \langle l'' \to e; l_{j_i} \leftrightarrow e\rangle,\ \langle l'' \to e; l_{k_i} \leftrightarrow e\rangle\}. \end{aligned}$$

For the simulation of the instructions $l_i : (\texttt{CHECKSUBADD}_1(r_i, s_i), l_{j_i}, l_{k_i}),\ 10 \leq i \leq 14$, we can use the programs in

$$\begin{aligned} P''_i &= \{\langle e \to l'; e \leftrightarrow l_i\rangle,\ \langle l_i \to l'_i; l' \leftrightarrow e\rangle,\ \langle l'_i \to l''_i; e \leftrightarrow r_i\rangle, \\ &\quad \langle r_i \to s_i; l''_i \to l_{j_i}\rangle,\ \langle l_{j_i} \to l_{j_i}; s_i \leftrightarrow l''\rangle,\ \langle l'_i \to l_{k_i}; e \leftrightarrow l''\rangle, \\ &\quad \langle l'' \to e; l_{j_i} \leftrightarrow e\rangle,\ \langle l'' \to e; l_{k_i} \leftrightarrow e\rangle\}, \end{aligned}$$

and for the simulation of the $l_{15} : (\texttt{CHECKSUBADD}_2(r_{15}, s_{15}), l_{i_{15}}, l_{j_{15}})$, instruction,

$$\begin{aligned} P''_{15} &= \{\langle e \to l'; e \leftrightarrow l_{15}\rangle,\ \langle l_{15} \to l'_{15}; l' \leftrightarrow e\rangle,\ \langle l'_{15} \to l''_{15}; e \leftrightarrow r_{15}\rangle, \\ &\quad \langle r_{15} \to e; l''_{15} \to l_{i_{15}}\rangle,\ \langle l_{i_{15},1} \to l_{i_{15}}; e \leftrightarrow l''\rangle,\ \langle l'_{15} \to l_{j_{15}}; e \leftrightarrow l''\rangle, \\ &\quad \langle l'' \to e; l_{i_{15}} \leftrightarrow e\rangle,\ \langle l'' \to s_{15}; l_{j_{15}} \to l_{j_{15}}\rangle,\ \langle s_{15} \leftrightarrow e; l_{j_{15}} \leftrightarrow e\rangle\}. \end{aligned}$$

Now, if we have

$$C''_i = \begin{cases} (ee, P''_i) & \text{for } i \in \{3, \ldots, 8, 10, \ldots, 16\}, \\ C'_i & \text{otherwise,} \end{cases}$$

then in $\Pi'' = (V', e, r_1, I_E, C''_0, \ldots C''_{16})$ we can simulate U_{15} with 16 cells, each having at most nine restricted program without checking rules, and altogether with 116 programs. □

The number of programs in the cells can be decreased if, instead of two, we allow three objects in each cell, and thus, three instructions in each program.

Theorem 5.3.6.

$$\text{PCOL}(3, 16, 7, \textit{no-check}) = \mathbb{N}\text{RE}.$$

Proof. Consider $\Pi' = (V', e, 1, I_E, C'_0, \ldots C'_{16})$ simulating U_{15} from Observation 5.3.2 as described in the of the proof of the previous theorem. Now define $C''_i = (eee, P''_i),\ 0 \leq i \leq 16$, where for $i \in \{0, 1, 9, 16\}$, P''_i is obtained from P'_i by adding a rule $e \to e$ to each program. The rest of the cells is constructed as follows.

For $l_2 : (\texttt{CHECK}(r_2), l_{i_2}, l_{j_2})$, we have

$$\begin{aligned} P''_2 &= \{\langle e \to l'; e \to l'_{i_2}; e \leftrightarrow l_2\rangle,\ \langle l_2 \to e; l' \leftrightarrow e; l'_{i_2} \to l'_{i_2}\rangle, \\ &\quad \langle l'_{i_2} \to l''_{i_2}; e \leftrightarrow r_2; e \leftrightarrow e\rangle,\ \langle r_2 \to e; l''_{i_2} \to l_{i_2}; e \leftrightarrow l''\rangle, \\ &\quad \langle l'_{i_2} \to e; e \leftrightarrow l''; e \to l_{j_2}\rangle,\ \langle l'' \to e; l_{i_2} \leftrightarrow e, e \to e\rangle, \\ &\quad \langle l'' \to e; l_{j_2} \leftrightarrow e; e \to e\rangle\}. \end{aligned}$$

For the rules $l_i : (\mathtt{CHECKSUB}(r_i), l_{j_i}, l_{k_i}),\ 3 \leq i \leq 8$, let

$$\begin{aligned} P_i'' \ &= \ \{\langle e \to l'; e \to l'_{j_i}; e \leftrightarrow l_i\rangle,\ \langle l_i \to e; l' \leftrightarrow e; l'_{j_i} \to l'_{j_i}\rangle, \\ &\quad \langle l'_{j_i} \to l''_{j_i}; e \leftrightarrow r_i; e \to e\rangle,\ \langle r_i \to e; l''_{j_i} \to l_{j_i}; e \leftrightarrow l''\rangle, \\ &\quad \langle l'_{j_i} \to e; e \leftrightarrow l''; e \to l_{j_i}\rangle,\ \langle l'' \to e; l_{j_i} \leftrightarrow e, e \to e\rangle, \\ &\quad \langle l'' \to e; l_{j_i} \leftrightarrow e; e \to e\rangle\}. \end{aligned}$$

For the instructions $l_i : (\mathtt{CHECKSUBADD}_1(r_i, s_i), l_{j_i}, l_{k_i}),\ 10 \leq i \leq 14$, let

$$\begin{aligned} P_i'' \ &= \ \{\langle e \to l'; e \to l'_{j_i}; e \leftrightarrow l_i\rangle,\ \langle l_i \to e; l' \leftrightarrow e; l'_{j_i} \to l'_{j_i}\rangle, \\ &\quad \langle l'_{j_i} \to l''_{j_i}; e \leftrightarrow r_i; e \to s_i\rangle,\ \langle r_i \to e; l''_{j_i} \to l_{j_i}; s_i \leftrightarrow l''\rangle, \\ &\quad \langle l'_{j_i} \to e; e \leftrightarrow l''; e \to l_{k_i}\rangle,\ \langle l'' \to e; l_{j_i} \leftrightarrow e, e \to e\rangle, \\ &\quad \langle l'' \to e; l_{k_i} \leftrightarrow e; e \to e\rangle\}, \end{aligned}$$

and for $l_{15} : (\mathtt{CHECKSUBADD}_2(r_{15}, s_{15}), l_{i_{15}}, l_{j_{15}})$, let

$$\begin{aligned} P_{15}'' \ &= \ \{\langle e \to l'; e \to l'_{i_{15}}; e \leftrightarrow l_{15}\rangle,\ \langle l_{15} \to e; l' \leftrightarrow e; l'_{i_{15}} \to l'_{i_{15}}\rangle, \\ &\quad \langle l'_{i_{15}} \to l''_{i_{15}}; e \leftrightarrow r_{15}; e \to e\rangle,\ \langle r_{15} \to e; l''_{i_{15}} \to l_{i_{15}}; e \leftrightarrow l''\rangle, \\ &\quad \langle l'_{i_{15}} \to s_{15}; e \leftrightarrow l''; e \to l_{j_{15}}\rangle,\ \langle l'' \to e; l_{i_{15}} \leftrightarrow e, e \to e\rangle, \\ &\quad \langle l'' \to e; l_{i_{15}} \leftrightarrow e; s_{15} \leftrightarrow e\rangle\}. \end{aligned}$$

Thus, if we have $C_i'' = (eee, P_i'')$ for $i \in \{3, \ldots, 8, 10, \ldots, 16\}$, and consider the P colony $\Pi'' = (V'', e, a_1, I_E, C_0'', \ldots C_{15}'')$ with $V'' = V' \cup \{l_i', l_i'' \mid i \in \{2, \ldots, 8, 10, \ldots, 15\}\ \}$, then we have a system with 16 cells, at most seven programs in each cell, so our statement is proved. $\square$

5.3.2 Simplifying the programs: P colonies with insertion/deletion

In this section we continue with the investigation of a possible simplification of P colonies. We consider systems where the cells can only remove objects from the environment or insert objects into the environment. We will use so called insertion and deletion programs for this purpose.

Definition 5.3.5. P colonies with capacity of two may have

- *deletion programs*, these are of the form $\langle a, in; bc \to d\rangle$ with $a, b, c, d \in V$, specifying that if bc is present inside the cell and a is present in the environment, then the objects inside are changed to d and a is brought in (a is "deleted" from the environment), or

- *insertion programs*, these are of the form $\langle a, out; b \to cd\rangle$ with $a, b, c, d \in V$, specifying that if ab is inside the cell, then a is sent out (a is "inserted" into the environment) and b is changed to cd.

Now we continue with the investigation of two object P colonies with insertion-deletion programs. It is not too difficult to see that if we allow a cell to contain both types of programs, then we can simulate the other types of programs in two steps, thus, it is more interesting to consider P colonies having cells which contain either insertion or deletion programs, but not both types at the same time.

Definition 5.3.6. A two object P colony $\Pi = (V, e, o_f, I_E, C_1, \ldots, C_n, k)$ with $C_i = (O_i, P_i)$ is called a *P colony with senders and consumers*, if for each i, $1 \leq i \leq n$, P_i contains either insertion programs or deletion programs, but not both. A *sender* is a cell with only insertion programs, a *consumer* is a cell with only deletion programs.

Notation. Let us denote by $\mathrm{PCOL}(2, n, h, s\text{-}c)$ the class of sets of numbers generated by P colonies with senders and consumers having at most $n \geq 1$ cells with at most $h \geq 1$ program each. We use $*$ in the notation if we need to express that one of the parameters is unbounded.

Example 5.3.1.
(a) A P colony with one sender cell can generate the Parikh set of a regular language $L \subseteq T^*$. Let $G = (N, T, P, S)$ be a regular grammar such that $L(G) = L$.

For generating the Parikh vectors of the words in L, we use, for each $S \to aB$ of P, the programs $\langle e, out; e \to eS\rangle$, $\langle e, out; S \to aB\rangle$ and then $\langle x, out; A \to aB\rangle, x \in T$ for every $A \to aB$ in P. Finally, for every rule of the form $A \to a$ we need $\langle x, out; A \to aF\rangle, x \in T$, $\langle a, out; F \to FF\rangle$, where $F \notin T \cup N$.

(b) A P colony with one consumer cell can "consume" the Parikh set of a regular language L. To see this, let $M = (Q, T, \delta, q_0, F)$ be a deterministic finite automaton such that $L(M) = L$.

We need the program $\langle e, in; ee \to q_0\rangle$, and to every transition $\delta(q_i, a) = q_j$ in M, we introduce $\langle a, in; xq_i \to q_j\rangle, x \in T \cup \{e\}$. If $q_j \in F$ in $\delta(q_i, a) = q_j$ we have to add the programs $\langle a, in; xq_i \to F\rangle, x \in T$, where $F \notin Q \cup T$.

Now we show that three cells, one sender and two consumers are sufficient to generate all recursively enumerable sets of nonnegative integers. We will simulate a register machine in the proof, but let us first consider the following.

Remark 5.3.1. We can replace the instructions of type $l_i : (\texttt{CHECK}(r), l_j, l_k)$ and $l_i : (\texttt{SUB}(r), l_j)$ of a register machine $M = (m, H, l_0, l_h, P)$ (see Chapter 2 for the definition) with instructions of the type $l_i : (\texttt{CHECKSUB}(r), l_j, l_k)$ in such a way, that the modified machine generates the same language as M. (The effect of the CHECKSUB instruction is the same as in the previous section: if the value stored in register r is not zero, then it is decremented and the machine continues with instruction l_j, otherwise, if register r is empty, it is left unchanged and the next instruction is l_k.)

To see that the replacement is possible, notice that the effect of l_i : $(\texttt{SUB}(r), l_j)$ can be obtained by $l_i : (\texttt{CHECKSUB}(r), l_j, l_j)$, and that the effect of $l_i : (\texttt{CHECKSUB}(r), l_j, l_k)$ is the same as the effect of $l_i : (\texttt{CHECK}(r), l_k, l'_j)$ and $l'_j : (\texttt{SUB}(r), l_j)$ (where l'_l is a new label).

Theorem 5.3.7. $\mathrm{PCOL}(2, 3, *, s\text{-}c) = \mathbb{N}\mathrm{RE}$.

Proof. Consider an m-register machine $M = (m, H, l_0, l_h, P)$, $m \geq 1$ with CHECKSUB instructions instead of CHECK and SUB. (see Remark 5.3.1). We simulate M by representing the content of the register i by the number of copies of a specific object a_i in the environment. We construct the P colony $\Pi = (V, e, o_f, I_E, C_1, C_2, C_3)$ with

$$\begin{aligned} V &= \{e, l, l', l'', l''', l^{iv}, l^{v}, \bar{l}, \bar{\bar{l}} \mid l \in H\} \cup \{a_i \mid 1 \leq i \leq m\} \cup \\ &\quad \{K, T_1, T_2, T_3, T_4, T_5\}, \\ o_f &= a_i \text{ where register } i \text{ is the output register}, \\ C_i &= (ee, P_i) \text{ for } 1 \leq i \leq 3, \text{ and} \end{aligned}$$

I_E contains only an infinite supply of the object e.

P colony Π starts its computation in the initial configuration $(ee, ee, ee; \varepsilon)$. We initialize the computation by generating the initial label l_0 with a program from P_1, $\langle e, out; e \to l_0 l_0\rangle \in P_1$ obtaining $(l_0 l_0, ee, ee; \varepsilon)$.

The simulation of an instruction with label l_i starts from a configuration $(l_i l_i, ee, ee; w)$ where $w \in V^*$, the multiset of objects in the environment, represents the counter contents of M.

In the following, for easier readability, instead of listing all the programs of a cell explicitly, we will present them in the form of several tables, each table containing only some of the programs, those which are necessary for the execution of a certain computational task.

To simulate an nADD instruction, we use the programs of P_1 and P_3. For each $l_i, l_j, l_k \in H$ with l_i being the label of an instruction $l_i : (\mathtt{nADD}(r), l_j, l_k)$, we have the following programs.

P_1		P_3	
$i_1 :$	$\langle l_i, out; l_i \to a_r l_j\rangle$	$i_1 :$	$\langle l_i, in; ee \to T_1\rangle$
$i_2 :$	$\langle l_i, out; l_i \to a_r l_k\rangle$	$i_2 :$	$\langle e, in; l_i T_1 \to e\rangle$
$i_3 :$	$\langle a_r, out; l_j \to l_j l_j\rangle$	$i_3 :$	$\langle l_i, in; \bar{\bar{l}}_i T_5 \to T_1\rangle$
$i_4 :$	$\langle a_r, out; l_k \to l_k l_k\rangle$		

Using these programs, we obtain a sequence of configurations

$$(l_i l_i, ee, ee; w) \Rightarrow (a_r l, ee, ee; l_i w) \Rightarrow (ll, ee, l_i T_1; a_r w)$$

where l is the label of the next instruction, that is, we either have $(l_j l_j, ee, l_i T_1; a_r w)$, or the configuration $(l_k l_k, ee, l_i T_1; a_r w)$. The contents of cell C_3, $l_i T_1$, will change in the next step to ee independently of the several ways of the continuation of the computation, as we shall see later.

The program labeled with i_3 is used if the instruction simulated before l_i was a CHECKSUB instruction (see below). In this case, the configuration in which the simulation of l_i starts is $(l_i l_i, ee, \bar{l}_i T_4; \bar{\bar{l}}_i w)$ and we need the steps $(l_i l_i, ee, \bar{l}_i T_4; \bar{\bar{l}}_i w) \Rightarrow (a_r l, ee, \bar{\bar{l}}_i T_5; l_i w) \Rightarrow (ll, ee, l_i T_1; a_r w)$ and program i_3 to obtain the same configuration as before.

To simulate a deterministic ADD instruction $l_i : (\mathtt{nADD}(r), l_j)$, we omit the programs denoted with i_2 and i_4 from the set P_1.

Now we show how to simulate a CHECKSUB instruction. For each $l_j, l_k, l_l \in H$ with l_j being the label of an instruction $l_j : (\mathtt{CHECKSUB}(r), l_k, l_l)$, and for all labels $l_s \in H$, we have the programs as seen in Figure 5.2.

To show how the programs of Figure 5.2 simulate the execution of the given CHECKSUB instruction, we use tables where each row represents a configuration of the system (with the columns corresponding to the contents of the cells, the environment, and the programs which are applied) to obtain the configuration of the next row of the table. To save space, we also use the sign "/" to separate the different possible multisets which might appear in the same column of a given table position.

First we consider the case when register r is not empty, that is, when there is at least one object a_r present in the environment. We see that starting with a configuration where C_1 contains the objects $l_j l_j$ and the environment contains a_r, in six steps we obtain a configuration where the object a_r is

P_1	P_2		P_3	
$j_1 : \langle l_j, out; l_j \to l'_j l'_j \rangle$	$j_1 :$	$\langle l_j, in; ee \to e \rangle$	$j_1 :$	$\langle l'_j, in; ee \to T_1 \rangle$
$j_2 : \langle l'_j, out; l'_j \to l''_j l''_j \rangle$	$j_2 :$	$\langle a_r, in; el_j \to e \rangle$	$j_2 :$	$\langle e, in; l'_j T_1 \to T_2 \rangle$
$j_3 : \langle l''_j, out; l''_j \to \underline{l'''_j} \underline{l^{iv}_j} \rangle$	$j_3 :$	$\langle l''_j, in; el_j \to e \rangle$	$j_3 :$	$\langle l''_j, in; eT_2 \to T_3 \rangle$
$j_4 : \langle l'''_j, out; l^{iv}_j \to \bar{l}_k \bar{l}_k \rangle$	$j_4 :$	$\langle l'''_j, in; a_r e \to e \rangle$	$j_{4,s} :$	$\langle \bar{l}_s, in; l''_j T_3 \to T_4 \rangle$
$j_5 : \langle l^{iv}_j, out; l'''_j \to \bar{l}_l \bar{l}_l \rangle$	$j_5 :$	$\langle e, in; l'''_j e \to e \rangle$	$j_{5,s} :$	$\langle \bar{l}_s, in; eT_2 \to T_4 \rangle$
$j_6 : \langle \bar{l}_k, out; \bar{l}_k \to \bar{\bar{l}}_k \bar{\bar{l}}_k \rangle$	$j_6 :$	$\langle l^{iv}_j, in; a_r e \to K \rangle$	$j_{6,s} :$	$\langle \bar{\bar{l}}_s, in; \bar{l}_s T_4 \to T_5 \rangle$
$j_7 : \langle \bar{\bar{l}}_k, out; \bar{\bar{l}}_k \to l_k l_k \rangle$	$j_7 :$	$\langle e, in; l^{iv}_j K \to K \rangle$	$j_{7,s} :$	$\langle e, in; \bar{\bar{l}}_s T_5 \to e \rangle$
$j_8 : \langle \bar{l}_l, out; \bar{l}_l \to \bar{\bar{l}}_l \bar{\bar{l}}_l \rangle$	$j_8 :$	$\langle e, in; eK \to K \rangle$		
$j_9 : \langle \bar{\bar{l}}_l, out; \bar{\bar{l}}_k \to l_l l_l \rangle$	$j_9 :$	$\langle l'''_j, in; l''_j e \to K \rangle$		
	$j_{10} :$	$\langle e, in; l'''_j K \to K \rangle$		
	$j_{11} :$	$\langle l^{iv}_j, in; l''_j e \to e \rangle$		
	$j_{12} :$	$\langle e, in; l^{iv}_j e \to e \rangle$		

Figure 5.2: Programs to simulate an instruction $l_j : (\texttt{CHECKSUB}(r), l_k, l_l)$.

removed from the environment, and C_1 either contains the label of the next instruction l_k, or because of the presence of program j_8, in P_2, the computation will never be able to halt.

	configuration of Π				programs to be applied		
	C_1	C_2	C_3	Env	P_1	P_2	P_3
1.	$l_j l_j$	ee	?	$a_r w'$	j_1	$-$	?
2.	$l'_j l'_j$	ee	?	$l_j a_r w''$	j_2	j_1	?
3.	$l''_j l''_j$	$l_j e$	ee	$l'_j a_r w$	j_3	j_2	j_1
4.	$\underline{l'''_j} \underline{l^{iv}_j}$	$a_r e$	$l'_j T_1$	$l''_j w$	j_4/j_5	$-$	j_2
5.	$\bar{l}_k \bar{l}_k / \bar{l}_l \bar{l}_l$	$a_r e$	eT_2	$(l'''_j / l^{iv}_j) l''_j w$	j_6/j_8	j_4/j_6	j_3
6.	$\bar{\bar{l}}_k \bar{\bar{l}}_k / \bar{\bar{l}}_l \bar{\bar{l}}_l$	$l'''_j e / l^{iv}_j K$	$l''_j T_3$	$(\bar{l}_k / \bar{l}_l) w$	j_7/j_9	j_5/j_7	$j_{4,k}/j_{4,l}$
7.	$l_k l_k / l_l l_l$	ee/eK	$(\bar{l}_k / \bar{l}_l) T_4$	$(\bar{\bar{l}}_k / \bar{\bar{l}}_l) w$	k_1/l_1	$-/j_8$	$j_{6,k}/j_{6,l}$
8.	$l'_k l'_k / l'_l l'_l$	ee/eK	$(\bar{\bar{l}}_k / \bar{\bar{l}}_l) T_5$	$(l_k / l_l) w$	k_2/l_2	k_1/j_8	$j_{7,k}/j_{7,l}$
9.	$l''_k l''_k / l''_l l''_l$	$(l_k / l_l) e / eK$	ee	$(l'_k / l'_l) w$	k_3/l_3	k_2/j_8	j_1

Now we show the simulation of the $l_j : (\texttt{CHECKSUB}(r), l_k, l_l)$ instruction when there is no object a_r is present in the environment, that is, when register r is empty. In this case, similarly to the previous one, we either get the objects $l_k l_k$ in the cell C_1, or the computation will not be able to halt.

	configuration of Π				programs to be applied		
	C_1	C_2	C_3	Env	P_1	P_2	P_3
1.	$l_j l_j$	ee	$?$	w	j_1	$-$	$?$
2.	$l'_j l'_j$	ee	$?$	$l_j w$	j_2	j_1	$?$
3.	$l''_j l''_j$	$l_j e$	ee	$l'_j w$	j_3	$-$	j_1
4.	$l'''_j l^{iv}_j$	$l_j e$	$l'_j T_1$	$l''_j w$	j_4/j_5	j_3	j_2
5.	$\bar{l}_k \bar{l}_k / \bar{l}_l \bar{l}_l$	$l''_j e$	eT_2	$(l'''_j / l^{iv}_j)w$	j_6/j_8	j_9/j_{11}	$-$
6.	$\bar{\bar{l}}_k \bar{\bar{l}}_k / \bar{\bar{l}}_l \bar{\bar{l}}_l$	$l'''_j K / l^{iv}_j e$	eT_2	$(\bar{l}_k/\bar{l}_l)w$	j_7/j_9	j_{10}/j_{12}	$j_{5,k}/j_{5,l}$
7.	$l_k l_k / l_l l_l$	eK/ee	$(\bar{l}_k/\bar{l}_l)T_4$	$(\bar{\bar{l}}_k/\bar{\bar{l}}_l)w$	k_1/l_1	$j_8/-$	$j_{6,k}/j_{6,l}$
8.	$l'_k l'_k / l'_l l'_l$	eK/ee	$(\bar{\bar{l}}_k/\bar{\bar{l}}_l)T_5$	$(l_k/l_l)w$	k_2/l_2	j_8/k_1	$j_{7,k}/j_{7,l}$
9.	$l''_k l''_k / l''_l l''_l$	$eK/(l_k/l_l)e$	ee	$(l'_k/l'_l)w$	k_3/l_3	j_8/k_2	j_1

The programs to be applied and the objects contained by the cell C_3 in row 1. and row 2. of the tables above depend on the instruction l_i which was simulated before l_j. If l_i is an `ADD` instruction, then we have $l_i T_1$ in the first row, and applying the program i_2 from P_3 we get ee in the second row, where no program is applied until the next step. Also, $w = w' = w''$ in this case.

If l_i is a `CHECKSUB` instruction, then (as we can also see from row 7. and row 8.) the contents of the cell C_3 is $\bar{l}_j T_4$ and $\bar{\bar{l}}_j T_5$ in the first two rows where the programs $i_{6,j}$ and $i_{7,j}$ are applied. In this case $w'' = \bar{\bar{l}}_j w$, and $w' = w$.

As we have seen above, the P colony successfully simulates each instruction of M and since there is no program to process l_h, the label of the halt instruction, it also halts when the computation of M is finished. It is also easy to see that M and Π compute the same set of non-negative integers. $\square$

5.3.3 Simplifying the cells: The number of symbols inside

Up to now we have studied P colonies with two or three objects inside the cells. It is a natural question to ask whether the number of objects inside the cells can be decreased from three or two to one without losing the computational power of the framework. In this section we show that this is indeed possible, that is, P colonies can generate any recursively enumerable set even if checking rules are not allowed and the number of objects inside is the least possible, just one.

Notation. We denote by PCOL$(1, n, h, \mathit{check})$ and PCOL$(1, n, h, \mathit{no\text{-}check})$ the classes of sets of numbers generated by P colonies with at most $n \geq 1$ cells of capacity one, having at most $h \geq 1$ programs associated to a cell which contain or do not contain checking rules, respectively. If a numerical parameter is unbounded, we use $*$ in the notation.

In (Cienciala et al., 2007) it was shown that if checking rules are allowed to be used, then all recursively enumerable sets of vectors can even be generated by P colonies with capacity one, that is,

$$\mathrm{PCOL}(1, 4, *, \mathit{check}) = \mathbb{N}\mathrm{RE}.$$

In the following we show that P colonies with six components generate all recursively enumerable sets even if checking rules are not used.

Theorem 5.3.8. PCOL$(1, 6, *, \mathit{no\text{-}check}) = \mathbb{N}$RE.

Proof. We construct a P colony simulating the computations of a register machine. Let us consider an m-register machine $M = (m, H, l_0, l_h, P)$ with `CHEKSUB` instructions (see Remark 5.3.1) and represent the content of the register i by the number of copies of a specific object a_i in the environment. We construct the P colony $\Pi = (V, e, o_f, I_E, C_1, \ldots, C_6)$ with:

$$\begin{aligned} V &= \{e, l_i, l_i', l_i'', \bar{l}_i, K_i, L_i, L_i', L_i'', L_i''', E_i, F_i, \$_i \mid \text{for each } l_i \in H\} \cup \\ &\quad\ \{a_i, a_{i,j} \mid 1 \leq i \leq m,\ 1 \leq j \leq |H|\} \cup \{D, D', T\}, \\ o_f &= a_i \text{ where register } i \text{ is the output register}, \\ C_i &= (e, P_i),\ \text{ for } 1 \leq i \leq 6,\ \text{ and} \end{aligned}$$

I_E contains only an infinite supply of the object e.

Because initially there are only copies of e in the environment and inside the cells, we have to initialize the simulation of the computation of M by generating the initial the label l_0, and an arbitrary number of l_i', l_i'' for all $l_i \in H$. These symbols are generated by C_1 and C_2 with the following programs:

$$P_1 \supset \{\langle e \to l'_r\rangle, \langle l'_r \leftrightarrow e\rangle, \langle e \to l''_r\rangle, \langle l''_r \leftrightarrow e\rangle \mid l_r \in H\} \cup \{\langle e \leftrightarrow D'\rangle, \langle D' \to l_0\rangle, \langle l_0 \leftrightarrow D\rangle\},$$

$$P_2 \supset \{\langle e \to D'\rangle, \langle D' \to D'\rangle, \langle D' \leftrightarrow l'_1\rangle, \langle l'_1 \to D\rangle, \langle D \leftrightarrow l''_1\rangle\}.$$

With these programs, from the configuration $(e, e, e, e, e, e; \varepsilon)$, we obtain $(D, l''_1, e, e, e, e; l_0 w)$ where the environment contains the label of the initial instruction, l_0, and w, a multiset of primed and double primed instruction labels.

The rest of the programs will be presented through tables, in a similar fashion to section 5.3.2, each table containing only some of the programs, those which are necessary for the execution of the given computational task.

To simulate the instruction $l_i : (\texttt{nADD}(r), l_j, l_k)$, cells C_1 and C_3 cooperate to add one copy of object a_r and object l_j or l_k to the environment.

P_1				P_3			
i_1 :	$\langle D \leftrightarrow a_{r,i}\rangle$	i_6 :	$\langle K_k \to l_k\rangle$	i_1 :	$\langle e \leftrightarrow l_i\rangle$	i_6 :	$\langle l'_i \to K_k\rangle$
i_2 :	$\langle a_{r,i} \to a_r\rangle$	i_7 :	$\langle l_j \leftrightarrow D\rangle$	i_2 :	$\langle l_i \to a_{r,i}\rangle$	i_7 :	$\langle K_j \leftrightarrow e\rangle$
i_3 :	$\langle a_r \leftrightarrow K_j\rangle$	i_8 :	$\langle l_k \leftrightarrow D\rangle$	i_3 :	$\langle a_{r,i} \leftrightarrow l'_i\rangle$	i_8 :	$\langle K_k \leftrightarrow e\rangle$
i_4 :	$\langle a_r \leftrightarrow K_k\rangle$			i_4 :	$\langle a_{r,i} \to t\rangle$	i_9 :	$\langle t \to t\rangle$
i_5 :	$\langle K_j \to l_j\rangle$			i_5 :	$\langle l'_i \to K_j\rangle$		

It is not difficult to follow how the interplay of these two cells produce the configuration $(D, l''_1, e, e, e, e; l_j a_r w')$ or $(D, l''_1, e, e, e, e; l_k a_r w')$ from a configuration $(D, l''_1, e, e, e, e; l_i w)$ where w, w' are multisets of l'_i, l''_i for $l_i \in H$ and a_r, $1 \leq r \leq m$. If there is no l'_i present in the environment when the program i_3 of cell C_3 should be used, then the programs i_4 and i_9 do not allow the halting of the computation.

To simulate a deterministic `ADD` instruction $l_i : (\texttt{nADD}(r), l_j)$, we need to omit the programs denoted with i_4, i_6, i_8 from the set P_1, and i_6, i_8 from the set P_3.

For each subtract instruction $l_f : (\texttt{CHECKSUB}(r), l_g, l_n)$ we have the programs in P_1, P_4, P_5 and in P_6 as indicated in Figure 5.3.

In the following table we show how a subtract instruction can be simulated by the programs above. Since C_2 and C_3 cannot apply any of their rules in any step of the following simulation, we omit them from the table. The multiset of objects in the environment is denoted by $[\ldots]$, and for now we assume that

P_1		P_4		P_5		P_6	
$f_1:$	$\langle D \leftrightarrow L_f\rangle$	$f_1:$	$\langle e \leftrightarrow l_f\rangle$	$f_1:$	$\langle e \leftrightarrow L'_f\rangle$	$f_1:$	$\langle e \leftrightarrow L''_f\rangle$
$f_2:$	$\langle L_f \rightarrow E_f\rangle$	$f_2:$	$\langle l_f \rightarrow L_f\rangle$	$f_2:$	$\langle L'_f \rightarrow l'_f\rangle$	$f_2:$	$\langle L''_f \rightarrow l'_f\rangle$
$f_3:$	$\langle E_f \rightarrow F_f\rangle$	$f_3:$	$\langle L_f \leftrightarrow l'_f\rangle$	$f_3:$	$\langle l'_f \leftrightarrow a_r\rangle$	$f_3:$	$\langle l'_f \leftrightarrow \$_f\rangle$
$f_4:$	$\langle F_f \rightarrow \$_f\rangle$	$f_4:$	$\langle l'_f \rightarrow L'_f\rangle$	$f_4:$	$\langle l'_f \leftrightarrow \$_f\rangle$	$f_4:$	$\langle \$_f \rightarrow l_g\rangle$
$f_5:$	$\langle \$_f \leftrightarrow D\rangle$	$f_5:$	$\langle L'_f \leftrightarrow l''_f\rangle$	$f_5:$	$\langle \$_f \rightarrow \bar{l}_n\rangle$	$f_5:$	$\langle l_g \leftrightarrow e\rangle$
		$f_6:$	$\langle l''_f \rightarrow L'''_f\rangle$	$f_6:$	$\langle a_r \rightarrow e\rangle$	$f_6:$	$\langle l'_f \leftrightarrow \bar{l}_n\rangle$
		$f_7:$	$\langle L'''_f \rightarrow L''_f\rangle$	$f_7:$	$\langle \bar{l}_n \leftrightarrow e\rangle$	$f_7:$	$\langle \bar{l}_n \rightarrow l_n\rangle$
		$f_8:$	$\langle L''_f \leftrightarrow e\rangle$			$f_8:$	$\langle l_n \leftrightarrow e\rangle$
		$f_9:$	$\langle L_f \rightarrow t\rangle$				
		$f_{10}:$	$\langle L'_f \rightarrow t\rangle$				
		$f_{11}:$	$\langle t \rightarrow t\rangle$				

Figure 5.3: Programs for the instruction $l_f : (\texttt{CHECKSUB}(r), l_g, l_n)$

it always contains a sufficient amount of l'_i, l''_i objects for any $l_i \in H$. First we consider the case when there is at least one object a_r in the environment, that is, if the simulation starts in a configuration $(D, l''_1, e, e, e, e; l_f a_r[\ldots])$.

	configuration of Π					programs to be applied			
	C_1	C_4	C_5	C_6	Env	P_1	P_4	P_5	P_6
1.	D	e	e	e	$l_f a_r[\ldots]$	$-$	f_1	$-$	$-$
2.	D	l_f	e	e	$a_r[\ldots]$	$-$	f_2	$-$	$-$
3.	D	L_f	e	e	$a_r[\ldots]$	$-$	f_3	$-$	$-$
4.	D	l'_f	e	e	$L_f a_r[\ldots]$	f_1	f_4	$-$	$-$
5.	L_f	L'_f	e	e	$D a_r[\ldots]$	f_2	f_5	$-$	$-$
6.	E_f	l''_f	e	e	$L'_f D a_r[\ldots]$	f_3	f_6	f_1	$-$
7.	F_f	L'''_f	L'_f	e	$D a_r[\ldots]$	f_4	f_7	f_2	$-$
8.	$\$_f$	L''_f	l'_f	e	$D a_r[\ldots]$	f_5	f_8	f_3	$-$
9.	D	e	a_r	e	$\$_f L''_f[\ldots]$	$-$	$-$	f_6	f_1
10.	D	e	e	L''_f	$\$_f[\ldots]$	$-$	$-$	$-$	f_2
11.	D	e	e	l'_f	$\$_f[\ldots]$	$-$	$-$	$-$	f_3
12.	D	e	e	$\$_f$	$[\ldots]$	$-$	$-$	$-$	f_4
13.	D	e	e	l_g	$[\ldots]$	$-$	$-$	$-$	f_5
14.	D	e	e	e	$l_g[\ldots]$	$-$	g_1	$-$	$-$

In 13 steps, from $(D, l_1'', e, e, e, e; l_f a_r[\ldots])$ we obtain $(D, l_1'', e, e, e, e; l_g[\ldots])$ where l_g is the label of the instruction which should follow the successful decrease of the value of the nonempty register r, and the environment contains a multiset of objects l_i', l_i'' for $l_i \in H$.

Now we consider the case when register r, which is the register to be decremented, stores zero, that is, if the simulation starts in a configuration $(D, l_1'', e, e, e, e; l_f[\ldots])$ where the environment does not contain any object a_r.

	configuration of Π					programs to be applied			
	C_1	C_4	C_5	C_6	Env	P_1	P_4	P_5	P_6
1.	D	e	e	e	$l_f[\ldots]$	$-$	f_1	$-$	$-$
2.	D	l_f	e	e	$[\ldots]$	$-$	f_2	$-$	$-$
3.	D	L_f	e	e	$[\ldots]$	$-$	f_3	$-$	$-$
4.	D	l_f'	e	e	$L_f[\ldots]$	f_1	f_4	$-$	$-$
5.	L_f	L_f'	e	e	$D[\ldots]$	f_2	f_5	$-$	$-$
6.	E_f	l_f''	e	e	$L_f'D[\ldots]$	f_3	f_6	f_1	$-$
7.	F_f	L_f'''	L_f'	e	$D[\ldots]$	f_4	f_7	f_2	$-$
8.	$\$_f$	L_f''	l_f'	e	$D[\ldots]$	f_5	f_8	$-$	$-$
9.	D	e	l_f'	e	$\$_f L_f''[\ldots]$	$-$	$-$	f_4	f_1
10.	D	e	$\$_f$	L_f''	$[\ldots]$	$-$	$-$	f_5	f_2
11.	D	e	$\bar{l}_n$	l_f'	$[\ldots]$	$-$	$-$	f_7	$-$
12.	D	e	e	l_f'	$\bar{l}_n[\ldots]$	$-$	$-$	$-$	f_6
13.	D	e	e	$\bar{l}_n$	$[\ldots]$	$-$	$-$	$-$	f_7
14.	D	e	e	l_n	$[\ldots]$	$-$	$-$	$-$	f_8
15.	D	e	e	e	$l_n[\ldots]$	$-$	n_1	$-$	$-$

Similarly to the previous case, in 14 steps we obtain a configuration $(D, l_1'', e, e, e, e; l_n[\ldots])$ where l_n is the label of the instruction which should follow l_f if register r is empty, that is, if the decrease of its value is not possible.

Consider now what happens if there is an insufficient amount of objects l_i', l_i'' for $l_i \in H$ is present in the environment. Notice that such symbols are needed in step 3 and 5 by cell C_4. If there is no more available (not enough of them were produced in the initial phase by C_1 and C_2), then the programs f_9, f_{10}, and f_{11} do not allow the halting of the computation.

From these considerations we can see that after the initialization phase, all instructions of the register machine M can be simulated by the P colony. If the label of the halt instruction, l_h is produced, the computation halts since there is no program for processing the object l_h. The reader can immediately see that Π computes the same set of numbers as M. □

5.3.4 Remarks

In section 5.3.1, we have shown that P colonies are able to simulate universal register machines, provided they are initialized as follows: besides the environmental object, a finite number of objects are placed in the environment. Thus, P colonies are able to generate any recursively enumerable set of nonnegative integers with a bounded number of cells, each containing a bounded number of programs of a bounded length. These results represent a first attempt to bound the number of cells and number of programs simultaneously, they appeared in (Csuhaj-Varjú et al., 2006b). The proofs are based on techniques from (Csuhaj-Varjú et al., 2006a), and the fact that there is an appropriate universal register machine with 15 instructions (Observation 5.3.2).

In sections 5.3.2 and 5.3.3 we have shown that the already very simple model of P colony can be further simplified: First, insertion/deletion programs can be used in such a way that the cells can either only insert, or only delete objects from the environment, and second, instead of two or three, it is sufficient to have just one object inside each cell even if checking rules are not allowed to be used. These last results first appeared in (Ciencialová et al., 2009). The fact that one-symbol P colonies generate any recursively enumerable language was known already from (Cienciala et al., 2007).

Bibliography

Alberts, B., Bray, D., Lewis, J., Raff, M., Roberts, K., and Watson, J. (1994). *The Molecular Biology of the Cell.* Garland Science, New York.

Alhazov, A., Freund, R., and Rogozhin, Y. (2006). Computational power of symport/antiport: History, advances, and open problems. In Freund, R., Păun, G., Rozenberg, G., and Salomaa, A., editors, *Membrane Computing*, volume 3850 of *Lecture Notes in Computer Science*, pages 1–30. Springer Berlin Heidelberg.

Alhazov, A., Margenstern, M., Rogozhin, V., Rogozhin, Y., and Verlan, S. (2005a). Communicative P systems with minimal cooperation. In Gutiérrez-Naranjo, M. A., Păun, G., and Pérez-Jiménez, M. A., editors, *Cellular Computing. Complexity Aspects*, pages 161–177. Fenix Editora, Sevilla.

Alhazov, A., Rogozhin, Y., and Verlan, S. (2005b). Symport/antiport tissue P systems with minimal cooperation. In Gutiérrez-Naranjo, M. A., Păun, G., and Pérez-Jiménez, M. A., editors, *Cellular Computing. Complexity Aspects*, pages 37–52. Fenix Editora, Sevilla.

Bernardini, F. and Gheorghe, M. (2003). On the power of minimal symport/antiport. In Alhazov, A., n Vide, C. M., and aun, G. P., editors, *Workshop on Membrane Computing, WMC-2003, Tarragona, July 17-22, 2003*, number 28/3 in Research Group on Mathematical Linguistics, pages 72–83, Tarragona, Spain. Rovira i Virgili University.

Bernardini, F. and Păun, A. (2004). Universality of minimal symport/antiport: Five membranes suffice. In Martín-Vide, C., Mauri, G., Păun, G., Rozenberg, G., and Salomaa, A., editors, *Membrane Computing. International Workshop WMC 2003, Tarragona, Spain, July 17-22,* *2003. Revised Papers*, volume 2933 of *Lecture Notes in Computer Science*, pages 43–54. Springer-Verlag.

Cienciala, L., Ciencialová, L., and A., K. (2007). On the number of agents in p colonies. In Eleftherakis, G., Kefalas, P., Păun, G., Rozenberg, G., and Salomaa, A., editors, *Membrane Computing. 8th International Workshop, WMC 2007. Thessaloniki, Greece, June 25-28, 2007. Revised Selected and Invited Papers*, volume 4860, pages 193–208, Berlin-Heidelberg. Springer.

Ciencialová, L., Csuhaj-Varjú, E., Kelemenová, A., and Vaszil, G. (2009). Variants of P colonies with very simple cell structure. *International Journal of Computers Communication and Control*, 4(3):224–233.

Ciobanu, G., Pérez-Jiménez, M., and Păun, G., editors (2006). *Applications of Membrane Computing.* Springer Berlin Heidelberg.

Csuhaj-Varjú, E., Dassow, J., Kelemen, J., and Păun, G. (1994). *Grammar Systems: A Grammatical Approach to Distribution and Cooperation*, volume 5 of *Topics in Computer Mathematics.* Gordon and Breach Science Publishers.

Csuhaj-Varjú, E., Kelemen, J., Kelemenová, A., Păun, G., and Vaszil, G. (2006a). Computing with cells in environment: P colonies. *Journal of Multiple-Valued Logic and Soft Computing*, 12(3-4):201–215.
Impact factor: 0.2.

Csuhaj-Varjú, E., Kelemen, J., Kelemenová, A., and Păun, G. (1997). Eco-grammar systems - A grammatical framework for life-like interactions. *Artificial Life*, 3:1–28.

Csuhaj-Varjú, E., Margenstern, M., and Vaszil, G. (2006b). P colonies with a bounded number of cells and programs. In Hoogeboom, H. J., Păun, G., Rozenberg, G., and Salomaa, A., editors, *Membrane Computing, 7th International Workshop, WMC 2006, Leiden, The Netherlands, July 17-21, 2006, Revised, Selected, and Invited Papers*, volume 4361 of *Lecture Notes in Computer Science*, pages 352–366. Springer.

Csuhaj-Varjú, E. and Mitrana, V. (2000). Dynamical teams in eco-grammar systems. *Fundamenta Informatica*, 44(1–2):83–94.

Csuhaj-Varjú, E., Oswald, M., and Vaszil, G. (2011). PC grammar systems with clusters of components. *International Journal of Foundations of Computer Science*, 22(1):203–212.
Impact factor: 0.459 (2010).

Csuhaj-Varjú, E., Păun, G., and Vaszil, G. (2003). PC grammar systems with five context-free components generate all recursively enumerable languages. *Theoretical Computer Science*, 299(1-3):785–794.
Impact factor: 0.764.

Csuhaj-Varjú, E. and Vaszil, G. (1999). On the computational completeness of context-free parallel communicating grammar systems. *Theoretical Computer Science*, 215(1-2):349–358.
Impact factor: 0.413.

Csuhaj-Varjú, E. and Vaszil, G. (2001). On context-free parallel communicating grammar systems: synchronization, communication, and normal forms. *Theoretical Computer Science*, 255(1-2):511–538.
Impact factor: 0.468.

Csuhaj-Varjú, E. and Vaszil, G. (2002). Parallel communicating grammar systems with bounded resources. *Theoretical Computer Science*, 276(1-2):205–219.
Impact factor: 0.417.

Csuhaj-Varjú, E. and Vaszil, G. (2009). On the size complexity of non-returning context-free PC grammar systems. In Dassow, J., Pighizzini, G., and Truthe, B., editors, *Proceedings Eleventh International Workshop on Descriptional Complexity of Formal Systems*, volume 3 of *Electronic Proceedings in Theoretical Computer Science*, pages 91–101.

Csuhaj-Varjú, E. and Vaszil, G. (2010a). On the descriptional complexity of context-free non-returning PC grammar systems. *Journal of Automata, Languages and Combinatorics*, 15(1–2):91–105.

Csuhaj-Varjú, E. and Vaszil, G. (2010b). Scattered context grammars generate any recursively enumerable language with two nonterminals. *Information Processing Letters*, 110(20):902–907.
Impact factor: 0.612.

Csuhaj-Varjú, E. and Vaszil, G. (2011). On the number of components and clusters of nonreturning parallel communicating grammar systems. In Holzer, M., Kutrib, M., and Pighizzing, G., editors, *Descriptional Complexity of Formal Systems. 13th International Workshop, DCFS 2011. Gießen/Limburg, Germany, July 2011. Proceedings*, volume 6808 of *Lecture Notes in Computer Science*, pages 121–134, Berlin Heidelberg. Springer-Verlag.

Čulik II, K. and Maurer, H. (1977). Tree controlled grammars. *Computing*, 19:129–139.

Dassow, J. and Păun, G. (1989). *Regulated Rewriting in Formal Language Theory.* Springer, Berlin.

Dassow, J., Păun, G., and Salomaa, A. (1997a). Grammars with controlled derivations. In (Rozenberg and Salomaa, 1997), pages 101–154.

Dassow, J., Păun, G., and Rozenberg, G. (1997b). Grammar systems. In Salomaa, A. and Rozenberg, G., editors, *Handbook of Formal Languages*, volume II, chapter 4, pages 155–213. Springer-Verlag, Berlin-Heidelberg.

Dassow, J., Stiebe, R., and Truthe, B. (2010). Generative capacity of subregularly tree controlled grammars. *International Journal of Foundations of Computer Science*, 21.

Dumitrescu, S. (1996). Non-returning PC grammar systems can be simulated by returning systems. *Theoretical Computer Science*, 165:463–474.

Fernau, H., Freund, R., Oswald, M., and Reinhardt, K. (2007). Refining the nonterminal complexity of graph controlled, programmed, and matrix grammars. *Journal of Automata, Languages and Combinatorics*, 12:117–138.

Fernau, H. and Meduna, A. (2003a). On the degree of scattered context-sensitivity. *Theoretical Computer Science*, 290:2121–2124.

Fernau, H. and Meduna, A. (2003b). A simultaneous reduction of several measures of descriptional complexity in scattered context grammars. *Information Processing Letters*, 86:235–240.

Fischer, P. C. (1966). Turing machines with restricted memory access. *Information and Control*, 9:364–379.

Frisco, P. (2004). About p systems with symport/antiport. In *Second Brainstorming Week in Membrane Computing. Sevilla, February 2-7, 2004*, number 01/2004 in Technical Reports of the Research Group in Natural Computing, pages 224–236, Sevilla. University of Sevilla.

Frisco, P. (2009). *Computing with Cells. Advances in Membrane Computing.* Oxford University Press, New York.

Frisco, P. and Hoogeboom, H. (2004). P systems with symport/antiport simulating counter automata. *Acta Informatica*, 41(2-3):145–170.

Frĭs, I. (1968). Grammars with partial ordering of rules. *Information and Control*, 12:415–425.

Geffert, V. (1988). Context-free-like forms for the phrase structure grammars. In Chytil, M., Janiga, L., and Koubek, V., editors, *Mathematical Foundations of Computer Science 1988, 13th International Symposium, Carlsbad, Czechoslovakia, August 29 - September 2, 1988*, volume 324 of *Lecture Notes in Computer Science*, pages 309–317, Berlin. Springer.

Goldstine, J., Kappes, M., Kintala, C. M. R., Leung, H., Malcher, A., and Wotschke, D. (2002). Descriptional complexity of machines with limited resources,. *Journal of Universal Computer Science*, 8(2):193–234.

Gonczarowski, J. and Warmuth, M. K. (1989). Scattered and context-sensitive rewriting. *Acta Informatica*, 20:391–411.

Greibach, S. A. and Hopcroft, J. E. (1969). Scattered context grammars. *Journal of Computer and System Sciences*, 3:232–247.

Gruska, J. (1969). Some classifications of context-free languages,. *Information and Control*, 14:152–179.

Holzer, M. and Kutrib, M. (2011). Descriptional complexity - an introductory survey. In Martín-Vide, C., editor, *Scientific Applications of Language Methods*, volume 2 of *Mathematics, Computing, Language, and Life: Frontiers in Mathematical Linguistics and Language Theory*, pages 1–58. Imperial College Press, London.

Kari, L., Martín-Vide, C., and Păun, G. (2004). On the universality of p systems with minimal symport/antiport rules. In Jonoska, N., Păun, G., and Rozenberg, G., editors, *Aspects of Molecular Computing, Essays Dedicated to Tom Head on the Occasion of His 70th Birthday*, volume 2950 of *Lecture Notes in Computer Science*, pages 254–265. Springer-Verlag.

Kari, L., Mateescu, A., Păun, G., and Salomaa, A. (1995). Teams in cooperating grammar systems. *Journal of Experimental and Theoretical Artificial Intelligence*, 7:347–359.

Kelemen, J. (1984). Conditional grammars: Motivations, definitions, and some properties. In *Proc. Conf. Automata, Languages and Math. Sciences, Salgótarján*, pages 110–123.

Kelemen, J. and Kelemenová, A. (1992). A grammar-theoretic treatment of multiagent systems. *Cybernetics and Systems*, 23:621–633.

Kelemen, J., Kelemenová, A., and Păun, G. (2004). Preview of p colonies: A biochemically inspired computing model. In Bedau, M., Husbands, P., Hutton, T., Kumar, S., and Suzuki, H., editors, *Workshop and Tutorial Proceedings. Ninth International Conference on the Simulation and Synthesis of Living Systems (Alife IX)*, pages 82–86, Boston Mass.

Kelemenová, A. (1982). Grammatical complexity of context-free languages and normal forms of context-free grammars. In Mikulecký, P., editor, *Second Czechoslovak-Soviet Conference of Young Computer Scientists, Smolenice Castle, November 8-12, 1982*, pages 239–258, Bratislava.

Korec, I. (1996). Small universal register machines. *Theoretical Computer Science*, 168(2):267–301.

Lázár, K. A., Csuhaj-Varjú, E., Lőrincz, A., and Vaszil, G. (2009). Dynamically formed clusters of agents in eco-grammar systems. *International Journal of Foundations of Computer Science*, 20(2):293–311.
Impact factor: 0.512.

Mandache, N. (2000). On the computational power of context-free PC grammar systems. *Theoretical Computer Science*, 237(1-2):135–148.

Martín-Vide, C., Păun, A., and Păun, G. (2002a). On the power of p systems with symport rules. *Journal of Universal Computer Science*, 8:317–331.

Martín-Vide, C., Păun, A., Păun, G., and Rozenberg, G. (2002b). Membrane systems with coupled transport. *Fundamenta Informaticae*, 49:317–331.

Masopust, T. (2009a). A note on the generative power of some simple variants of context-free grammars regulated by context conditions. In *LATA 2009*, volume 5457 of *Lecture Notes in Computer Science*, pages 554–565, Berlin Heidelberg.

Masopust, T. (2009b). On the descriptional complexity of scattered context grammars. *Theoretical Computer Science*, 410(1):108–112.

Masopust, T. (2010). Bounded number of parallel productions in scattered context grammars with three nonterminals. *Fundamenta Informaticae*, 99(4):473–480.

Masopust, T. and Meduna, A. (2009). Descriptional complexity of three nonterminal scattered context grammars: An improvement. In Dassow, J., Pighizzini, G., and Truthe, B., editors, *11th International Workshop on Descriptional Complexity of Formal Systems (DCFS 2009*, volume 3 of *Electronic Proceedings in Theoretical Computer Science*, pages 183–192.

Mateescu, A., Mitrana, V., and Salomaa, A. (1993/94). Dynamical teams of cooperating grammar systems. *Analele Universitatii Bucuresti. Matematica Inform.*, 42(43):3–14.

Meduna, A. (1995a). Syntactic complexity of scattered context grammars. *Acta Informatica*, 32:285–298.

Meduna, A. (1995b). A trivial method of characterizing the family of recursively enumerable languages by scattered context grammars. *Bulletin of the EATCS*, 56:104–106.

Meduna, A. (1997). Four nonterminal scattered context grammars characterize the family of recursively enumerable languages. *International Journal of Computer Mathematics*, 63:67–83.

Meduna, A. (2000a). Generative power of three-nonterminal scattered context grammars. *Theoretical Computer Science*, 246:423–427.

Meduna, A. (2000b). Terminating left-hand sides of scattered context productions. *Theoretical Computer Science*, 237:423–427.

Meduna, A. and Gopalaratnam, A. (1994). On semi-conditional grammars with productions having either forbidding or permitting conditions. *Acta Cybernetica*, 11:307–323.

Meduna, A. and Švec, M. (2002). Reduction of simple semi-conditional grammars with respect to the number of conditional productions. *Acta Cybernetica*, 15:353–360.

Mihalache, V. (1994). On parallel communicating grammar systems with context-free components. In aun, G. P., editor, *Mathematical Linguistics and Related Topics*, pages 258–270. The Publishing House of the Romanian Academy of Sciences, Bucharest.

Minsky, M. (1967). *Computation – Finite and Infinite Machines.* Prentice Hall, Englewood Cliffs, NJ.

Okubo, F. (2009). A note on the descriptional complexity of semi-conditional grammars. *Information Processing Letters*, 110(1):36–40.

Păun, G. (1979). On the generative capacity of tree controlled grammars. *Computing*, 21:213–220.

Păun, G. (1985). A variant of random context grammars: Semi-conditional grammars. *Theoretical Computer Science*, 41:1–17.

Păun, G. (2000). Computing with membranes. *Journal of Computer and System Sciences*, 61(1):108–143.

Păun, G. and Rozenberg, G. (1994). Prescribed teams of grammars. *Acta Informatica*, 31(6):525–537.

Păun, G., Rozenberg, G., and Salomaa, A., editors (2010). *The Oxford Handbook of Membrane Computing.* Oxford University Press.

Păun, G. and Sântean, L. (1989). Parallel communicating grammar systems: The regular case. *Annals of the University of Bucharest, Mathematics-Informatics Series*, 38(2):55–63.

Păun, A. and Păun, G. (2002). The power of communication: P systems with symport/antiport. *New Generation Computing*, 20(3):295–306.

Păun, G. (2002). *Membrane Computing: An Introduction.* Natural Computing Series. Springer, Berlin.

Rozenberg, G. and Salomaa, A., editors (1997). *Handbook of Formal Languages.* Springer, Berlin.

Salomaa, A. (1973). *Formal Languages.* Academic Press, New York.

ter Beek, M. H. (1996). Teams in grammar systems: hybridity and weak rewriting. *Acta Cybernetica*, 12(4):427–444.

ter Beek, M. H. (1997). Teams in grammar systems: sub-context-free cases. In Păun, G. and Salomaa, A., editors, *New trends in formal languages. Control, cooperation and combinatorics*, volume 1218 of *Lecture Notes in Computer Science*, pages 197–216. Springer, Berlin.

Turaev, S., Dassow, J., Manea, F., and Selamat, M. (2012). Language classes generated by tree controlled grammars with bounded nonterminal complexity. *Theoretical Computer Science*, 449:134–144.

Turaev, S., Dassow, J., and Selamat, M. (2011a). Language classes generated by tree controlled grammars with bounded nonterminal complexity. In Holzer, M., Kutrib, M., and Pighizzini, G., editors, *Descriptional Complexity of Formal Systems, 13th International Workshop, DCFS 2011, Gießen-Limburg, Germany, July 2011, Proceedings.*, volume 6808 of *Lecture Notes in Computer Science*, pages 289–300, Berlin Heidelberg. Springer.

Turaev, S., Dassow, J., and Selamat, M. (2011b). Nonterminal complexity of tree controlled grammars. *Theoretical Computer Science*, 412(41):5789–5795.

Vaszil, G. (1998). On simulating non-returning PC grammar systems with returning systems. *Theoretical Computer Science*, 209(1-2):319–329. Impact factor: 0.349.

Vaszil, G. (2005a). On the descriptional complexity of some rewriting mechanisms regulated by context conditions. *Theoretical Computer Science*, 330(2):361–373.
Impact factor: 0.743.

Vaszil, G. (2005b). On the size of P systems with minimal symport/antiport. In Mauri, G., Păun, G., Pérez-Jiménez, M. J., Rozenberg, G., and Salomaa, A., editors, *Membrane Computing, 5th International Workshop, WMC 2004, Milan, Italy, June 14-16, 2004, Revised Selected and Invited Papers*, volume 3365 of *Lecture Notes in Computer Science*, pages 404–413. Springer.
Impact factor: 0.402.

Vaszil, G. (2007). Non-returning PC grammar systems generate any recursively enumerable language with eight context-free components. *Journal of Automata, Languages and Combinatorics*, 12(1-2):307–315.

Vaszil, G. (2012). On the nonterminal complexity of tree controlled grammars. In Bordihn, H., Kutrib, M., and Truthe, B., editors, *Languages Alive*, volume 7300 of *Lecture Notes in Computer Science*, pages 265–272. Springer, Berlin Heidelberg.

Virkkunen, V. (1973). On scattered context grammars. *Acta Universitatis Ouluensis Scientiae Rerum Naturalium Series A, Mathematica*, 6:75–82.