Computing by Observing: Simple Systems and Simple Observers

Matteo Cavaliere [a,1] Peter Leupold [b,1]

[a] *CNB-CSIC*
Madrid, Spain
mcavaliere@cnb.csic.es

[b] *Fachbereich Elektrotechnik/Informatik*
Fachgebiet Theoretische Informatik
Universität Kassel, Kassel, Germany
Peter.Leupold@web.de

Abstract

We survey and extend the work on the paradigm called "computing by observing". Its central feature is that one considers the behavior of an evolving system as the result of a computation. To this purpose an external observer records this behavior. In this way, several computational trade-offs between the observer and the observed system can be determined. It has turned out that *the observed behavior of computationally simple systems can be very complex*, when an appropriate observer is used. For example, a restricted version of context-free grammars with regular observers suffices to obtain computational completeness. As a second instantiation presented here, we apply an observer to sticker systems, an abstract model of DNA computing. Finally, we introduce and investigate the case where the observers can read only one measure of the observed system (e.g., mass or temperature) modeling in this way, limitations in the observation of real physical systems. Finally a research perspective on the topic is presented.

Key words: complexity, observer, automata theory, behavior, natural computing

[1] Part of this work was done while Matteo Cavaliere was a post-doctoral researcher at CoSBi, Trento, Italy, and Peter Leupold was funded as a post-doctoral fellow by the Japanese Society for the Promotion of Science under number P07810.

1 Introduction

The paradigm of *computing by observing* was originally introduced under the name "evolution and observation", [7], based on the following reflections. Nearly all models in the area of DNA and natural computing follow the classical computer science (input/output) paradigm of processing an input directly to an output, which is then the result of the computation. Only the mechanisms of processing are different from conventional models; instead of a finite state control or a programming language it is biomolecular mechanisms that are used, or rather abstractions of such mechanisms.

However, in many experiments in biology and chemistry the setup is fundamentally different. The matter of interest is not some product of the system but rather the change observed in certain, selected quantities. To cite two simple examples: the predator-prey relationship, where the interesting fact is how the change in one population effects a change in the other; a reaction with catalyst often has the same product as without, but the energy curves during the reactions are different.

The goal was to formalize this approach in an architecture for computation. The resulting paradigm is that of "computing by observing". It consists of an underlying observed system, which evolves in discrete steps from one configuration to the next. An external observer reads these configurations and transforms into single letters; a type of classification. In this way a sequence of configurations is transformed into a simple sequence of symbols, i.e. a string. This corresponds to the protocol of an experiment in biology or chemistry and for us is the result of the computation.

The architecture that protocols the observed evolution of a system in the way described is schematically depicted in Figure 1.

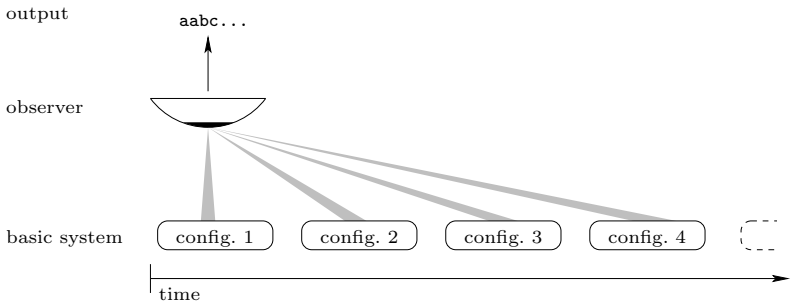


Fig. 1. Basic structure of the "computing by observing" architecture. To each configuration the observer associates a symbol. The sequence of symbols (string) is the observed behavior of the basic system, and is interpreted as the result (output) of the computation. Computing by observing investigates the interplay and the trade-offs between the observer and the observed system.

In this combination the components are frequently more powerful than just by themselves and, often, the complexity of the observed behaviors is very different from the complexity of the components. For example, context-free string-rewriting and a regular observer suffice to obtain computational completeness [7,8]. Similar results were obtained for membrane systems, sticker systems [1], and splicing systems. Recently several articles with the character of overviews of the field have appeared [9,16,5].

The goal of the paradigm is to stress the role of the observer in natural computation. When the observer can be freely programmed, its role can actually be crucial as we will see in Section 2: in fact, any computational device can be obtained by observing in an appropriate manner a fixed (and computationally simple) basic system (one can then talk of "computing by only observing..."). However the situation seems different when the observer is restricted as suggested by limitations of real physical systems. For these reasons, in Section 4, we introduce and investigate the paradigm with an observer that can read only one specific measure of the observed system (e.g., temperature, volume, mass etc). Equivalence of the paradigm with known generative devices is also shown.

We shortly recall here the basic notions of formal languages theory used in the paper. For details about these we refer to the books of Harrison [13] or Salomaa [25].

By Σ we usually denote a finite alphabet, and Σ^* is the set of all strings over this alphabet, including the empty string λ. For a string w we denote by $|w|$ its length. $|w|_u$ denotes the number of distinct occurrences of u in w. The letter at the i-th position of a word w is denoted by $w[i]$. For factors we use the notation $w[i \dots j]$. A *prefix* of a word w is any factor starting in the first position, i.e. $w[1 \dots i]$ for some i. The *reverse* of a word w is $w^R := w[|w|]w[|w|-1] \cdots w[1]$. This notion is extended to languages in the canonical way such that $L^R := \{w^R : w \in R\}$. As standard in the area, we denote by FIN, REG, CF, CS, and RE we respectively denote the classes of the finite, regular, context-free, context-sensitive and recursively enumerable languages.

2 Observed Complexity of Simple Grammars

In this section we present *grammar/observer (G/O) systems* that are generative devices based on the discussed "computing by observing" paradigm. In this case, a formal grammar plays the role of the observed basic system and the external observer is a finite state automaton. This section surveys the work presented in [7] and [6].

2.1 The Observers: Monadic Transducers

A grammar's configurations are the sentential forms of its derivations. So for the observer we need a device mapping these arbitrarily long sentential forms (strings) into just one singular symbol. We use a special variant of finite automata, called a monadic transducer, with some feature known from Moore machines, [13], or also from subsequential transducers: the set of states is labeled with the symbols of an output alphabet. Any computation of the automaton produces as output the label of the state it halts in. For an input w, and an observer A, we denote $A(w)$ the output of A on input w. Because we find it preferable that the observation of a certain string always leads to a fixed result, we consider here only deterministic and complete automata. [2]

For simplicity, in what follows, we present only the mappings that the observers define, without giving a real implementation (in terms of finite automata) for them. Therefore no more formal definition of monadic transducers is necessary here. The class of all monadic transducers is denoted by FA_O.

2.2 Grammar/Observer Systems

A *Grammar/Observer (G/O) system* is a pair $\Omega = (G, A)$ constituted by a generative grammar $G = (N, T, S, P)$ and a monadic transducer (observer) A with output alphabet $\Sigma \cup \{\perp\}$, which is also the output alphabet of the entire system Ω. The observer's input alphabet must be the union of N and T from the grammar so that it can read all sentential forms.

Several modes of generation can be defined (see, e.g., [7]). Here we will consider the mode of generation that admits writing an empty and non-empty output in an arbitrary manner (*free G/O systems*); i.e., the observer can either output one letter or the empty word and freely alternate between these two options. A *free G/O system* generates a language in the following manner:

$$L_f(\Omega) = \{A(w_0, w_1, \ldots, w_n) \mid S = w_0 \Rightarrow w_1 \Rightarrow \cdots \Rightarrow w_n,\ w_n \in T^*\}.$$

Here $A(w_0, w_1, \ldots, w_n)$ is used as a more compact way of writing the catenation $A(w_0)A(w_1)\cdots A(w_n)$.

In other words, the language contains all those words obtained by catenation of the symbols output by the observer during the terminating derivations of

[2] The name monadic transducer is motivated from monadic string-rewriting rules; these can have arbitrarily long left sides that are rewritten to strings of length one or zero.

the observed grammar. Derivations which do not terminate do not produce a valid output; this means that we only take into account finite words. Of course, by considering the other case of non-terminating derivations the G/O systems could also be used to generate languages of infinite words.

We also consider the variant where we define the language produced by Ω as

$$L_{\perp,f}(\Omega) = L_f(\Omega) \cap \Sigma^*.$$

In this way the strings in $L_f(\Omega)$ containing $\perp$ are filtered out and they are not present in $L_{\perp,f}(\Omega)$. Thus the observer has in some sense the ability to reject a computation, when configurations of a certain class appear. The language generated by Ω is also called observed behavior of Ω.

The functioning of a (free) G/O system is sketched in Example 2.1.

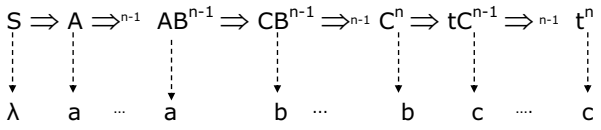
Example 2.1

G = (N, T, S, P)

N = {S, A, B, C} T = {t}

P= { S → A,
A → AB, A → C, B → C, C → t }

$$A(w) = \begin{cases} \lambda & \text{if } w = S \\ a & \text{if } w \in AB^* \\ b & \text{if } w \in C^+B^* \\ c & \text{if } w \in t^+C^* \\ \perp & \text{else} \end{cases}$$



observed behavior of the grammar {a^n b^n c^n | n >0}

To each sentential form produced by the grammar G the observer A associates a symbol that can be $a, b, c, \perp$ or the empty string λ (the vertical arrow is the observer mapping). Thus every computation of the observer produces one output symbol and the concatenation of these symbols is then the output string. For instance, in the Example 2.1, the output string is $\lambda a \cdots ab \cdots bc \cdots c$. The mapping defined by the observer is specified by the regular expressions. The language $L_f(\Omega)$ is obtained by considering all possible halting derivations of G and collecting all the output strings. It is easy to see that in this case $L_{\perp,f}(\Omega)$ is $\{a^n b^n c^n \mid n > 0\}$.

In [7] it has been shown that a G/O system composed of very simple components, namely a locally commutative context-free grammar ($LCCF$), a proper subclass of context-free grammars, and a finite state automaton is computationally complete.

Theorem 2.1 *[7] For each $L \in RE$ there exists a G/O system $\Omega = (G, A)$, with G an LCCF, such that $L_{\perp,f}(\Omega) = L$.*

An interesting fact is that the observer's ability to produce $\perp$, i.e., to eliminate certain computations, seems to be a powerful and essential feature in all the variants explored in [7]. Besides the free variant presented here, systems that have to write in every single step and systems that have to write in every step after they have started to write are investigated there.

As can be seen from the definition, for free systems we obtain all recursively enumerable languages over Σ simply by intersection of a language over $\Sigma \cup \{\perp\}$ with the regular language Σ^*. Now notice that recursive languages are closed under intersection with regular sets. Therefore, there must exist some grammar/observer systems Ω generating a *non-recursive* $L_f(\Omega)$ (i.e., not using the filtering with $\perp$). However, this intersection filters out a great many words produced from undesired computations; so despite being simple, this intersection distinguishes between good and bad computations. Therefore it seems very unlikely that the same model without this feature will be computationally complete, although, as previously mentioned, it generates non-recursive languages.

As discussed in the introduction, the goal of the presented framework is to stress the role of the observer in computations. Therefore, it is interesting to understand how much one can compute by making changes only in the observer, keeping the observed basic system unchanged. We show by means of an example that a G/O system can generate very different languages if the observer is changed while the grammar remains fixed.

Let us consider the context-free grammar $G = (\{S, A, B, C\}, \{t, p\}, S, \{S \to pS, S \to p, S \to A, A \to AB, A \to C, B \to C, C \to t\})$. If G is coupled with the observer A' such that $A'(w) = a$ if $w \in \{S, A, B, C, t, p\}^+$, then $\Omega = (G, A')$ defines the language $L_{\perp,f}(\Omega) = \{a^i \mid i \geq 2\}$, a regular language. In fact, the derivation $S \to pS \overset{n-2}{\Rightarrow} p^{n-1}S \to p^n$ produces (when observed) the string a^{n+1}.

Keeping the same grammar G we change the observer into A, the observer used in Example 2.1. In this case, one can verifies that $\Omega = (G, A)$ generates the language $L_{\perp,f}(\Omega) = \{a^n b^n c^n \mid n > 0\}$, a context-sensitive language.

This example suffices to underline that part of the computation can be done by choosing the appropriate observer, keeping unchanged the underlying basic system. Actually, the choice of observer can be really crucial: As shown in [6] one can construct a *universal context-free grammar* that can generate all recursively enumerable languages when observed in the appropriate manner.

We can also consider several *restrictions on the observed system*. In particular, we can bound the number of nonterminals in the sentential forms.

In this respect, we notice that the universal context-free grammar used in [6] has no bound on the number of nonterminals in its sentential forms.

The next result shows that indeed this is a necessary property of context-free grammars that are observationally complete for type-0 grammars. In fact, when a bound is imposed, the observed behaviors are regular. Recall that a context-free grammar is *nonterminal bounded* if there exists a constant k such that all sentential forms generated by the grammar have at most k nonterminals. Clearly, a regular grammar is nonterminal bounded.

Theorem 2.2 *[6] For every G/O system $\Omega = (G, A)$, with G nonterminal bounded context-free, $L_{\perp,f}(\Omega)$ is regular.*

3 Observed Complexity of Simple BioSystems

As we have seen in Section 2.2 the complexity of the produced output is determined by the particular dynamics of the observed system. This means that the entire trajectory followed by the observed system is important rather than the momentary reached states. In this section we stress this distinction by considering a sticker system (a computational model inspired by the self-assembly of DNA strands) as the observed system. In the area, it is known ([21]) that *sticker systems have a computational power that is less than that of regular grammars*. However their behavioral complexity is computationally complete when observed in an appropriate manner. e.g., the dynamics of a single strand is followed. This means that observed behaviors can represent all possible computable languages (if the observer can be freely programmed). In other words, a basic system (sticker system) that is less powerful than a regular grammar, when considered as an input/output device, is universal in terms of observed behaviors (but, as we will see, this assumes the ability to follow the dynamics of a single marked strand). The importance of the particular dynamics of a system is clear when we contrast this result with Theorem 2.2: observed behaviors of regular grammars are still regular.

Sticker systems were introduced in [20] as a formal model of the operation of *annealing* (and the operation of *ligation*) operation that is largely used in the DNA computing area. The basic operation of a sticker system is the *sticking* operation that constructs double stranded sequences out of "DNA dominoes" (*polyominoes*) that are sequences with one or two sticky ends, or single stranded sequences, attaching to each other by *ligation* and *annealing*.

An *observable sticker system* was introduced in [1]. The idea of an observable sticker system can be expressed in the following way: an observer (for example, a microscope) is placed outside the "test tube", where (an unbounded number of copies of) DNA strands and DNA dominoes are placed together. Some of these molecules are marked (for example, with a fluorescent particle). The molecules in the solution will start to self-assemble (to stick to each other) and, in this way, new molecules are obtained. The observer watches the trajectory of the marked molecules and stores such evolution on an external tape in a chronological order. For each possible trajectory of the marked molecules a certain string is obtained. Collecting all the possible trajectories of such marked strands we obtain a language.

We first recall the definition of sticker systems and we couple them with an external observer, defining an observable sticker system. In this section we survey the work presented in [1].

Consider a symmetric relation $\rho \subseteq V \times V$ over V (of *complementarity*). Following [21], we associate with V the monoid $V^* \times V^*$ of pairs of strings. Because it is intended to represent DNA molecules, we also write elements $(x_1, x_2) \in V^* \times V^*$ in the form $\begin{pmatrix} x_1 \\ x_2 \end{pmatrix}$ and $V^* \times V^*$ as $\begin{pmatrix} V^* \\ V^* \end{pmatrix}$. We denote by $\begin{bmatrix} V \\ V \end{bmatrix}_\rho = \left\{ \begin{bmatrix} a \\ b \end{bmatrix} \mid a, b \in V, (a, b) \in \rho \right\}$ the set of *complete double symbols*, and $WK_\rho(V) = \begin{bmatrix} V \\ V \end{bmatrix}_\rho^*$ is the set of the *complete double-stranded sequences (complete molecules)* also written as $\begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$, where x_1 is the *upper strand* and x_2 is the *lower strand*.

As in [21], we use *single strands* – the elements of $S(V) = \begin{pmatrix} \lambda \\ V^* \end{pmatrix} \cup \begin{pmatrix} V^* \\ \lambda \end{pmatrix}$ and the molecules with (a possible) *overhang* on the right, which are the elements of $R_\rho(V) = \begin{bmatrix} V \\ V \end{bmatrix}_\rho^* S(V)$, from now on called *well-started* molecules (upper and lower strand are defined as in the case of complete molecules).

Given a well started molecule $u \in R_\rho(V)$ and a single strand $v \in S(V)$, we recall in Figure 2 the partial operation $\mu : R_\rho(V) \times S(V) \longrightarrow R_\rho(V)$ of sticking, as defined in [21].

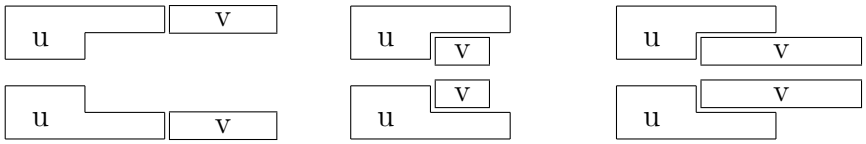


Fig. 2. Possible ways of sticking a single strand v to a well-started molecule u.

We point out that we use a case of sticking, restricted to pasting a single strand to the right side of a well-started molecule (with a possible overhang on the right), corresponding to the *simple regular* sticker systems.

A (simple regular) sticker system is a construct $\gamma = (V, \rho, X, D)$, where $X \subseteq R_\rho(V)$ is the (finite) set of axioms, and $D \subseteq S(V)$ is the (finite) set of *dominoes* (in this case these are single strands). Given $u, w \in R_\rho(V)$, we write $u \Rightarrow w$ iff $w = \mu(u, v)$ for some $v \in D$. A sequence $(w_i)_{1 \leq i \leq k}$, $w_i \in R_\rho(V)$, is called a *complete computation* if $w_1 \in X$, $w_i \Rightarrow w_{i+1}$ for $1 \leq i < k$ and $w_k \in WK_\rho(V)$.

The *language* generated by a sticker system γ is the set of upper strands of all complete molecules derived from the axioms. It is known that the *family of languages generated by simple regular sticker systems is strictly included in the family of regular languages* (see [21] for the proof).

For an alphabet V, our *double-symbol alphabet* constructed over V is

$$V_d = \left[\begin{array}{c} V \\ V \end{array} \right]_\rho \cup \left(\begin{array}{c} V \\ \lambda \end{array} \right) \cup \left(\begin{array}{c} \lambda \\ V \end{array} \right).$$

We define an observer $A \in FA_O$, with input alphabet V_d, that reads an entire molecule (element of $R_\rho(V)$) and outputs one symbol from the output alphabet $\Sigma \cup \{\perp\}$ (every well-started molecule in $R_\rho(V) \subseteq V_d^*$ is read, in a classical way, from left to right, scanning one double symbol from V_d at a time). For a molecule $w \in R_\rho(V)$ and the observer A we write $A(w)$ to indicate such output; for a sequence $w_1, \ldots, w_n$ of $n \geq 1$ of molecules in $R_\rho(V)$ we write $A(w_1, \ldots, w_n)$ for the string $A(w_1) \cdots A(w_n)$.

An *observable sticker system* with output alphabet $\Sigma \cup \{\perp\}$ is a construct $\phi = (\gamma, A)$, where γ is the sticker system with alphabet V, and $A \in FA_O$ is the observer with input alphabet V_d and output alphabet $\Sigma \cup \{\perp\}$.

We denote the collection of all complete computations of ϕ by $\mathcal{C}(\phi)$. The language, over the output alphabet $\Sigma \cup \{\perp\}$, generated by an observable sticker system ϕ, is defined as $L(\phi) = \{A(s) \mid s \in \mathcal{C}(\phi)\}$. As we have done for G/O systems, we want to filter out the words that contain the special symbol $\perp$, and so we consider the language $L_\perp(\phi) = L(\phi) \cap \Sigma^*$.

We will illustrate with a simple example how an observable sticker system

works. At the same time this example shows how one can construct an observable sticker system with an observed behavior that is a non regular language despite the fact that the power of simple regular sticker systems, when considered in the classical way, is subregular.

Consider the following observable sticker system $\phi = (\gamma, A)$:

$$\gamma = (V = \{a, \mathbf{c}, \mathbf{g}, t\}, \rho = \{(a,t), (\mathbf{c}, \mathbf{g}), (t,a), (\mathbf{g}, \mathbf{c})\}, X = \left\{ \begin{bmatrix} a \\ t \end{bmatrix} \right\}, D),$$

$$D = \left\{ \begin{pmatrix} a \\ \lambda \end{pmatrix}, \begin{pmatrix} \lambda \\ t \end{pmatrix}, \begin{pmatrix} \mathbf{c} \\ \lambda \end{pmatrix}, \begin{pmatrix} \lambda \\ \mathbf{g} \end{pmatrix} \right\},$$

with the observer A defined by the following mapping:

$$A(w) = \begin{cases} b, & \text{if } w \in \begin{bmatrix} a \\ t \end{bmatrix}^* \begin{pmatrix} a^* \\ \lambda \end{pmatrix} \cup \begin{pmatrix} \lambda \\ t^* \end{pmatrix}, \\ d, & \text{if } w \in \begin{bmatrix} a \\ t \end{bmatrix}^* \begin{pmatrix} a^*\mathbf{c} \\ \lambda \end{pmatrix} \cup \begin{pmatrix} \lambda \\ t^*\mathbf{g} \end{pmatrix}, \\ \lambda, & \text{otherwise.} \end{cases}$$

The language generated by γ is $L_1 = \{b^m d^n \mid m \geq n, m \geq 1, n \geq 0\} \notin REG$.

Below is an example for a computation of ϕ (generating *bbbbdd*):

Step	0	1	2	3	4	5	6
Added		$\begin{pmatrix} a \\ \lambda \end{pmatrix}$	$\begin{pmatrix} a \\ \lambda \end{pmatrix}$	$\begin{pmatrix} \lambda \\ t \end{pmatrix}$	$\begin{pmatrix} \mathbf{c} \\ \lambda \end{pmatrix}$	$\begin{pmatrix} \lambda \\ t \end{pmatrix}$	$\begin{pmatrix} \lambda \\ \mathbf{g} \end{pmatrix}$
Molecule	a t	aa t	aaa t	aaa tt	$aaa\mathbf{c}$ tt	$aaa\mathbf{c}$ ttt	$aaa\mathbf{c}$ $ttt\mathbf{g}$
Output	b	b	b	b	d	d	λ

The idea of the system ϕ is the following: think of symbols $\mathbf{c}$, $\mathbf{g}$ as "markers". While we stick to the current molecule either $\begin{pmatrix} a \\ \lambda \end{pmatrix}$ or $\begin{pmatrix} \lambda \\ t \end{pmatrix}$, the observer maps the result (a molecule without markers) to b. As soon as a marker becomes part of the current molecule, the new obtained molecules is mapped by

the observer maps to the symbol d. This continues until either the strand with a marker is extended or we have obtained a complete molecule.

Suppose that, when the first marker is attached, the length of the strand with that marker is l_1, and the length of the other strand is l_2 (clearly, $l_1 > l_2$). Then the output produced so far is $b^{l_1+l_2-2}d$. To complete the molecule by extending the strand without the marker we need to attach $l_1 - l_2$ symbols to it, and in this case the observer outputs $d^{l_1-l_2-1}\lambda$. Thus, the resulting string x consists of $l_1 + l_2 - 2$ b's and $l_1 - l_2$ d's. Since $l_2 \geq 1$, the difference between the number of b's and the number of d's is $l_1 + l_2 - 2 - (l_1 - l_2) = 2l_2 - 2 \geq 0$. (recall that in the case where we attach a symbol to a string with the marker, the observer only outputs λ, so the inequality $m = |x|_b \geq |x|_d = n$ remains valid, and all the combinations (m, n), $m \geq n$ are possible). Hence, $L(\gamma) = L_1$.

To get computational completeness we do not need "complicated" sticker systems but simple regular sticker systems and an observer that is able to discard any "bad" evolution.

Theorem 3.1 *[1] For each $L \in RE$ there exists an observable sticker system ϕ such that $L_\perp(\phi) = L$.*

Notice that, as already remarked earlier, using Theorem 3.1, the definition of $L_\perp(\phi)$ and the fact that recursive languages are closed under intersection with regular languages, we obtain:

Corollary 3.1 *[1] There exists an observable sticker system ϕ such that $L(\phi)$ is a non-recursive language.*

As briefly discussed in the introduction to this section, an interesting question here is why we obtain computational completeness with a regular system while with regular grammars the architecture is much weaker (Theorem 2.2). This is because the dynamics of sticker systems and of regular grammars is different and the observer can stress, and use, such difference. Specifically, a grammar can only write a terminal once, while the sticker system can do so twice – once in the upper strand, and once in the lower. Thus the workspace is rewritable in a limited manner while for a regular grammar the workspace is essentially only its one non-terminal.

4 Computing by Observing Changes

The relatively easy achievement of computational completeness and universal grammars, presented in the previous two sections, suggests considering even less powerful observers. Natural candidates for such restrictions are ones that

are suggested by limitations of physical observation of biological systems. Our goal here is to take another step in the direction toward more realistic systems without narrowing down the architecture too much to fit just one specific physical setup. We do this based on the observation that it is an unrealistic assumption to think that the observer could read the entire configuration in every step. Technical reasons, a lack of time, and other factors will prevent this in most settings. Therefore in reality we deal here with observers that can only read one specific measure derived from the system's configuration; this could model for example temperature, volume etc. which are changed by the ongoing process and might be observable without any intrusion into the system itself.

When assigning some type of observable measure, we have two obvious choices of where to assign it to: to the single objects that compose the configurations of the observed system or to the rules that govern the transitions of the observed system. With these two choices, we could abstract the observation of energy consumed or freed in a certain reaction, or the change of quantity (mass) observed during a certain transformation. However, in this paper we are interested only in the theoretical consequences of such restrictions, so we will not discuss the technicalities of real observation of mass or energy.

Going back to the idea of observing grammars presented earlier rules like $a \to bc$ and $a \to cb$ would effect the same change in the total change, because the objects on either side of the rules are the same in type and number. So measuring (observing) changes of quantities would not allow to distinguish the applications of these two rules. However, this can be done by assigning the observable measures directly to the rules: in fact, we could simply assign different values to these rules.

This very simple example already shows that assigning a measure to objects is a special case of assigning it to rules. In this later model we can always simulate the former by simply giving a rule $U \to V$ the value of the sum of values of objects in U minus the sum resulting from V. As they model very different approaches, we will still distinguish them as two variants.

We want to mention that there are three concepts in formal language theory that work in a somewhat similar manner although their motivations were completely different. On the one hand, there are *control languages*. An early version restricted to left-most derivations was introduced by Ginsburg and Spanier [12]. Here a grammar's rules are seen as an alphabet, and a derivation is successful only if the sequence of rules that are applied belongs to a certain language. Thus only some of the possible derivations are considered valid.

On the other hand, there are *Szilard languages* as introduced by Moriya [19]. Again the derivation rules form an alphabet, but in this case all derivations

are valid. However, the Szilard language consists of the sequences of rules of a grammar rather than the terminal words that are generated. It is known that Szilard languages can be more complicated than the ones generated in the conventional manner by the respective grammars. For example, context-free grammars can have non-context-free Szilard languages. For a survey of early results on both Szilard languages and control languages we refer to the book by Salomaa [25], Mäkinen gives a more recent overview of the literature on Szilard languages [17].

There are also similarities with valence grammars [22], where to each production an integer value is associated. This is called the valence of the production. Then only those derivations are taken into account for which the valences of the applied productions add up to zero. Thus the difference to our case is that we consider the trajectory of the changes of valences during a derivation instead of the their sum or aggregation. This provides more control over the sequence of rules, while a summation is commutative.

In what follows we shortly recall string-rewriting systems, that will constitute the observed systems. Then, in Section 4.2 we introduce change-observing acceptors which formalize the ideas explained above, and we illustrate them with some examples. Section 4.3 then provides characterizations of the computational power of several implementations of change-observing acceptors, most noteworthy we show that with inverse context-free string-rewriting they are equivalent to state grammars and thus also to matrix grammars.

4.1 String-Rewriting Systems and McNaughton Languages

We follow the notations and terminology as exposed by Book and Otto [3]. We only recall briefly the most basic notions needed here.

Definition 4.1 *A* string-rewriting system R *on an alphabet* Σ *is a subset of* $\Sigma^* \times \Sigma^*$. *Its elements are called* rewrite rules*, and are written either as ordered pairs* (ℓ, r) *or as* $\ell \to r$ *for* $\ell, r \in \Sigma^*$. *By* $\mathsf{Dom}(R) := \{\ell : \exists r((\ell, r) \in R)\}$ *and* $\mathsf{Range}(R) := \{r : \exists \ell((\ell, r) \in R)\}$ *we denote the set of all left-hand respectively right-hand sides of rules in* R.

The *single-step reduction relation* induced by R is defined for any $u, v \in \Sigma^*$ as $u \Rightarrow_R v$ iff there exists an $(\ell, r) \in R$ and words $w_1, w_2 \in \Sigma^*$ such that $u = w_1 \ell w_2$ and $v = w_1 r w_2$. The *reduction relation* $\overset{*}{\Rightarrow}_R$ is the reflexive, transitive closure of $\Rightarrow_R$. If the rewriting system used is clear, we will write simply $\Rightarrow$, omitting its name.

A string w is *irreducible* with respect to R, if no rewrite rule from R can be applied to it, i.e. it does not contain any factor from $\mathsf{Dom}(R)$. The set of all

such strings is denoted by $IRR(R)$.

By imposing restrictions on the set of rewriting rules, many special classes of rewriting systems can be defined. Following Hofbauer and Waldmann [15] we will call a rule (ℓ, r) *context-free* (*inverse context-free*), if $|\ell| \leq 1$ ($|r| \leq 1$). The class of rewriting-systems with only (inverse) context-free rules we denote by `CF` (`InvCF`). A system is *monadic*, if it is inverse context-free and for all its rewrite rules (ℓ, r) we have $|\ell| > |r|$. The class of monadic systems is denoted by `mon`. A rule (ℓ, r) is called a *painter* rule if $|l| = 1$ and $|r| \leq 1$. A string-rewriting system with only painter rules is called a *painter string-rewriting system.*

As previously shown in other frameworks, we are especially interested in seeing whether string-rewriting systems become computationally more powerful when observed. Therefore we need a characterization of their accepting power just by themselves. The most appropriate reference here are McNaughton languages. These were defined by McNaughton et al. [18], and later investigated in more detail by Beaudry et al. [2]. Finally, Woinowski formalized this in so-called *Church-Rosser language systems* [23]. We do not need to use the entire formalism of these systems here and therefore simply say that a language $L \subseteq \Sigma^*$ is a McNaughton language of a string-rewriting system R, iff there exist an alphabet Γ containing Σ, strings $t_1, t_2 \in (\Gamma \setminus \Sigma)^* \cap IRR(R)$ and a letter $Y \in \Gamma$ such that for every word $w \in \Sigma^*$ we have $w \in L$ if and only if $t_1 w t_2 \stackrel{*}{\Rightarrow}_R Y$. This is denoted by $L \in R$-`McNL`.

A class of string-rewriting systems $\mathbb{S}$ defines its corresponding *McNaughton family of languages* $\mathbb{S}$-`McNL` in the canonical way that $\mathbb{S}$-`McNL` consists of all languages accepted by at least one rewriting system from the class $\mathbb{S}$. Without restrictions, string-rewriting systems are computationally universal in this sense.

Theorem 4.1 *[2] The family of all McNaughton languages coincides with the class of recursively enumerable languages.*

Seen as a string-rewriting system, regular and context-free grammars are almost the same. The difference in the definition of the grammar types is just a positional restriction for non-terminals on the right-hand side of rules.

Definition 4.2 *A string-rewriting system $[\Sigma, R]$ is called* regular *if there is a partition (N, T) of Σ and a symbol $S \in N$ such that $[N, T, R, S]$ is a regular grammar. A string -rewriting system is called* inverse regular*, if it is the inverse of a regular system.*

We now proceed to define the formal implementation of the ideas described in the preceding section. In contrast to the implementations of the computing by observing paradigm presented in Sections 2 and 3, here we do not need explicit general observers, since the general paradigm is here much more simplified. The observation is not derived from reading the observed system's configuration with a device; rather the change in change effected by the application of a rule can form the observation directly.

Definition 4.3 *A* change-observing acceptor *is a tuple* $\Omega = (\Sigma, P, \mathcal{O}, F)$*, where* Σ *is the input alphabet,* P *is the rule set of a string-rewriting system over an alphabet that contains* Σ*,* F *is a language that is called the* filter*, and the* Observer $\mathcal{O}$ *is a mapping from* P *to the alphabet of* F*.*

So the observer $\mathcal{O}$ assigns a letter to each rule in P (this is a very restricted version of the observers defined in Section 2.1). In what follows we explicitly assign to each rule the associated observation, e.g., $(a \to c, 1)$, where 1 is the observed change of energy.

If we take a filter over an alphabet, e.g., $\{-2, -1, 0, 1, 2\}$, this explicitly reflects the change in some quantity that the rule effects. The functioning of a change-observing acceptor is as follows. The input word is processed by the string-rewriting system P. Every time a rule r is applied, $\mathcal{O}(r)$ is appended to a word called the *observation*, starting from the empty string. The input word is accepted if the string-rewriting system halts and the observation is an element of the filter. Otherwise the input word is rejected.

We illustrate the definitions and the operation of change-observing acceptors by two examples.

Example 4.1 *Consider a test tube, in which two types of reactions can occur: one consumes (i.e. binds) energy, the other one sets energy free. Good input words for this system are ones that can be processed completely; this means they do not need to consume more energy than is present in the system initially. Furthermore, their processing should not lead to excessive setting free of energy, because this might lead to technical problems – for example in the form of explosions.*

We model this situation by the following change-observing acceptor: the input alphabet is $\{b, c\}$*, the string rewriting system with the energy values already assigned to the rules is* $\{(b \to a, 1), (c \to a, -1)\}$*; thus processing the symbol* b *sets one unit of energy free (symbol* 1 *is observed when* $b \to a$ *is executed in the observed system), while processing of* c *binds one unit (symbol* -1 *is observed).*

Let k be the constant such that the energy balance should never be greater than k and never be smaller than $-k$. Rule sequences are restricted to this form by setting the decider to the language

$$\{w : \textit{for every prefix } u \textit{ of } w \textit{ there is } \left||u|_1 - |u|_{-1}\right| \leq k\}.$$

This decider language is clearly regular. The language accepted by the system, however is

$$\{w : ||w|_b - |w|_c| \leq k\}.$$

Thus in prefixes of words in the language there can be arbitrarily big imbalances between bs and cs, and the resulting language is known not to be regular. So we can accept non-regular languages with very simple string-rewriting systems and regular decider. In this case the reason is that the string-rewriting system does not need to process the input string from left to right but can jump back and forth arbitrarily. In a very straight-forward manner, this method can be extended to more than two letters, and in this way even non-context-free languages can be accepted.

Next we take a look at a string-rewriting system that is somewhat more complicated, because the rules have left sides of length greater than one. In this way, the relative position of different letters in the input string can be checked.

Example 4.2 *The input alphabet is $\{a, b, c\}$, and we assign to the three letters the weights 1, 2, and 3 respectively. In this way the results of the observation of the rules are not assigned arbitrarily, but are computed from the weight of the letters that are involved, as the difference between the right side's and left side's total weight of the rules. Furthermore the working alphabet contains a letter A of weight 6. The string-rewriting rules are the following, where we explicitly give the weight for ease of reading: $(aa \to a, -1)$, $(bb \to b, -2)$, $(cc \to c, -3)$, $(abc \to A, 0)$, $(a \to A, 5)$, $(b \to A, 4)$, $(c \to A, 3)$. As in the previous example, we associate to each rule the observed change of energy.*

The filter applied to the observations is the language $(-1-2-3)^*0$. The iteration of complete blocks $(-1-2-3)$ ensures that the same number of all three letters is deleted. The final 0 checks the structure of the original word, i.e. whether there were first as followed by bs, then cs. The other rules have the function of checking that no other letters were present; they would be rewritten before the system halts and thus cause the observation of a positive number.

So the language accepted by the observation of this monadic string-rewriting system is $\{a^n b^n c^n : n > 0\}$, which is not context-free. We strongly believe that this language cannot be accepted using the simpler painter rules employed in Example 4.1.

These initial examples have demonstrated that observation as defined above can increase the computational power of the total system compared to the

power of the single components. We now proceed to characterize more exactly this computational power for specific variants of change-observing acceptors.

4.3 The Computational Power of Observing Change

To start with, we investigate the case of the very simple painter systems, which can only replace one symbol by another,

Proposition 4.1 *The class of languages accepted by change-observing acceptors with painter string-rewriting systems is not in any set-theoretic inclusion relation with the classes REG and CF.*

Proof. Example 4.1 has already shown that change-observing acceptors with painter string-rewriting systems can accept non-regular and even non-context-free languages. On the other hand, these acceptors cannot distinguish between two words like ab and ba, which consist of the same letters but in different order. The letters a and the b are always rewritten independent of their relative position to other letters, because painter rules have left sides of length one. This means that every sequence of rules that is applied to ab can also be applied to ba. While the resulting strings will in general be different, the acceptance of a change-observing acceptor depends only on the sequence of rules that is applied. Thus any change-observing acceptor with painter string-rewriting system that accepts ab will also accept ba, and $\{ab\}$ is a finite language that cannot be recognized by any acceptor of this type. □

So the class of languages accepted by change-observing acceptors with painter string-rewriting systems is somewhat orthogonal to the lower part of the Chomsky hierarchy. This changes when we use inverse context-free string-rewriting. We obtain a nice characterization of a language class well-known from the theory of regulated rewriting. To show this, we first need to recall state grammars that were introduced by Kasai [24], see also the book by Dassow and Păun [11].

Definition 4.4 *A* state grammar *is a tuple* $[Q, N, T, R, q_0, S]$ *where* N *is a set of non-terminals,* T *is a set of terminals, and* $S \in N$ *is the start symbol as for general phrase structure grammars;* Q *is a set of states,* $q_0 \in Q$ *is the start state, and the rules in* R *are from* $Q \times (N \cup T)^+ \mapsto Q \times (N \cup T)^*$.

A configuration of a state grammar consists of a sentential form plus a state. Rules can be applied if the state on their left-hand side coincides with the one in the configuration and their component from $(N \cup T)^+$ is a factor of the current sentential form; then the state is changed and a rewriting step is done in the obvious way. A word of terminals is generated by the grammar if it can be derived from the configuration $[q_0, S]$.

Thus state grammars are somewhat of a hybrid between grammars and automata. It may not surprise that they find a counterpart in the computing by observing paradigm, since both consist of a rewriting system and a finite-state control with some influence on the types of derivations that are permitted.

Theorem 4.2 *The class of languages accepted by change-observing acceptors with inverse context-free string-rewriting systems is equal to the class of languages generated by state grammars with context-free rules.*

Proof. Let $G = [Q, N, T, R, q_0, S]$ be a state grammar. We construct the change-observing acceptor that accepts the language generated by G as follows. The input alphabet is obviously T, the string-rewriting system will work over $T \cup N$, just as the grammar's rule set. In fact, it will mainly consist of R's reverse without the states, i.e. the set $P := \{u \to v : (v, q_1) \to (u, q_2) \in R\}$.

For convenience we may assume without loss of generality that S appears only on the left-hand side of R's rules; this means it is rewritten in the first step and does not occur in the sentential forms later on. Similarly, we assume that q_0 only occurs on the left-hand sides of rules rewriting S and thus is never returned to during any computation. The observer maps every rule to the state on the left-hand side of the original. That is, a rule $v \to u$ originating from $(v, q_1) \to (u, q_2)$ is mapped to q_1. Notice that here possibly several rules from R might result in the same rule in P, if the string-rewriting is the same while the states are different. In this case the observation is the set of all states on the left-hand sides of possible original rules. Thus the set of observations is a subset of the powerset of G's state set.

Finally, we add rules $x \to x$ for all symbols of $T \cup N$ except S. All of these rules are mapped to a special symbol $\perp$ by the observer. The function of this is to ensure that in the end there is only S, and all other symbols have been reduced to it; otherwise the system will simply not stop. A similar technique has been used in Example 4.2.

It remains to define the filter. It is derived from a slightly modified state-transition graph of G. This graph contains one node for every state of G. There is an edge from q_1 to q_2 iff there exists some rule on R changing state q_1 to state q_2. These edges receive as a label the state of their origin, in the example q_1. In the case described above where several rules from R result in the same rule in P, because the string-rewriting is the same while the states are different, also here we use the set of all states on the left-hand sides of possible original rules to label all of these transitions. We can look at this as a finite automaton accepting a language over the alphabet Q, all states are final except for q_0. Let us call this language F'. Since the grammar generates a word while the change-observing acceptor receives it as input, they work in somewhat the opposite direction. Therefore our filter will be $F := F'^R$.

Thus an input word is accepted if and only if there is a derivation for it under the state grammar G. The derivation and the computation by the corresponding change-observing acceptor are in such close correspondence that no formal proof of this fact seems necessary.

For a given change-observing acceptor $\Omega = (\Sigma, P, \mathcal{O}, F)$ we construct the state-grammar $G = [Q, N, T, R, q_0, S]$ as follows. Again, the direction of work for the two constructs is somewhat the reverse. While the grammar generates the word, Ω receives it as input and then processes it. This processing ends in an arbitrary irreducible word of P, while the grammar always starts from a distinct start symbol. So in a first phase, we let the grammar generate an arbitrary string from $IRR(P)$. It is well-known and easy to see that the set of all irreducible strings of a finite string-rewriting system is always regular. Therefore $IRR(P)$ can be generated by state grammars in a straight-forward manner, and we do not elaborate the details of this. Rather we suppose without restriction of generality that G starts from an arbitrary word from $IRR(P)$ instead of S.

G's rule set is just the reverse of P, i.e. the set $\{u \to v : v \to u \in P\}$. The states of G will ensure that only rules admissible under Ω's filter will be applied. But because the grammar works in the reverse direction, the reverse F^R is used. Let $\mathcal{O}'$ be a finite automaton for this language. Essentially the state set and transition function of G will be the same as for $\mathcal{O}'$. However, state grammars do not employ final states. Therefore all prefixes of words in F^R would also generate words. Therefore we introduce one more state q_f and consider all letters appearing in rules of P as the alphabet of non-terminals.

The grammar's terminals are copies of the letters in Σ marked by one prime. To R we add the following productions:

$$\{(x, q) \to (x', q_f) : x \in \Sigma, q \text{ is a final state of } \mathcal{O}'\}$$

and $\{(x, q_f) \to (x', q_f) : x \in \Sigma\}$. The transitions from the first set give the grammar the possibility to exit the simulation of Ω, if an accepted filter word has been traversed. After this q_f only permits the derivation of all letters to terminals. In this way only words that consisted only of input letters from Σ can lead to successful derivations, and only possible input words of Ω can be generated.

Except for the initial generation of a word from $IRR(P)$ and the final derivation to terminals, the steps of the grammar and the change-observing acceptor are in exact one-to-one correspondence. Therefore we do not consider more arguments necessary to show the languages generated and accepted by the two constructs are equal. $\square$

It is worth mentioning that the class of languages characterized in Theorem 4.2 is maybe best known as the class of languages generated by matrix grammars, which are equivalent to state grammars [11]. Further, it is interesting to compare the results presented here with the corresponding ones for the Monadic Transducers from Section 2.1 as observers. In that case painter systems and inverse context-free systems lead to the same power, namely all context-sensitive languages are recognized with either type [8]. Thus using such restricted observers we lose computational power, especially for painter systems.

5 Research Perspective on Computing by Observing

Computing by observing is applicable to discrete systems in a very general way, as long as an appropriate observer can be constructed. This is rather straightforward for most models in Theoretical Computer Science. As soon as a simple way of mapping its configurations, or transitions, to single symbols is found, any such system can be the base of a computation by observation. In this respect we remark that for systems where configurations are not finite or that are characterized by infinite behaviors, e.g., cellular automata, computing by observing could be the natural way to investigate them as computing devices. However, not every type of system is equally adapt for achieving great computational power. Actually, Theorem 2.2 shows that even a decrease in power can occur. Here the crucial factor seems to be space that can repetitively be rewritten; if it is finite, only finite-state computations seem possible.

Besides investigating further specific systems with respect to their power, when observed, it seems very interesting to try and formalize the thoughts of the preceding paragraph and to try to identify other such crucial factors. This could eventually lead to a type of measure for the capacity of systems to process information. On the other hand, limiting these crucial resources would lead to complexity hierarchies. A first result in this direction is that a linear space bound on a grammar's sentential form in an otherwise computationally complete model characterizes the context-sensitive languages [7].

Another major direction for future research consists in studying how realistic limitations on the observers change the already obtained results.

In this respect, in Section 4, we have introduced a proposal for more realistic yet very general observers, called change-observing acceptors, that, in contrast to the observers used so far, do not need to read the entire configuration of the observed system, rather they just notice a change in some specified measurable quantity. In this way they have constant running time and do not need to physically interfere with the underlying system. In this they are much closer

to real recording devices.

However, the presented systems still rely on a few assumptions that might not be easy to meet in reality. The division into discrete steps seems like one, because biochemical systems typically do not follow any clock. However, we do not require the single steps to have equal length. In the model using change-observing acceptors, where only certain changes are recorded, one step can simply be the variable time between one event and the other. A problematic case is the one where this time is zero, i.e. two or more events take place in parallel. For example, this would mean an infinite number of possible observations if we observe energy being consumed or set free like in Examples 4.1 and 4.2, namely all integers would be possible observations. Many computational models work with strong parallelism, i.e. all reactions that can take place have to take place in a given step. It would be a challenge to devise an implementation of the computing by observing paradigm that is able to deal with a situation where any number of reactions can occur in every step.

More general applications of the paradigm include the possibility to formalize the notion of "abstraction", comparing with other notions present in computer science, e.g., abstract interpretation, [10]. In fact, the process of observation, as defined here, collapses the state-space of the basic system into a smaller one, doing a sort of abstraction on the observed system (keeping some specific information, while ignoring others). The observed behaviors are then "upper approximations" of the behaviors of the basic system. Two interesting problems should be investigated: (i) how to guarantee that the observed behaviors maintains some specific properties of the basic system (e.g., if the basic system oscillates then this is maintained in the observed behaviors); (ii) given the basic system and the observer, how to provide, in an algorithmically efficient manner, a finite description (e.g., a grammar) of the sets of observed behaviors.

One more application of the paradigm could be in the area of quantum computation. There the concept of observer is crucial, and actually, a major problem in quantum experiments is to design the correct observer. However no notion of computational complexity of the observer has been formally defined. In the proposed paradigm one could formalize this notion by explicitly dividing the observed system from the observer, as proposed in [4].

Acknowledgement. The authors are very much indebted to the anonymous referees for careful reading of the manuscript and for their elaborate and detailed comments.

References

[1] A. Alhazov, M. Cavaliere, Computing by observing bio-systems: The case of sticker systems, in: C. Ferretti, G. Mauri, C. Zandron (Eds.), DNA, Vol. 3384 of Lecture Notes in Computer Science, Springer, 2004, pp. 1–13.

[2] M. Beaudry, M. Holzer, G. Niemann, F. Otto, McNaughton families of languages, Theoretical Computer Science 290 (2003) 1581–1628.

[3] R. Book, F. Otto, String-Rewriting Systems, Springer, Berlin, 1993.

[4] C.S. Calude, M. Cavaliere, R. Mardare, Observer Complexity and Deutsch's Problem, submitted, 2010.

[5] M. Cavaliere, Computing by observing: A brief survey, in: A. Beckmann, C. Dimitracopoulos, B. Löwe (Eds.), CiE, Vol. 5028 of Lecture Notes in Computer Science, Springer, 2008, pp. 110–119.

[6] M. Cavaliere, P. Frisco, H. J. Hoogeboom, Computing by only observing, in: O. H. Ibarra, Z. Dang (Eds.), Developments in Language Theory, Vol. 4036 of Lecture Notes in Computer Science, Springer, 2006, pp. 304–314.

[7] M. Cavaliere, P. Leupold, Evolution and observation — a non-standard way to generate formal languages, Theoretical Computer Science 321 (2004) 233–248.

[8] M. Cavaliere, P. Leupold, Observation of string-rewriting systems, Fundamenta Informaticae 74 (4) (2006) 447–462.

[9] M. Cavaliere, P. Leupold, Complexity through the observation of simple systems, in: T. Neary, D. Woods, A. K. Seda, N. Murphy (Eds.), CSP, Cork University Press, 2008, pp. 23–34.

[10] P. Cousot, R. Cousot, Abstract interpretation frameworks, *Journal of logic and computation*, 2, 4, 1992.

[11] J. Dassow, G. Păun, Regulated Rewriting in Formal Language Theory, Springer-Verlag, Berlin, 1989.

[12] S. Ginsburg, E. H. Spanier, Control sets on grammars, Mathematical Systems Theory 2 (2) (1968) 159–177.

[13] M. A. Harrison, Introduction to Formal Language Theory, Addison-Wesley, Reading, Massachusetts, 1978.

[14] J.E. Hopcroft, J.D. Ullman, *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, 1979.

[15] D. Hofbauer, J. Waldmann, Deleting string rewriting systems preserve regularity, Theoretical Comput. Sci. 327 (3) (2004) 301–317.

[16] P. Leupold, How to make biological systems compute: Simply observe them, in: BIONETICS '08, ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), ICST, Brussels, Belgium, 2008, pp. 1–6.

[17] E. Mäkinen, A bibliography on Szilard languages, Bulletin of the EATCS 65 (1998) 143–148.

[18] R. McNaughton, P. Narendran, F. Otto, Church-Rosser Thue systems and formal languages, Journal of the ACM 35 (2) (1988) 324–344.

[19] E. Moriya, Associate languages and derivational complexity of formal grammars and languages, Information and Control 22 (1973) 139–162.

[20] L. Kari, Gh. Păun, G. Rozenberg, A. Salomaa, S. Yu, DNA computing, sticker systems, and universality. *Acta Informatica*, 35, 5, 1998.

[21] Gh. Păun, G. Rozenberg, A. Salomaa, *DNA Computing - New Computing Paradigms*. Springer-Verlag, Berlin, 1998.

[22] Gh. Păun, A New Generative Device: Valence Grammars. *Revue Roumaine de Mathématiques Pures et Appliquées*, 6, 1980.

[23] J. R. Woinowski, The context-splittable normal form for church-rosser language systems, Inf. Comput. 183 (2) (2003) 245–274.

[24] T. Kasai, An hierarchy between context-free and context-sensitive languages, Journal of Computer and Systems Sciences 4 (5) (1970) 492–508.

[25] A. Salomaa, Formal Languages, Academic Press, New York, 1973.