How to make Biological Systems Compute: Simply Observe Them

Peter Leupold
Department of Mathematics, Faculty of Science
Kyoto Sangyo University
Kyoto 603-8555, Japan
leupold@cc.kyoto-su.ac.jp

ABSTRACT

We survey work on the paradigm called computing by observing. This is a formal model of the way many biochemical experiments are conducted, where an external observer protocols the evolution of a system in a test tube or some other limited environment. So this observer, which is not acting on the input, produces the output. In this the paradigm contrasts strongly to many formal models of bio-computation that follow the classical input/output paradigm from computer science.

In the presentation we put a special focus on the identification of key features that make systems especially suitable for serving as the basis of a computer in the computing by observing architecture. It turns out that context-sensitivity can be simulated well by simpler mechanisms, while unboundedly reusable workspace plays a key role.

Keywords

Computing by Observing, Unconventional Computation, Computational Completeness

1. INTRODUCTION

The majority of formal models of computation via biochemical processes have been defined by computer scientists. Therefore it does not come as a surprise that they basically follow the dominant architecture in computer science, the input/output paradigm. Some data is fed into a system, which processes this input according to its set of rules and transforms it into an output, which is then available to the user. The main difference between current computers and these new models of computation is that the data consists of molecules, very often DNA strands, and the processing is done by biochemical reactions rather than by electrical processors working on digital data.

Many experiments in biology or chemistry, however, do not produce their results along these lines. There, it is a standard proceeding to conduct an experiment, observe the entire progress of a system, and then take the result of this observation as the final output. Consider for example the use of a catalyst in a chemical reaction. The reaction might produce the same result, but the energy used or the time required might be less than without the catalyst. This difference only becomes clear when looking at a protocol of how the quantifier in question evolved during the course of the experiment. Another classical example would be the equilibrium between hunter and prey. Any snapshot of the relation of their numbers in one moment would be rather meaningless, only the dynamics between the two numbers over time show their interdependence.

These two examples share several differences to the models of computation described above. First, it is not the system that evolves that produces the interesting output. Rather, some external observer has to look at the state of the system during all of its evolution and record these states, or at least some of their characteristics at each step. This protocol is the result. Second, this result is in a format readable to humans and has nothing to do with the format of the input, which in the examples above were test tubes full of chemicals or a biosphere with animals in it.

The paradigm of "computing by observing" tries to capture these two features common in conducting experiments in biochemistry. The key feature here is the introduction of an observer between on the one side the system that processes the input and on the other side the output. This observer must have the capacity of reading all or part of the current configuration and of outputting a single letter determined by what has been read. As we study only underlying systems that evolve in discrete steps, the observer will read the configuration of every step exactly once. Figure 1 depicts this architecture schematically.

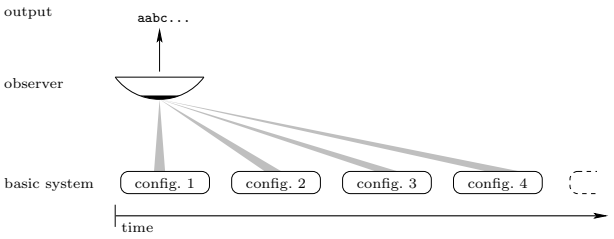

Figure 1: Basic structure of the "Computing by Observing" architecture.

An evident advantage over many models is the fact that the output of such a system can be in a format that is compatible with humans or with conventional information pro-

Bionetics '08, November 25-28, 2008, Hyogo, Japan

cessing devices. Here we will map all possible configurations of the underlying system to a single letter. The finite alphabet of these letters can be seen as a set of classes into which the configurations are categorized. The temporal succession of configurations from these classes then simply becomes a string over the alphabet, and strings are actually the basic format of information used in electronic information processing and also in written human communication.

We now proceed to justify our choice of string-rewriting systems as underlying systems and then present some implementations of these in the computing by observing architecture.

2. WHICH SYSTEMS TO OBSERVE

At a first glance it might seem especially promising to observe formal models of biochemical processes in order to be close to possible real implementations. And indeed this has been one approach and several models of biologically inspired computation such as sticker systems [1], splicing systems [4], and membrane systems [5] have served as the underlying systems of instantiations of the computing by observing paradigm. The monograph by Păun et al. [11] and more recent research would offer a wide range of other models still to explore.

However, all of these are idealized models of just isolated phenomena and thus do not seem to be much closer to biochemical reality than purely abstract models. On the other hand, the fact that they mainly model one kind of reaction or mutation results in systems that use mainly one very special kind of transformation. As long as it is unclear whether any of these models will ever evolve towards an actual, physical realization, there is no reason to focus on this very specialized type of systems. Rather, it seems to be promising to keep the systems on a very abstract level and to try to identify features that are important to turn them into powerful computing devices in connection with an observer. At the same time some features might be found not to increase the power at all.

The identification of such features would then make it easier to select among biochemical systems that are actually understood and controlled well enough to be able to use them in an actual implementation of the computing by observing paradigm. Therefore we will not consider formal models of biochemical reactions as underlying systems here, rather we will use a very basic and general model of information processing, string-rewriting. Here, information is represented in a string of symbols and it is processed by letting rules change, insert and delete these symbols. These are basically the type of operations we expect to perform with biological systems via reactions that change some chemical, and via attaching or removing new molecules to or from others. In this sense, string-rewriting seems to be a good abstract model.

When judging how much a given string-rewriting system gains in power, when it is used in the computing by observing architecture, we have to compare the obtained power to that of the string-rewriting system (and the observer) by itself. We will not make this very explicit in what follows. Therefore we state here that usually we will refer to the power of a class of string-rewriting systems when used in the well-known form of generative grammars in the sense of Chomsky. However, we also want to mention the possibility of McNaughton languages as introduced in [10]; these are especially appropriate for length-reducing systems that do not generate anything when used as grammars.

3. OBSERVING STRING-REWRITING

As stated in the preceding section we now proceed to explore the possibilities of observing string-rewriting systems. On the one hand, they are very simple and basic devices, just taking a part of a string and replacing (rewriting) it by a different string. On the other hand, they can simulate almost any other known device that computes on strings in very natural ways. Thus they combine simplicity in their definition with great expressive power and therefore appear as good candidates for underlying systems, when we want to investigate what the key features are that can make a system suitable for being used in the computing by observing architecture. The results of this section are essentially derived from [6].

3.1 String-Rewriting Systems

Terms and notation from general formal language theory, logic and set theory are assumed to be known by the reader and can be consulted in standard textbooks like the ones by Harrison [9] or Rozenberg and Salomaa [12]. In our notation on string-rewriting systems we mostly follow Book and Otto [2] and define a *string-rewriting system (SRS)* R on an alphabet Σ to be a subset of $\Sigma^* \times \Sigma^*$. Its single-step reduction relation is defined as $u \rightarrow_R v$ iff there exists $(\ell, r) \in R$ such that for some u_1, u_2 we have $u = u_1 \ell u_2$ and $v = u_1 r u_2$. We also write simpler just $\rightarrow$, if it is clear which is the underlying rewriting system. By $\stackrel{*}{\rightarrow}$ we denote the relation's reflexive and transitive closure, which is called the *reduction relation* or *rewrite relation*. Note that we also use the notation $u \rightarrow v$ for rewrite rules, mainly when speaking about rules in a natural language sentence to make it graphically clear that we are speaking about a rewrite rule and not some other ordered pair. All the SRSs in this article will be finite.

A derivation of a string-rewriting system terminates, when it reaches a string that cannot be rewritten anymore. This means that no left-hand side of any rule is contained in this string. This type of string is called irreducible or terminal. They will play a crucial role in what follows, because reaching a terminal string will signal the end of our computations. We will also call single symbols terminals, if there does not exist any rule that has them as its left-hand side.

3.2 The Observers: Monadic Transducers

A string-rewriting system's states are the strings of its derivations. So for the observer we need a device mapping arbitrarily long strings into just one singular symbol. We use a special variant of finite automata with some feature known from Moore machines 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. Because we find it preferable that the observation of a certain string always lead to a fixed result, we consider here only deterministic and complete automata. The name monadic transducer for these devices is motivated from monadic string-rewriting rules; these can have arbitrarily long left sides that are rewritten to strings of length one or zero.

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.

3.3 Rewriting/Observer Systems

We now define the central concept of this article, the implementation of the computing by observing architecture consisting of a string-rewriting system and a monadic transducer. A *Rewriting/Observer (R/O) system* is a pair $\Omega = (G, A)$ constituted by a string-rewriting system G over an alphabet Γ and a monadic transducer (observer) A with output alphabet Σ, which then is also the output alphabet of the entire system Ω. The transducer's input alphabet must be the alphabet Γ of the string-rewriting system so that it can read all the strings of the derivation. Γ will always contain a special start symbol S from which all derivations start, i.e., the first string of all derivations consists of the single symbol S.

Several modes of generation can be defined (see, e.g., [6]). Here we will consider the mode of generation that allows the observer to write an empty and non-empty output in an arbitrary manner (*free R/O systems*); i.e., the transducer can either output one letter or the empty word and freely alternate between these two options. A free R/O system generates a language in the following manner:

$$L_f(\Omega) = \{A(\quad w_0, w_1, \ldots, w_n) : \\ S = w_0 \to w_1 \to \ldots \to w_n,\ w_n \in \Gamma^*\}.$$

In other words, the language contains all those words which the observer can produce during the possible terminating derivations of the underlying string-rewriting system. Those derivations that do not terminate do not produce any valid output; this means that we only take into account finite words. Of course, by considering the other case of non-terminating derivations the R/O systems could also be used to generate languages of infinite words.

We also consider the variant where we define as 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.

For a class $\mathcal{G}$ of string-rewriting systems and a class $\mathcal{O}$ of observers, we denote by $\mathcal{L}_f(\mathcal{G}, \mathcal{O})$ and $\mathcal{L}_{\perp,f}(\mathcal{G}, \mathcal{O})$ the classes of all languages that are generated by R/O systems with string-rewriting systems from $\mathcal{G}$ and observers from $\mathcal{O}$ in the respective modes.

3.4 Simulating Context-Sensitivity

An important ability of computing devices like Turing Machines is their ability to coordinate what is happening in different parts of their memory. This makes possible conditional statements reading one bit of information in one place and changing another bit depending on the value of the first. A very basic way of doing this are context-sensitive rewriting rules. For example, via rules $ab \to ad$ and $cb \to ca$ one can realize the choice: if after a b is preceded by an a, replace it by d, if it is preceded by c, then replace it by a.

A Turing Machine can implement such a rule $ab \to cd$ with two simple rewrite rule $a \to c$ and $b \to d$ and by remembering between the two steps that the first one has been realized and that the second one still has to be enacted before anything else is done. If we have only these rules but no explicit control on them we need to employ more elaborate techniques to implement context-free rewriting.

We illustrate this for the example of $ab \to cd$, where the simple rules to simulate it are $a \to c'$, $b \to d''$, $c' \to c'''$, $d'' \to d$, and $c''' \to c$.

To do the simulation, the system should go through a sequence of derivation steps

$$ab \to c'b \to c'd'' \to c'''d'' \to c'''d \to cd.$$

The problem here is to ensure that the five rules implied are applied in this order and that nothing is rewritten in any other place of the string. Ensuring this is the task of the observer. It does this by accepting only strings that that are marked with apostrophes in one place of the string and by accepting only those combinations that appear in the derivation shown above. The mapping realizing this is

$$\mathcal{A}(w) = \begin{cases} + & \text{if } w \in \Sigma^*, \\ & \text{or if } w \in \Sigma^* c'b\Sigma^*, \\ & \text{or if } w \in \Sigma^* c'd''\Sigma^*, \\ & \text{or if } w \in \Sigma^* c'''d''\Sigma^*, \\ & \text{or if } w \in \Sigma^* c'''\Sigma^*, \\ & \text{or if } w \in \Sigma^* c'b\Sigma^*, \\ - & \text{else.} \end{cases}$$

It is relatively straight-forward to see that the application of any rule in a different manner will lead to the output of $-$, and we can interpret this as signaling that something has gone wrong.

This very simple example shows how to simulate context-sensitivity with very simple rules and a regular observer. The rewriting rules employed here do nothing else but rename the letters. If we think of their values as colors, then the rules just repaint them, and this motivated the name of *painter* systems for systems consisting only of this type of rule. The class of all these systems is called denoted by PA.

With basically the same technique it is possible to simulate linearly bounded automata and to accept all context-sensitive languages in a slightly different framework from the one presented here.

THEOREM 1. *[8]* $\mathcal{L}(PA, FA_O) = CS$.

Essentially the setup of the system is the same as here. However, the input is a word to be accepted rather than the start symbol S, and the output is not the word created by the observer, but rather a yes/no answer based on the membership of this word in a regular language.

3.5 Expanding the Workspace

In general, context-sensitive systems can do all the computations that Turing machines can, only with the restriction that the workspace used cannot be much bigger than the space occupied already by the input. This suggests that the combination presented in Theorem 1 can become computationally complete, if it is enhanced with a way of expanding its workspace unboundedly.

In terms of string-rewriting rules, the easiest way of doing this are context-free rules. These have left-hand sides of length one, just like the painter rules from Section 3.4, but their right-hand sides can be of arbitrary length. So by using

rules of the form $a \to bc$ we can increase the length of the string that is being processed and thus increase the available workspace.

We now proceed to illustrate in an example how to coordinate things happening in different parts of the string over longer distances than in the example of Section 3.4. Also we will show how to extract a string produced by the underlying system to the level of the output by successively marking its letters from one side to the other.

Example 1. We consider an R/O system with the output alphabet Σ. The underlying string-rewriting system works over the alphabet $\Sigma \cup \underline{\Sigma} \cup (\{1,2\} \times \Sigma) \cup \{1'\}$ where $\underline{\Sigma}$ is a copy of Σ where all letters are underlined.

The initial rule is $S \to 12$. After that the system will write a letter $x \in \Sigma$ to the left of the symbol 1 and then write the same letter to the left of 2. This happens via the rules $1 \to x1_x$, $2 \to x2_x$, $1_x \to 1'$, $2_x \to 2$, and $1' \to 1$ that are defined for all $x \in \Sigma$. Finally there are rules $1 \to \lambda$ and $2 \to \lambda$ deleting the symbols 1 and 2 and rules $x \to \underline{x}$ for all $x \in \Sigma$. The former two rules end the process of producing the word that will be out put afterwards by marking it with the underline by the latter type of rules.

The observer checks whether the rewrite rules are applied in the desired order, otherwise it outputs $\perp$ which means that the computation is stopped and considered as unsuccessful. The mechanism is essentially the same as in the preceding section, and the mapping that is used for this observer $\mathcal{A}$ is

$$\mathcal{A}(w) = \begin{cases} \lambda & \text{if } w = S, \\ & \text{or if } w \in \Sigma^* 1 \Sigma^* 2, \\ & \text{or if } w \in \Sigma^* 1_x \Sigma^* 2 \text{ for } x \in \Sigma, \\ & \text{or if } w \in \Sigma^* 1_x \Sigma^* 2_x \text{ for } x \in \Sigma, \\ & \text{or if } w \in \Sigma^* 1' \Sigma^* 2_x \text{ for } x \in \Sigma, \\ & \text{or if } w \in \Sigma^* 1' \Sigma^* 2 \text{ for } x \in \Sigma, \\ & \text{or if } w \in \Sigma^* 2 \text{ for } x \in \Sigma, \\ x & \text{if } w \in \underline{\Sigma}^* \underline{x} \Sigma^*, \\ \perp & \text{else.} \end{cases}$$

It is rather straight-forward to see that the language generated in this way is the non-context-free $\{ww : w \in \Sigma^*\}$.

Here the string is first generated by the underlying system and the observer controls that the computation happens only along the intended lines. So always the same letter is written in front of the 1 and in front of the 2 such that we will finally obtain a square ww. During this phase nothing is written in the output. Then in a final phase the generated string is extracted by marking it letter by letter from the left to the right and by outputting in every step the last letter marked. Since none of the rewrite rules of our string-rewriting system has underlined symbols on its left-hand side, the derivation reaches a terminal string when all letters are underlined and the computation stops.

After Theorem 1 it is very straight-forward to show how to simulate the work of a general Turing Machine on an input with context-free rules using the control techniques exhibited in Example 1. First, the workspace is expanded by attaching arbitrarily many blank symbols to the input string. Then the actual processing of the input starts, and any computable function can be computed in this way. The underlying string-rewriting system has to be context-free, where the really context-free rules are only used to create workspace in the beginning. The actual computation of the Turing Machine is then simulated by the same kind of painter rules as in Section 3.4.

THEOREM 2. $\mathcal{L}_{\perp,f}(CF, FA_O) = RE$.

So with a context-free and a regular component we obtain computational completeness. This is an immense increase in power compared to the two components taken by themselves and can therefore be seen as proof that the computing by observing paradigm is quite a powerful architecture and achieves one of its goals: combining simple systems in such a way that they can do significantly more than by themselves.

Actually, these components are so simple, and the obtained power so big that on should consider somehow limiting the power that comes from their interaction. Since all systems of this type will remain within the confines of computable functions, computational completeness is already the maximum that can be achieved. Arriving there so easily suggests that we allow the observer too much control. For example, not letting it read the entire configuration in every step might be a good restriction. In reality this might simply not be technically feasible, or for big configurations time might not suffice to scan everything.

3.6 Limits of Observation

Section 3.5 has shown us the crucial importance of the amount of workspace that is available for rewriting. We will now take a brief look at another result underlining this importance. Namely, we consider a constant bound on the amount of rewritable workspace. We call a string-rewriting system nonterminal k-bounded context-free if it is a context-free system and we impose the following restriction on its derivations: at any given step, there may not be more than k nonterminal symbols in its string. A system is called nonterminal bounded context-free, if there exists some k for which it is nonterminal k-bounded context-free.

Thus we do not limit the space that the systems configurations can occupy, but we limit the amount of this space that can be rewritten. Example 1 has shown how we even needed to be able to rewrite symbols (by underlining them) to be able to really read them with the observer; without the possibility to mark them in this way, it cannot know what has been read yet and the information contained in the string cannot be handled well.

So nonterminal k-bounded context-freeness means that a big part of the current configuration will actually be unchanged until the end of the computation, i.e. it will be present in every one of the following steps, and thus in some sense the information it contains cannot influence the computation anymore. So in this sense we have only a finite amount of workspace. This intuitively explains the following result that states that we can only obtain regular languages in this way.

THEOREM 3. *[3] For every R/O system $\Omega = (G, A)$ with G nonterminal bounded context-free, $A \in FA_O$, $L_{\perp,f}(\Omega)$ is regular.*

Thus we see that in this combination the composition of rewriting-system and observer is less powerful than the separate components. Of course, context-free systems even with a bound on the number of non-terminals in their sentential

forms can generate non-regular languages. Even with a single non-terminal it is possible, for example, to generate the language $\{a^n b^n : n \geq 0\}$.

So obviously not for every type of system it makes sense to integrate it in the computing by observing architecture, because by adding more machinery you can even loose computational power. The reason is, as explained above, that even unlimited space to be occupied by the string provides only a finite useful workspace, if it cannot be rewritten anymore.

Another implicit limitation of workspace that was considered in [6] is to oblige the observer to write a non-empty output in every step. Since, for example, a context-free string-rewriting system can only produce a constant number of new symbols in every step, this implicitly imposes a linear space bound on the R/L system observing such a string-rewriting system. Therefore only context-sensitive languages can be generated.

These systems still generate all the context-free languages. This can be seen easily via the Greibach Normal Form, that generates exactly one terminal per derivation step, and it is possible to let the observer output exactly this last terminal that was produced. However, it seems very improbable that these systems can generate all context-sensitive languages, because to use their space bound, they have to produce new symbols in every step.

Thus the symbols already in the string cannot be rewritten very much, and in this intuitive sense not much computation can be done. A rigorous proof of this has, however, not been done and thus the proper inclusion of always writing R/O systems with context-free string-rewriting and regular observer in the class of context-sensitive languages remains an open problem.

4. OUTLOOK

As the surveyed results show, the paradigm of evolution and observation is indeed quite universally applicable to discrete systems known in Theoretical Computer Science. As soon as a simple way of mapping its configurations to single letters is found, any such system can be the base of a computation by observation. However, not every type of system is equally apt for achieving great computational power. Actually, Theorem 3 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 in an evolution-observation-system, 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 string-rewriting systems sentential form in an otherwise computationally complete model characterizes the context-sensitive languages, see Theorem 1.

One feature that strings offer, but which will usually not be present in biochemical systems, is the linear order of letters. Clearly, a string 00101 contains more information than a multiset of three 0 and two 1. Over a binary alphabet like in this example, a string of length n can encode 2^n bits of information, while an unordered multiset of the same size has only $n + 1$ different states. It has been shown that this type of structural information is not always of actual use. Thus there is a stronger version of Theorem 2 stating the same power for systems observing a variant of context-free rewriting, where the right-hand sides of rules are in some sense commutative, so-called locally commutative context-free grammars [6]. However this does not mean that the same will be true for all forms of rewriting, and thus one sensible step would be paying more attention to multiset rewriting than to string-rewriting.

Finally, it is important to also notice the shortcomings of the presented paradigm, mainly the obstacles that will have to be overcome before any implementation can be thought of. For one thing, biochemical systems typically do not evolve in discrete steps that were used here to generate the single letters of the system. If we insist on digital output, then in some way appropriate intervals will have to be determined or each step must exhibit an observable signal showing that it is finished.

Further, it will in general not be possible to read the entire configuration of a system; only some parameters can be extracted. A special case of this problem the fact that about symbols in a string we can get exact quantitative information, while for chemicals often it will only be possible to test their presence or absence. The exact amount of some substance will normally not be possible to determine.

Most probably, implementations of the architecture will loose some of its power if these complications are taken into account. However, the same is true for most formal models of bio-computation; and since we have shown that here very simple components can lead to computational completeness, and in general gain power compared to their autonomous use, there is hope that such practical difficulties can be dealt with more easily in our framework.

5. ACKNOWLEDGMENTS

This work was done, while Peter Leupold was funded as a post-doctoral fellow by the Japanese Society for the Promotion of Science under number P07810.

6. REFERENCES

[1] A. Alhazov , M. Cavaliere: *Computing by observing bio-systems: The case of sticker systems.* 10^{th} International Workshop on DNA Computing, LNCS 3384, Springer, 2004.

[2] R. Book and F. Otto: *String-Rewriting Systems.* Springer, Berlin, 1993.

[3] M. Cavaliere, H.J. Hoogeboom, and P. Frisco: *Computing by only observing.* Proceedings Developments in Language Theory 2006, DLT 2006, LNCS 4036, Springer, 2006.

[4] M. Cavaliere, N. Jonoska, and P. Leupold: *DNA Splicing: Computing by Observing.* To appear in Natural Computing. Earlier version: *DNA Splicing: Computing by Observing.* In: DNA 11, Lecture Notes in Computer Science 3354, Springer-Verlag, Berlin, 2005, pp. 152–162.

[5] M. Cavaliere and P. Leupold: *Evolution and Observation – A New Way to Look at Membrane Systems.* In: Membrane Computing, International Workshop, WMC 2003, Revised Papers. Lecture Notes

in Computer Science 2933, Springer-Verlag, Berlin, 2004, pp. 70–87.

[6] M. Cavaliere and P. Leupold: *Evolution and Observation — A Non-Standard Way to Generate Formal Languages*. In: Theoretical Computer Science 321, 2004, pp. 233–248.

[7] M. Cavaliere and P. Leupold: *Evolution and Observation — A Non-Standard Way to Accept Formal Languages*. In: MCU 2004, Lecture Notes in Computer Science 3354, Springer-Verlag, Berlin, 2005, pp. 152–162.

[8] M. Cavaliere and P. Leupold: *Observation of String-Rewriting Systems*. In: Fundamenta Informaticae 74(4), 2006, pp. 447–462.

[9] M.A. Harrison: *Introduction to Formal Language Theory*. Reading, Mass., 1978.

[10] R. McNaughton, P. Narendran and F. Otto: *Church-Rosser Thue systems and formal languages*. In: Journal of the ACM 35, 1988, pp. 324–344.

[11] Gh. Păun, G. Rozenberg and A. Salomaa: *DNA Computing – New Computing Paradigms*. Springer Verlag, Berlin, 1998.

[12] G. Rozenberg and A. Salomaa (eds.): *Handbook of Formal Languages*. Springer-Verlag, Berlin, 1997.