Complexity Results for Deciding Networks of Evolutionary Processors[1]

Florin Manea

Institut für Informatik, Christian-Albrechts-Universität zu Kiel, D-24098 Kiel, Germany,
and
Faculty of Mathematics and Computer Science, University of Bucharest, Str. Academiei 14, RO-010014 Bucharest, Romania
e-mail: `flm@informatik.uni-kiel.de`

Abstract

The Accepting Networks of Evolutionary Processors (ANEPs for short) are bio-inspired computational models which were introduced and thoroughly studied in the last decade. In this paper we propose a method of using ANEPs as deciding devices. More precisely, we define a new halting condition for this model, which seems more coherent with the rest of the theory than the previous such definitions, and show that all the computability related results reported so far remain valid in the new framework. Further, we are able to show a direct and efficient simulation of an arbitrary ANEP by an ANEP having a complete underlying graph; as a consequence of this result, we conclude that the efficiency of deciding a language by ANEPs is not influenced by the network's topology. Finally, focusing on the computational complexity of ANEP-based computations, we obtain a surprising characterization of $\mathbf{P}^{\mathbf{NP}[\log]}$ as the class of languages that can be decided in polynomial time by such networks.

Keywords:
Networks of Evolutionary Processors, Normal Form, Computational Complexity, $\mathbf{P}^{\mathbf{NP}[\log]}$
2000 MSC: 68Q05, 68Q15, 68Q45

[1]An extended abstract of this paper was presented at 15th International Conference on Developments in Language Theory, DLT 2011 [12].

1. Introduction

The Accepting Networks of Evolutionary Processors (ANEPs, for short) are a bio-inspired computational model, introduced in [16], and having its roots in [9, 3, 4]. An ANEP consists in a graph having in each node a processor, which is able to perform very simple operations on words: insertion of a symbol, deletion of a symbol, and substitution of a symbol with another one. These operations are similar to the point mutations in a DNA sequence (insertion, deletion or substitution of nucleotides), thus, the processors are called evolutionary processor. Furthermore, each node contains data, which are organized in the form of multisets of words, each word appearing in an arbitrarily large number of copies, and all copies are processed in parallel such that all the possible events that can take place do actually take place. Following the biological motivation, each node may be viewed as a cell containing genetic information, encoded in DNA sequences which may evolve by local evolutionary events, i.e., point mutations. Moreover, each node is specialized in just one of these evolutionary operations. The nodes of a network are connected by edges, which can be seen as communication channels that allow the nodes to exchange data between them.

The computation of an ANEP is conducted as follows. Initially, only one special node, the input node, contains a certain word, the input word. Further, the computation consists in applying alternative evolutionary and communication steps on the words contained by the nodes. More precisely, in an evolutionary step, the words found in each node are rewritten according to the rules of that node. Rewriting a word in a node means that exactly one symbol of that word is changed (deleted, in the case of nodes specialized in deletion, or substituted with another symbol, in the case of nodes specialized in substitution) according to a rule of the node, or exactly one symbol is inserted in that word (in the case of nodes specialized in insertion); if there are more possibilities to rewrite a word, we obtain from the initial word a set of new words, containing all the possible words into which it can be rewritten. Moreover, if a rule cannot be applied to a word of a node, a copy of that word will remain in the node (although, it may be rewritten according to other rules of the same node, to obtain some new strings). In conclusion, the words obtained in an evolutionary step are exactly those derived by applying one rule to one of the words found before this step in the node, in exactly one of the possible places where that rule can be applied to that word; if there exists a rule and the node contains a word such that the rule cannot be applied to the word, we also preserve a copy of that word. In a communication step, the words of a node, obtained in the previous evolutionary step, are communicated to the other nodes, as permitted by some filtering condition associated with both the sending and the receiving node. Such filters are simply checking for the presence or the absence of a set of symbols in the communicated words. Going more into details, in such a step, a node tries to send a copy of each word it contains to its neighbours. A word leaves the node if it contains a set of permitting output symbols (specific to the sending node) and it does not contain any symbol of a set of forbidden output symbols (also, specific to the sending node). Then, the word enters the node to which it was sent if it contains a set of permitting input symbols (specific to the receiving node) and it does not contain any symbol of a set of forbidden input symbols (also, specific to the receiving node). After such a step, a node will contain the strings that were not able to be communicated to any of its neighbours and the strings that were communicated by its neighbours and were allowed to enter it. All the other strings are lost (in the environment). The classical definition assumes that a computation halts and accepts, when a word enters a special node of the network, the output node, or halts and rejects when the words contained in each node do not change in consecutive evolutionary or communication steps. In this paper we redefine the notions of halting computation, accepting computation, and rejecting computation. Starting from the idea that one should first define precisely which are the finite computations in a model, we propose a new halting condition for such a computation, that seems more coherent with the rest of the definition of the model. Namely, a computation halts (and is called halting or finite) as soon as at least one word enters the output node. Then, we split the halting computations into two categories: accepting computations and rejecting computations. The input word is accepted if the computation of the ANEP on this word is finite, and at least one word that is found in the output node at the end of the computation (we know that there is at least one word in that node, as the computation is halting) contains a distinguished symbol; otherwise, if the computation is finite, it is called a rejecting computation. All the other computations are infinite. The motivations behind this new halting, accepting and rejecting conditions are discussed in Section 3.

A series of works devoted to accepting networks of evolutionary processors appeared in the last decade (see [15] for a survey on this topic) and the results obtained concerned mostly the computational power, computational and descriptional complexity aspects, existence of universal networks, efficiency of these models viewed as problem solvers, and the relationships between different variants of such networks (e.g., simulations between them, comparison between their efficiency as problems solvers, etc.).

In this paper we see how part of these results change when the new halting condition is used. While the computational power of the ANEPs remains the same, the time complexity results are not preserved. To this end, we obtain a surprising characterization of $\mathbf{P}^{\mathbf{NP}[\log]}$ as the class of languages that can be decided in polynomial time by ANEPs. We also show that an arbitrary ANEP can be simulated efficiently by an ANEP with complete underlying graph. This answers an open question from [2] and shows that one cannot expect to decide a language faster using ANEPs with special topology, than in the case when complete ANEPs are used.

2. Basic definitions

We start by summarizing the notions used throughout the paper; for all unexplained notions the reader is referred to [18]. An *alphabet* is a finite and non-empty set of symbols. The cardinality of a finite set A is written $\mathrm{card}(A)$. Any sequence of symbols from an alphabet V is called *word* over V. The set of all words over V is denoted by V^* and the empty word is denoted by λ. The length of a word x is denoted by $|x|$ while $alph(x)$ denotes the minimal alphabet W such that $x \in W^*$. For a word $x \in W^*$, x^r denotes the reversal of the word.

In the following, we introduce a series of rewriting operations, called *evolutionary operations* as they may be viewed as linguistic formulations of local gene mutations. We say that a rule $a \to b$, with $a, b \in V \cup \{\lambda\}$ is a *substitution rule* if both a and b are not λ; it is a *deletion rule* if $a \neq \lambda$ and $b = \lambda$; it is an *insertion rule* if $a = \lambda$ and $b \neq \lambda$. The sets of all substitution, deletion, and insertion rules over an alphabet V are denoted by Sub_V, Del_V, and Ins_V, respectively.

Given a rule σ as above and a word $w \in V^*$, we define the following *actions* of σ on w:

- If $\sigma \equiv a \to b \in Sub_V$, then
  $\sigma^*(w) = \begin{cases} \{ubv \mid \exists u, v \in V^* \ (w = uav)\}, \\ \{w \mid w \text{ contains no } a\}, \end{cases}$
- If $\sigma \equiv a \to \lambda \in Del_V$, then

$$\sigma^*(w) = \begin{cases} \{uv \mid \exists u, v \in V^* \ (w = uav)\}, \\ \{w \mid w \in (V \setminus \{a\})^*\}, \end{cases}$$
$$\sigma^r(w) = \begin{cases} \{u \mid w = ua\}, \\ \{w \mid w \notin V^*a\}, \end{cases} \quad \sigma^l(w) = \begin{cases} \{v \mid w = av\}, \\ \{w\} w \notin aV^*\}, \end{cases}$$

- If $\sigma \equiv \lambda \to a \in Ins_V$, then
  $\sigma^*(w) = \{uav \mid \exists u, v \in V^* \ (w = uv)\}, \ \sigma^r(w) = \{wa\}, \ \sigma^l(w) = \{aw\}$.

We say that $\alpha \in \{*, l, r\}$ defines the way of applying a deletion or insertion rule to a word, namely at any position ($\alpha = *$), in the left ($\alpha = l$), or in the right ($\alpha = r$) end of the word, respectively. For a rule σ, an action $\alpha \in \{*, l, r\}$, and a language $L \subseteq V^*$, we define the *α-action of σ on L* by $\sigma^\alpha(L) = \bigcup_{w \in L} \sigma^\alpha(w)$. Given a finite set of rules M, we define the *α-action of M* on the word w and the language L by:
$M^\alpha(w) = \bigcup_{\sigma \in M} \sigma^\alpha(w)$ and $M^\alpha(L) = \bigcup_{w \in L} M^\alpha(w)$, respectively.

For two disjoint subsets P and F of an alphabet V and a word w over V, we define the predicates:
$$\varphi^{(s)}(w; P, F) \equiv \quad P \subseteq alph(w) \quad \wedge \quad F \cap alph(w) = \emptyset$$
$$\varphi^{(w)}(w; P, F) \equiv \quad (P = \emptyset \vee alph(w) \cap P \neq \emptyset) \quad \wedge \quad F \cap alph(w) = \emptyset.$$

The construction of these predicates is based on *random-context conditions* defined by the two sets P (*permitting contexts/symbols*) and F (*forbidding contexts/symbols*). Informally, the first condition requires that all permitting symbols are present in w and no forbidding symbol is present in w, while the second one is a weaker variant of the first, requiring that at least one permitting symbol (whenever the set of such symbols is not empty) appears in w and no forbidding symbol is present in w. Note that whenever one of the sets P or F is empty, we assume that it is undefined, so its effect on the final result of the predicates is null.

For every language $L \subseteq V^*$ and $\beta \in \{(s), (w)\}$, we define:
$$\varphi^\beta(L, P, F) = \{w \in L \mid \varphi^\beta(w; P, F)\}.$$

An *evolutionary processor over* V is a tuple (M, PI, FI, PO, FO), where:

- M is a set of substitution, or deletion, or insertion rules over the alphabet V, the set of rules of the processor. Formally ($M \subseteq Sub_V$) or ($M \subseteq Del_V$) or ($M \subseteq Ins_V$). Note that a processor is "specialized" in one type of operations only.

- $PI, FI \subseteq V$ are the *input* permitting, respectively forbidding, filters of the processor, while $PO, FO \subseteq V$ are the *output* permitting, respectively forbidding, filters of the processor. Informally, the permitting input (output) filters are the set of symbols that should be present in a word, when it enters (respectively, leaves) the processor, while the forbidding filters are the set of symbols that should not be present in a word in order to enter (respectively, leave) the processor. Note, once more, that whenever one of the sets PI, PO, FI, FO is empty, we say that it was left undefined, and, by the previous definitions, this set does not interfere with the filtering process.

We denote the set of evolutionary processors over V by EP_V.

Next we define the central notion of our paper, the Accepting Networks of Evolutionary Processors (ANEPs for short). Our definition is slightly different from the one that was used in literature so far (see, for instance, [15]), by the introduction and usage of a special accepting symbol μ.

The main reason for giving a different definition is that we are interested in using such networks as deciding devices (thus, devices that halt on every input), not only as accepting devices. To this end, our opinion is that the halting conditions assumed in the previous definitions of ANEPs were somehow artificial, and quite unrelated to all the other concepts defined for these devices. We will motivate more our definition later, and the rest of the paper will be focused on analysing the computational properties of the newly defined variant of ANEPs.

An *accepting hybrid network of evolutionary processors* (ANEP for short) is a 9-tuple $\Gamma = (V, U, \mu, G, \mathcal{N}, \alpha, \beta, x_I, x_O)$, where:

- V and U are the input and network alphabets, respectively, $V \subset U$; the symbol $\mu \in U \setminus V$ is a distinguished symbol, called accepting symbol.
- $G = (X_G, E_G)$ is a directed graph, with the set of nodes X_G and the set of edges $E_G \subseteq X_G \times X_G$. The graph G is called the *underlying graph* of the network, and card(X_G) is the size of Γ.
- $\mathcal{N} : X_G \longrightarrow EP_U$ is a mapping which associates with each node $x \in X_G$ the evolutionary processor $\mathcal{N}(x) = (M_x, PI_x, FI_x, PO_x, FO_x)$.
- $\alpha : X_G \longrightarrow \{*, l, r\}$; $\alpha(x)$ defines the action of the rules of node x when applied to the words existing in that node.
- $\beta : X_G \longrightarrow \{(s), (w)\}$ defines the type of the *input and output filters* of a node. More precisely, for every node $x \in X_G$ the following filters are defined:

$$\text{input filter: } \rho_x(\cdot) = \varphi^{\beta(x)}(\cdot; PI_x, FI_x),$$
$$\text{output filter: } \tau_x(\cdot) = \varphi^{\beta(x)}(\cdot; PO_x, FO_x).$$

That is, $\rho_x(w)$ (respectively, $\tau_x(w)$) indicates whether or not the word w can pass the input (respectively, output) filter of x.

- x_I and $x_O \in X_G$ are the *input node*, and, respectively, the *output node* of the network Γ.

An ANEP is said to be complete if the underlying graph G has the edges $E_G = \{(x, y) \mid x \neq y \text{ and } x, y \in X_G\}$ or, in other words, it is a complete undirected graph.

A *configuration* of an ANEP Γ is a mapping $C : X_G \longrightarrow 2^{V^*}$, associating a set of words with every node of the graph. A configuration may be understood as the sets of words which are present in any node at a given moment; it can change either by an *evolutionary step* or by a *communication step.*

When changing by an evolutionary step each component $C(x)$ of the configuration C is changed in accordance to the set of evolutionary rules M_x, of node x, and $\alpha(x)$, the way these rules should be applied. Formally, the configuration C' is obtained in *one evolutionary step* from the configuration C, written as $C \Longrightarrow C'$, if and only if

$$C'(x) = M_x^{\alpha(x)}(C(x)), \text{ for all } x \in X_G.$$

When changing by a communication step, each node-processor $x \in X_G$ sends one copy of each word it contains, and is able to pass its output filter, to all the node-processors connected to x, and receives all the words sent by all the other node processor connected with x, provided that they can pass its input filter. Formally, the configuration C' is obtained in *one communication step* from configuration C, written as $C \vdash C'$, if and only we have the equality

$$C'(x) = (C(x) - \tau_x(C(x))) \cup \bigcup_{y \in V_G, (y,x) \in E_G} \rho_x(\tau_y(C(y))), \text{ for all } x \in X_G.$$

Note that the words which leave a node are eliminated from that node; if such a word cannot pass the input filter of any node, it is lost.

The computation of Γ on the input word $w \in V^*$ is a (potential infinite) sequence of configurations $C_0^w, C_1^w, C_2^w, \ldots$. The initial configuration C_0^w is defined by $C_0^w(x_I) = \{w\}$ and $C_0^w(x) = \emptyset$ for all $x \in X_G$, $x \neq x_I$. Further, $C_{2i}^w \Longrightarrow C_{2i+1}^w$ and $C_{2i+1}^w \vdash C_{2i+2}^w$, for all $i \geq 0$.

The acceptance symbol μ is important when defining the accepting and rejecting computations of an ANEP.

To begin with, we define what a *halting computation* of the ANEP means. A computation $C_0^w, C_1^w, C_2^w, \ldots$ is said to be a *halting computation* (or a finite computation) if there exists $t \geq 0$ such that $C_t^w(x_O) \neq \emptyset$ and $C_\ell^w(x_O) = \emptyset$ for all $\ell < t$. We say that Γ halts on the input word w in t computational steps.

We distinguish two situations, in the case of such a halting computation:

i. There exists $u \in C_t^w(x_O)$, such that u contains the symbol μ. In this case, the computation is an *accepting computation*, and Γ accepts w.

ii. Any word $u \in C_t^w(x_O)$ does not contain the symbol μ. In this case, the computation is a *rejecting computation*, and Γ rejects w.

The *language accepted* by Γ is $L_a(\Gamma) = \{w \in V^* \mid$ the computation of Γ on w is an accepting one$\}$.

We say that an ANEP Γ decides the language $L \subseteq V^*$, and write $L(\Gamma) = L$ if and only if $L_a(\Gamma) = L$ and Γ halts on all input words.

Let Γ be an ANEP deciding the language L. The *time complexity* of the finite computation C_0^w, C_1^w, C_2^w, $\ldots C_m^w$ of Γ on $w \in L$ is denoted by $Time_\Gamma(w)$ and equals m. The time complexity of Γ is the function $Time_\Gamma(n) = \max\{Time_\Gamma(x) \mid |x| = n\}$. We say that Γ works in polynomial time, if there exists a polynomial function f such that $f(n) \geq Time_\Gamma(n)$.

For a function $f : \mathbf{N} \longrightarrow \mathbf{N}$ we define $\mathbf{Time}_{ANEP}(f(n)) = \{L \mid$ there exists the ANEP Γ, such that $L(\Gamma) = L, Time_\Gamma(n) \leq f(n)$, for all $n \in I\!N\}$. Moreover, we write $\mathbf{PTime}_{ANEP} = \bigcup_{k\geq 0} \mathbf{Time}_{ANEP}(n^k)$.

One can define in a similar manner space and length complexity classes (see the definitions given in the survey [15]).

3. The new halting condition and computability results

The single difference between the form of the initial definition of ANEPs ([16]) and form of ours is the presence and usage of the symbol μ. The way a computation of an ANEP is conducted remains basically the same, but the halting, accepting, and rejecting conditions are essentially different.

But let us first recall the definition of a halting ANEP-computation from the literature (see the seminal work [16], the survey [15], and the references therein). A computation halts and accepts if there exists a configuration in which the set of words existing in the output node is non-empty. A computation halts and rejects if there exist two identical configurations obtained either in consecutive evolutionary steps or in consecutive communication steps. The language accepted by the ANEP Γ is $L_a(\Gamma) = \{w \in V^* \mid \text{the computation of } \Gamma \text{ on } w \text{ accepts}\}$. Also, it was said that an ANEP Γ decides the language $L \subseteq V^*$ iff $L_a(\Gamma) = L$ and L halts on every input.

The main reason that made us consider a new definition is that we strongly believe that one should first define the notion of halting computation, and only then highlight the difference between accepting and rejecting computations. Other reasons that let to the switch to a new definition are related mainly to the previous definition of the rejecting computations.

First, checking whether the rejecting condition was fulfilled did not seem coherent with the other verifications that were performed in an ANEP. For instance, the filters check the existence or absence of several symbols in the communicated words; the application of a rule by a processor consists (in the more complicated cases of substitution and deletion rules) in looking for the occurrences of a symbol in the words of that node (if it contains any), and replacing an arbitrary such occurrence with a symbol or with λ. On the other hand, verifying whether the rejecting condition was fulfilled was a very different process: one checked whether the configurations of all the nodes, in two consecutive steps of the same kind, were equal.

Also, the processes executed by an ANEP are localized: filters are associated with nodes, rules are associated with nodes, and the accepting condition concerned only one node, the output node. In the case of a rejecting computation the definition took us to a global level: we looked at all the words present at a given moment in the network. The condition seemed an artificial formalization of the case when the network enters in an infinite loop, and the computation should halt. However, only infinite loops in which the configurations are repeated in consecutive steps were detected. Although avoiding infinite loops seems to us a good-practice in programming (regardless of the computational model), ruling out the occurrence of such a situation by definition does not seem justified to us.

Nevertheless, verifying the equality between two configurations required to memorize, at any moment, all the words from the last two configurations. Thus, an additional memory-device was needed, and this was not (explicitly) part of an ANEP. This affected the self-containment of the definition.

The new halting and deciding conditions seem to overcome these problems. The computation halts as soon as a word enters in a special node.

Although it seems to be also a condition of a different nature from the ones that are checked in an ANEP, one may note that, in fact, it is natural to think that before each processing step a processor checks whether it contains some words, and then it looks for the places where the rules can be applied. Further, the decision of a computation is taken according to a test in which we check for the existence of a symbol in the words of a node; this seems coherent with the rest of the definition of a ANEP. Moreover, we do not need any auxiliary devices (as it was the memory we needed in the former case). Finally, but of great importance, as we have already mentioned, it seems natural to us to distinguish between the conditions that should be fulfilled by a computation in order to halt and the conditions that should be fulfilled by a halting computation in order to be accepting or rejecting. Indeed, we can only talk about accepting or rejecting an input word as long as our device has a finite computation on that word; therefore, we first see whether the computation halted, by checking the emptiness of the configuration of a special node, and, then, we look at the strings contained in that node and take the right decision.

It only remains to be settled in which measure the results already reported for ANEPs ([15]) still hold, with respect to the new definition.

Accepted languages. All the ANEP constructions proposed in the literature (for instance, in [13, 10, 1]), where one was interested only in accepting a language by complete ANEPs, can be still be used. However, we must modify such an ANEP in order to work properly in the new setting: the former output node becomes an insertion node where the symbol μ is inserted in the words that were accepted inside, and then we add a new output node, in which all the words containing μ are allowed to enter; the network can still be complete, by adding μ to the forbidding input filters of all the nodes, except for the new output node. Thus, one can construct, for a recursively enumerable language, an ANEP accepting it, w.r.t. the new definition.

Decided languages. In [13] one shows that the class of languages decided by an ANEP, with respect to the classical halting condition, is the class of recursive languages. The proof was based on simulating, in parallel, all the possible computations of a nondeterministic Turing machine; the words communicated in the network were encodings of the Turing machine configurations. As we have already mentioned, these proofs can be used, as long as we are not interested in deciding the language, but only in accepting it. However, any recursive language can be decided by a deterministic Turing machine that for each input either enters a single final state and accepts, or enters a single blocking state and rejects. The ANEP simulating a Turing machine, presented in [13], can be easily modified to decide a recursive language L: we simulate the deterministic Turing machine deciding L and allow in the former output node all the words that contain the final state or the blocking state; if a word containing the final state enters the output node, then the network accepts, otherwise, it rejects. In conclusion, the languages decided by ANEPs, w.r.t. the new definition, are the recursive languages.

Computational Complexity. The results regarding polynomial space complexity or polynomial deterministic time complexity reported in [13] can be also proved by simulating deterministic Turing machines by ANEPs and vice versa. Therefore, they remain valid when the new acceptance/rejection conditions are used. However, the results that were based on time-efficient simulations of nondeterministic machines are not preserved, as we will see in Section 5.

4. Complete ANEPs

It is worth mentioning that most of the computability and computational complexity results reported so far in literature deal with complete ANEPs or with ANEPs with a restricted topology (see, e.g., [16, 13, 14, 5], or the survey [15]). More precisely, there are many results stating that particular types of networks are accepting all the recursively enumerable languages or perform a series of tasks efficiently (e.g. solving NP-complete problems, or simulating efficiently different types of universal devices).

A natural question arises: in which measure is the topology of the networks important, with respect to the computational power or the efficiency of the computations? Such a result would be interesting if we consider that sometimes it is easier to construct an ANEP with a particular topology, solving a given problem efficiently, than to construct a complete one (thus, it is simpler to solve a problem by a non-uniform approach than by an uniform one). For instance, in [13, 10, 1] several involved technical tricks were used in order to show all the results for complete ANEPs, while in [2] it was left as an open problem to see if the reported results still hold for complete ANEPs.

A first answer is immediate, but unsatisfactory. Complete ANEPs can simulate (with respect to the former halting conditions) nondeterministic Turing machines, and nondeterministic Turing machines can simulate ANEPs of any kind (see [13]). So one can construct a complete ANEP simulating an arbitrary ANEP via the simulation by Turing machines. However, such a simulation is not time-efficient since the Turing machine simulates a computation of t steps of the ANEP on an input word of length n in time $\mathcal{O}(\max(t^2, tn))$; this approach is also complicated due to the construction of the intermediate Turing machine. Also, such an approach could lead to a complete ANEP that solve quite inefficiently a given problem, compared to the best ANEP-based solution for that problem (see, for instance, Example 1).

In the following we propose a new and precise answer to the above question: we can accept (respectively, decide) with a complete ANEP any language accepted (respectively, decided) by an ANEP with an arbitrary underlying graph, within the same computing time. Following the explanations from the previous section, the new halting and deciding conditions are not relevant in the case when we are interested only in accepting languages: it makes no difference whether we use the former conditions or we use these new conditions, we still obtain that given an arbitrary ANEP one can construct a complete ANEP accepting, as efficiently as the arbitrary ANEP, the same language. In fact, these new conditions come into play in the case when we are interested in deciding languages. Basically, the proof of our result consists in simulating an ANEP by a complete ANEP. Two consecutive steps of the initial ANEP are simulated in exactly 54 consecutive steps of the complete ANEP. In the classical setting, the initial ANEP rejected when the configurations obtained in two consecutive steps of the same kind were identical; but, in our simulation, these configurations do not occur in the complete ANEP in consecutive evolutionary or communication steps so, if the former deciding conditions would be used, the new network would not reject, but enter in an infinite cycle. Thus, such a simulation would not preserve the halting property of a computation. However, when the new conditions are used the halting property is preserved canonically.

The proof of the announced results is based on the following two Lemmas.

Lemma 1. *Given an ANEP $\Gamma = (V, U, \mu, G, \mathcal{N}, \alpha, \beta, In, Out)$, one can construct an ANEP $\Gamma' = (V, U, \mu, G', \mathcal{N}', \alpha', \beta', In', Out')$ such that Γ' accepts (decides) the same language as Γ accepts (respectively, decides), each node of Γ' has at most one rule and In' has no rules. Moreover, two consecutive steps of Γ (an evolutionary and a communication step) are simulated in exactly 6 consecutive steps (3 evolutionary and 3 communication steps) of Γ'.*

Proof. We will show how we can construct for each node x of the network

Γ a subnetwork $s(x)$ contained in Γ', whose processors simulate the computation of the processor $\mathcal{N}(x)$. We denote by $\mathrm{set}(s(x))$ the nodes of the subnetwork $s(x)$, and distinguish two nodes of this subnetwork, namely $i(s(x))$ and $o(s(x))$, that make possible the communication with the subnetworks constructed for the other nodes (these two nodes can be seen as the input node, and, respectively, the output node of the subnetwork, while all the other nodes can be seen as internal nodes). We also denote by $\mathrm{edge}(s(x))$ the set of edges of the subnetwork (that is, the edges connecting the nodes of the subnetwork between them).

Let us assume that the node x verifies $\mathcal{N}(x) = (M, PI, FI, PO, FO)$, $M = \{r_1, \ldots, r_n\}$, and $\alpha(x) = (s)$. Then we have:

- $\mathrm{set}(s(x)) = \{x_0, x_0', x_1, x_2\} \cup \{x_a \mid a \in PO\} \cup \{x_1^r, \ldots, x_n^r\}$, $i(s(x)) = x_0$, $o(s(x)) = x_2$.
- $\mathrm{edge}(s(x)) = \{(x_a, x_0'), (x_i^r, x_a) \mid a \in PO, i \in \{1, \ldots, n\}\} \cup \{(x_0, x_i^r),$ (x_i^r, x_1), (x_i^r, x_2), (x_1, x_0'), $(x_0', x_i^r) \mid i \in \{1, \ldots, n\}\}$.
- $\mathcal{N}'(x_0) = (\emptyset, PI, FI, \emptyset, \emptyset)$, $\alpha'(x_1) = (s)$, $\beta'(x_1) = *$.
- $\mathcal{N}'(x_1) = (\emptyset, FO, \emptyset, U, \emptyset)$, $\alpha'(x_1) = (w)$, $\beta'(x_1) = *$.
- For $a \in PO$ we have: $\mathcal{N}'(x_a) = (\emptyset, W, \{a\}, U, \emptyset)$, $\alpha'(x_a) = (w)$, $\beta'(x_a) = *$.
- For $i \in \{1, \ldots, n\}$ we have: $\mathcal{N}'(x_i^r) = (\{r_i\}, U, \emptyset, U, \emptyset)$, $\alpha'(x_i^r) = (w)$, $\beta'(x_i^r) = *$.
- $\mathcal{N}'(x_0') = (\emptyset, U, \emptyset, U, \emptyset)$, $\alpha'(x_0') = (w)$, $\beta'(x_0') = *$.
- $\mathcal{N}'(x_2) = (\emptyset, PO, FO, \emptyset, \emptyset)$, $\alpha'(x_2) = (s)$, $\beta'(x_2) = *$.

To see that the subnetwork $s(x)$ defined above simulates the behaviour of the node x, let us assume that w is a word that was sent towards the node x in a communication step of Γ, and the same word was also sent towards the node x_0 in Γ'. It is immediate that the word w can enter (that is, fulfils the conditions requested by the input filter of) node x if and only if it can enter node x_0. In node x, a rule r_i is applied to the word w; this is simulated in $s(x)$ in the following steps: the word w goes from node x_0 to node x_i^r, where the rule r_i is applied to it. Back in Γ, the word can exit the node x, if it can pass the filters PO and FO, or remain in the node and be further processed. In Γ' the word can go to node x_2, if it verifies the filters PO and FO, and leave the subnetwork in the next communication step; if it does not verify these filters, it goes to node x_1 (if it has a forbidden symbol in it) or to a node x_a (if it does not contain the symbol $a \in PO$), and from these nodes is sent to x'_0, and then it is resent to the nodes x^r_j, for $j \in \{1, \ldots, n\}$, to be further processed. From these explanations, it follows that the subnetwork $s(x)$ behaves exactly like the node x. Moreover, one processing step and one communication step of Γ are simulated in Γ' by 3 processing and 3 communication steps.

The case of the nodes with weak filters is easier. Let us assume that the node x verifies $\mathcal{N}(x) = (M, PI, FI, PO, FO)$, $M = \{r_1, \ldots, r_n\}$, and $\alpha(x) = (w)$. Then we have:

- $\mathrm{set}(s(x)) = \{x_0, x_1, x'_1, x_2\} \cup \{x^r_1, \ldots, x^r_n\}$, $i(s(x)) = x_0$, $o(s(x)) = x_2$.
- $\mathrm{edge}(s(x)) = \{(x_0, x^r_i), (x^r_i, x_1), (x^r_i, x_2), (x'_0, x^r_i), (x^r_i, x'_1) \mid 1 \le i \le n\} \cup$ $\cup \{(x_1, x'_0), (x'_1, x'_0)\}$.
- $\mathcal{N}'(x_0) = (\emptyset, PI, FI, U, \emptyset)$, $\alpha'(x_0) = (w)$, $\beta'(x_0) = *$.
- $\mathcal{N}'(x_1) = (\emptyset, FO, \emptyset, U, \emptyset)$, $\alpha'(x_1) = (w)$, $\beta'(x_1) = *$.
- $\mathcal{N}'(x'_1) = (\emptyset, W, PO, U, \emptyset)$, $\alpha'(x_a) = (w)$, $\beta'(x_a) = *$.
- For $i \in \{1, \ldots, n\}$ we have: $\mathcal{N}'(x^r_i) = (\{r_i\}, U, \emptyset, U, \emptyset)$, $\alpha'(x^r_i) = (w)$, $\beta'(x^r_i) = *$.
- $\mathcal{N}'(x'_0) = (\emptyset, U, \emptyset, U, \emptyset)$, $\alpha'(x'_0) = (w)$, $\beta'(x'_0) = *$.
- $\mathcal{N}'(x_2) = (\emptyset, PO, FO, U, \emptyset)$, $\alpha'(x_2) = (w)$, $\beta'(x_2) = *$.

The simulation of the computation of node x by the subnetwork $s(x)$ goes similarly to the above. The only difference is that in the case of weak filters, a word cannot exit, and remains blocked, in the node x only in the case when it contains a forbidden symbol (in the simulation, the word goes to x_1) or in the case it does not contain any of the permitting symbols (in this case, during the simulation, the word goes to x'_1). As in the former case, one processing step and one communication step of Γ are simulated in Γ' by 3 processing and 3 communication steps.

If a node x has no rules, our intuition is that it should be kept in the exact same form in the new network. However, in order to be sure that the ANEP Γ' we construct simulates correctly Γ, we must be sure that each step of the initial network is simulated in the same number of steps by the new one. For this, we must be sure that an evolutionary step of the initial network, in which no rule are applied, and the consequent communication step are simulated in 3 processing (in which no rule will be applied) and 3 communication steps of the new network. Therefore, in the case of $\alpha(x) = (s)$, we set:

- $\text{set}(s(x)) = \{x_0, x_1, x_2\}$, $i(s(x)) = x_0$, $o(s(x)) = x_2$.
- $\text{edge}(s(x)) = \{(x_0, x_1), (x_1, x_2)\}$.
- $\mathcal{N}'(x_0) = (\emptyset, PI, FI, U, \emptyset)$, $\alpha'(x_0) = (s)$, $\beta'(x_0) = *$.
- $\mathcal{N}'(x_1) = (\emptyset, PO, FO, U, \emptyset)$, $\alpha'(x_1) = (s)$, $\beta'(x_1) = *$.
- $\mathcal{N}'(x_2) = (\emptyset, U, \emptyset, U, \emptyset)$, $\alpha'(x_2) = (w)$, $\beta'(x_2) = *$.

The case of weak filters can be treated in the same manner: we just have to change the permitting output filters of x_0 and x_1, and make them equal to U, and set $\alpha(x_0) = \alpha(x_1) = (s)$.

To finish the construction of the network Γ' we set:

- $G' = (X_{G'}, E_{G'})$, where the set of nodes is $X_{G'} = \bigcup_{x \in X_G} \text{set}(s(x))$ and the set of edges is
$$E_{G'} = \{(o(s(x)), i(s(y))) \mid y \in V_G \text{ with } (x, y) \in E_G\} \cup \bigcup_{x \in X_G} \text{edge}(s(x)).$$
- The input node of the network, denoted In', is $i(s(x_I))$. The output node of the network, denoted Out', is $i(s(x_O))$.

From the remarks made when we explained our simulations it is clear that the network Γ' accepts (decides) exactly the same language as Γ. □

Lemma 2. *Given an ANEP $\Gamma = (V, U, \mu, G, \mathcal{N}, \alpha, \beta, In, Out)$, such that all the processors Γ have at most one rule and In has no rules, one can construct a complete ANEP $\Gamma' = (V, U', \mu, G', \mathcal{N}', \alpha', \beta', In'_0, Out')$ such that Γ' accepts (decides) the same language as Γ accepts (respectively, decides). Moreover, two consecutive steps of Γ (an evolutionary and a communication step) are simulated in exactly 18 consecutive steps (9 evolutionary and 9 communication steps) of Γ'.*

Proof. Let $U_1 = \{\#_x, \#'_x, \#''_x, \#^b_x, \#_{x,y} \mid b \in U, x, y \in X_G, (x,y) \in E_G\}$.

The complete network Γ' simulates the computation of Γ using the following strategy. We construct for each node x of Γ a subnetwork $s(x)$ of Γ' whose processors simulate the computation of the processor $\mathcal{N}(x)$; we denote by $\mathrm{set}(s(x))$ the nodes of the subnetwork $s(x)$. The underlying graph of Γ' is complete and has the nodes $\bigcup_{x \in X_G} \mathrm{set}(s(x))$. All the words processed in the new network have a special symbol from U_1. The symbols of U_1 that encode one processor of the initial network indicate the nodes whose actions must be simulated at that point, thus which of the subnetworks should act on the word. The symbols that encode two nodes indicate a possible way to communicate the word containing it between the subnetworks of Γ'. The symbol $\#_{In}$ is inserted in the input word at the beginning of the computation, so we should start by simulating the input node of Γ. Further, the way the computation is conducted, described below, and the way symbols of U_1 are inserted, deleted or modified, in the processed words enable us to simulate, in parallel, all the possible derivations of the input word in Γ, and ensure that the subnetworks act independently (that is, that they can actually interact only after they simulate completely one evolutionary step of the corresponding nodes of Γ, in order to simulate the communication step of the original network).

The alphabet of Γ' is defined as

$$U' = U \cup \{b', b'' \mid b \in U\} \cup \{\#^{(i)} \mid 1 \leq i \leq 8\} \cup U_1.$$

In the following, we define the rest of the network.

We will split our discussion in many cases, according to the type of the node (with no rules, insertion, substitution, deletion), the way the operations are applied (left, right, arbitrary) and the type of the filters. We stress out from the beginning that one processing step and the subsequent communication step of the network Γ will be simulated in exactly 9 processing steps and the corresponding communication steps in Γ'; this is because in the most intricate case, namely the case of deletion nodes, our network actually needs this many steps to simulate correctly one step of Γ.

Assume that the node x verifies $\mathcal{N}(x) = (\emptyset, PI, FI, PO, FO)$. Then we have:

- $\mathrm{set}(s(x)) = \{x_0, x_1\}$,
- $\mathcal{N}(x_0) = (\{\#_x \to \#'_x\}, PI, FI \cup (U_1 \setminus \{\#_x\}), \{\#'_x\}, \emptyset)$, $\alpha'(x_0) = \alpha(x)$.
- $\mathcal{N}(x_1) = (\{\#'_x \to \#^{(1)}, \#^{(7)} \to \#_y \mid (x,y) \in E_G\} \cup \{\#^{(i)} \to \#^{(i+1)} \mid$

$1 \leq \ i \leq \ 6\}, U', \emptyset, PO, FO \cup \{\#'_x\} \cup \{\#^{(i)} \mid 1 \leq i \leq 7\})$, $\alpha'(x_1) = \alpha(x)$.

In $s(x)$ we only change the symbol $\#_x$ into a new symbol $\#_y$, indicating that the word can now go towards the nodes of the subnetwork $s(y)$, and cannot enter the nodes of any other subnetwork. The only trick is that we must do this change in nine steps, instead of a single rewriting step. The rest of the word is left unchanged, as it was also the case in the node x of the initial network, where the whole word stayed unchanged.

In the case of the input node In of Γ, the only difference is that we add a new node In'_0, which is an insertion node, where $\#'_{In}$ is inserted. The input node of the network is In'_0. This node does not allow any word to enter, and allows all the words to exit, so it only acts at the beginning of the computation. The subnetwork associated with the output node Out has only the node Out_0, which is the output node of the new network.

For the node x, with $\mathcal{N}(x) = (\{\lambda \to a\}, PI, FI, PO, FO)$, $\alpha(x) = (s)$ and $\beta(x) = l$, we have:

- $\text{set}(s(x)) = \{x_0, x_1, x_2, x_3\}$.
- $\mathcal{N}(x_0) = (\{\lambda \to \#'_x\}, PI, FI \cup (U_1 \setminus \{\#_x\}), U', \emptyset)$, $\alpha'(x_0) = (s)$, $\beta'(x_0) = r$.
- $\mathcal{N}(x_1) = (\{\#_x \to \lambda\}, \{\#'_x, \#_x\}, U_1 \setminus \{\#'_x, \#_x\}, U', \emptyset)$, $\alpha'(x_1) = (s)$ and $\beta'(x_1) = \ *$.
- $\mathcal{N}(x_2) = (\{\lambda \to a'\}, \{\#'_x\}, U_1 \setminus \{\#'_x\}, (PO \cup \{a'\}) \setminus \{a\}, FO \cup \{b' \mid b \in FO\})$, $\alpha'(x_2) = (s)$, $\beta'(x_2) = l$.
- $\mathcal{N}(x_3) = (\{a' \to a\} \cup \{\#'_x \to \#^{(1)}, \#^{(1)} \to \#^{(2)}, \#^{(2)} \to \#^{(3)}, \#^{(3)} \to \#^{(4)}, \#^{(4)} \to \#_y \mid (x, y) \in E_G\}, \{a'\}, U_1 \setminus \{\#'_x\}, \{\#_y \mid (x, y) \in E_G\}, \{a', \#'_x\})$, $\alpha'(x_3) = (w)$, $\beta'(x_3) = *$.

The subnetwork associated with a right insertion node, with strong filters, is very similar, the only different things being that $\beta'(x_0) = l$ and $\beta'(x_2) = r$. Also, the subnetwork associated with an arbitrary insertion node, with strong filters, is similar, this time the different things being that $\beta'(x_0) = \beta'(x_2) = *$.

In the following we explain how the subnetwork works in the case of left insertion nodes (as the other cases are treated analogously). Let us assume that w is a word that was sent towards the node x in a communication step of Γ, and the word $w_1\#_x w_2$, with $w_1 w_2 = w$, was communicated by some node in the network Γ'. If the word w can pass the input filters of x then $w_1\#_x w_2$

can also enter x_0 (and no other node, in fact); the reversal holds as well. In the node x of Γ we obtain from w the word aw, which leaves the node if and only if it verifies the conditions imposed by the output filters; note that if the word does not leave the node after one processing step, then it will never leave, since any new insertion would not make it contain any new symbol (because, as we have assumed, we have at most one rule per node). In the network Γ' the word is processed as follows. In the node x_0 it becomes $w_1\#_x w_2\#'_x$ and is sent out. As we will see after the whole network is defined, it can only enter the node x_1 (because all the other nodes block the words containing both $\#_x$ and $\#'_x$); here it becomes $w_1w_2\#'_x = w\#'_x$. Further, it is communicated in the network and can only enter the node x_2, where it becomes $a'w\#'_x$. It is not hard to see that this word can leave the node x_2 if and only if aw can leave the node x, of Γ; as in the case of the initial network Γ, if the word does not leave the node after one processing step, then it will never do so. Therefore, if the word fulfils the conditions of the output filters and leaves the node it can only go to node x_3 where we obtain, in 6 processing steps, all the words $aw\#_y$, for a node y such that $(x,y) \in E_G$. Each word $aw\#_y$ leaves the node and can go to the nodes of the subnetworks associated with the node y. Clearly, $s(x)$ simulates correctly one processing step done by x, and the consequent communication step, in 9 processing and communication steps.

The case of insertion nodes with weak filters is very similar. We just have to redefine the filters of nodes x_0 and x_2 to be weak, and to set the permitting output filter of x_0 to be equal to U'.

Now we move on to substitution nodes. In this case we only have two cases, according to the way filters are used, since substitutions are always applied in the $*$ mode (i.e., to an arbitrary position). For the node x, with $\mathcal{N}(x) = (\{a \to b\}, PI, FI, PO, FO)$, $a \in FO$, $\alpha(x) = (s)$ and $\beta(x) = *$, we have:

- $\mathrm{set}(s(x)) = \{x_0, x_1, x_2, x_3, x_4, x_5, x_6\}$.
- $\mathcal{N}(x_0) = (\{\#_x \to \#'_x\}, PI, FI \cup (U_1 \setminus \{\#_x\}), U', \emptyset)$, $\alpha'(x_0) = (s)$, $\beta'(x_0) = *$.
- $\mathcal{N}(x_1) = (\{a \to b'\}, \{a\}, U_1 \setminus \{\#'_x\}, U', \emptyset)$, $\alpha'(x_1) = (s)$, $\beta'(x_1) = *$.
- $\mathcal{N}(x_2) = (\{\#'_x \to \#''_x\}, \{\#'_x\}, \{a\} \cup (U_1 \setminus \{\#'_x\}), U', \emptyset)$, $\alpha'(x_2) = (s)$, $\beta'(x_3) = *$.

- $\mathcal{N}(x_3) = (\{\#'_x \to \#^\circ_x\}, \{b'\}, U_1 \setminus \{\#'_x\}, \{a\}, FO \cup \{c' \mid c \in FO\})$, $\alpha'(x_3) = (s)$, $\beta'(x_3) = *$.

- $\mathcal{N}(x_4) = (\{b' \to b,\ \#^\circ_x \to \#^{(1)},\ \#^{(1)} \to \#^{(2)},\ \#^{(2)} \to \#^{(3)},\ \#^{(3)} \to \#^{(4)},\ \#^{(4)} \to \#'_x\}, \{b'\}, U_1 \setminus \{\#^\circ_x\}, \{a\}, FO \cup \{b', \#^\circ_x\})$, $\alpha'(x_4) = (w)$, $\beta'(x_4) = *$.

- $\mathcal{N}(x_5) = (\{b' \to b \mid b \in U\} \cup \{\#'_x \to \#^{(2)}, \#''_x \to \#^{(1)}, \#^{(1)} \to \#^{(2)}, \#^{(2)} \to \#^{(3)}, \#^{(3)} \to \#^{(4)}, \#^{(4)} \to \#_{x,y} \mid y \in V_G \text{ with } (x,y) \in E_G\}, \{b', \#''_x\}, FO \cup \{c' \mid c \in FO\} \cup (U_1 \setminus \{\#'_x, \#''_x\}), U', \{\#^{(i)} \mid 1 \leq i \leq 4\} \cup \{\#''_x, b', \#'_x\})$, $\alpha'(x_5) = (w)$, $\beta'(x_5) = *$.

- $\mathcal{N}(x_6) = (\{\#_{x,y} \to \#_y \mid y \in X_G\}, PO, U_1 \setminus \{\#_{x,y} \mid y \in V_G \text{ with } (x,y) \in E_G\}, U', \{\#_{x,y} \mid (x,y) \in E_G\})$, $\alpha'(x_6) = (s)$, $\beta'(x_6) = *$.

If $a \notin FO$ then we simply delete the nodes x_3 and x_4.

The simulation implemented by the subnetwork is, in this case, more involved. Let us assume that w is a word that was sent towards the node x in a communication step of Γ, and the word $w_1\#_x w_2$, with $w_1 w_2 = w$, was communicated in the network Γ'. If the word w can pass the input filters of x then $w_1\#_x w_2$ can also enter x_0 (and no other node); the reversal holds as well. In the node x we obtain from w a word w' by substituting several symbols a with symbols b (actually, we either substitute exactly one occurrence of an a, if $a \notin FO$, all of them, if $a \in FO$, or none, if w contains no a) and w' leaves the node if it fulfils the output conditions. In the network Γ' the word is processed as follows. In x_0 it becomes $w_1\#'_x w_2$ and is sent out. It can only enter x_1, if it contains at least one a, or x_2, otherwise. In the first case it becomes $w'_1\#'_x w'_2$, by substituting exactly one symbol a with a symbol b'. In the second case, the word becomes $w_1\#''_x w_2$ (only the symbol encoding the current node is changed, and the rest of the word remains the same because it contains no a). In the both cases, the obtained words enter node x_5 where we obtain, in 5 processing steps, all the words $w_3\#_{x,y}w_4$, for a node y such that we have $(x,y) \in E_G$ and $w_3 w_4 = w'$, where w' was obtained from w by substituting at most one symbol a with a symbol b. Such words can only be communicated to node x_6, if they fulfil the conditions of the output filters of x. In x_6 they are transformed into words that have the form $w_3\#_y w_4$, and can go to the nodes of the subnetwork associated with the node y of the network Γ. There is one more thing to be analysed: the words that leave x_2 and contain a, in the case when $a \in FO$. These words can go to node x_3, where they become $w'_1\#^\circ_x w'_2$ and, further, can only enter node x_4 (but only if they do not contain any more forbidden output symbols of the node x or their primed copies). In this node, b' becomes b and $\#_x^\circ$ becomes $\#_x'$, after 6 processing steps are performed, and the obtained words are sent back to node x_2, where another processing step of the original node x is simulated. It is not hard to see, from these explanations, that the action of the node x in Γ was correctly simulated by the subnetwork $s(x)$ in Γ'; more precisely one processing and one communication step of Γ are simulated in 9 processing and 9 communication steps of Γ'.

The case of substitution nodes with weak filters can be treated in a similar fashion. We simply have to redefine the filters of x_0 and x_6 in the previous network as being weak, and to set their permitting output filters to be equal to U'. As in the former case, if w is a word that was sent towards the node x in a communication step of Γ and the word $w_1\#_x w_2$, with $w_1w_2 = w$, was communicated by some node in the network Γ' we obtain in the latter network all the words $w_3\#_y w_4$, such that $(x, y) \in E_G$ and $w_3w_4 = w'$ where w' was obtained in node x from w and was allowed to leave that node. In this manner, the action of the node x in Γ was simulated soundly by the subnetwork $s(x)$ in Γ'; once more, one processing and one communication step of Γ are simulated in 9 processing and 9 communication steps of Γ'.

Finally, we present the simulation of the deletion nodes. As in the case of insertions, we can have left, right or arbitrary deletion rules, and strong or weak filters. However, all the cases are based on the same general idea. Therefore, we present and discuss in details only one case: the case of left deletion nodes, with strong filters. For right and arbitrary insertions we will describe only the differences from this construction.

Let us assume that the node x has $\mathcal{N}(x) = (\{a \to \lambda\}, PI, FI, PO, FO)$, $\alpha(x) = (s)$ and $\beta(x) = l$. First, we will assume that $a \in FO$. In this case, we have:

- $\text{set}(s(x)) = \{x_0, x_1, x_2, x_3, x_4, x_5, x_6\} \cup \{x_b, x^b \mid b \in U\}$.
- $\mathcal{N}(x_0) = (\{\lambda \to \#_x'\}, PI, FI \cup (U_1 \setminus \{\#_x\}), U', \emptyset)$, $\alpha'(x_0) = (s)$, $\beta'(x_0) = r$.
- $\mathcal{N}(x_1) = (\{\#_x \to \lambda\}, \{\#_x', \#_x\}, (U_1 \setminus \{\#_x, \#_x'\}) \cup \{a'\}, U', \emptyset)$, $\alpha'(x_1) = (s)$, $\beta'(x_1) = *$.
- $\mathcal{N}(x_2) = (\{b \to b' \mid b \in U\}, \{\#_x'\}, U_1 \setminus \{\#_x'\}, \{b' \mid b \in U\}, \emptyset)$, $\alpha'(x_2) = (w)$, $\beta'(x_2) = *$.

- $\mathcal{N}(x_3) = (\{\#'_x \to \#^b_x \mid b \in V\}, \{b' \mid b \in V\}, U_1 \setminus \{\#'_x\}, U', \{\#'_x\})$, $\alpha'(x_3) = (w)$, $\beta'(x_3) = *$.
- For all $b \in U$ we have: $\mathcal{N}(x_b) = (\{b' \to \lambda\}, \{\#^b_x\}, \{c' \mid c \in U \setminus \{b\}\} \cup \{c'' \mid c \in U\}, U', \{c' \mid c \in U\})$, $\alpha'(x_b) = (w)$, $\beta'(x_b) = l$.
- For all $b \in U \setminus \{a\}$ we have: $\mathcal{N}(x^b) = (\{\lambda \to b''\}, \{\#^b_x\}, \{c' \mid c \in U\}, U', \emptyset)$, $\alpha'(x^b) = (w)$, $\beta'(x^b) = l$.
- $\mathcal{N}(x^a) = (\{\#^a_x \to \#''_x\}, \{\#^a_x\}, \{c' \mid c \in U\}, U', \emptyset)$, $\alpha'(x^a) = (w)$, $\beta'(x^a) = *$.
- $\mathcal{N}(x_4) = (\{\#''_x \to \#^{(1)}, \#^{(1)} \to \#^{(2)}, \#^{(2)} \to \#^{(3)}, \#^{(3)} \to \#^{(4)}, \#^{(4)} \to \#'_x\}, \{a\}, \{b'' \mid b \in U\} \cup (U_1 \setminus \{\#''_x\}), \{\#'_x\}, \emptyset)$, $\alpha'(x_4) = (w)$, $\beta'(x_4) = *$.
- $\mathcal{N}(x_5) = (\{\#''_x \to \#^{(1)}, \#^{(1)} \to \#_{x,y}, \#^b_x \to \#_{x,y}, b'' \to b \mid b \in U, (x,y) \in E_G\}, \{\#''_x\} \cup \{b'' \mid b \in U\}, FO \cup \{b'' \mid b \in FO\} \cup (U_1 \setminus \{\#''_x\}), U', \{\#^{(1)}\} \cup \{\#''_x\} \cup \{b'', \#^b_x \mid b \in U\})$, $\alpha'(x_5) = (w)$, $\beta'(x_5) = *$.
- $\mathcal{N}(x_6) = (\{\#_{x,y} \to \#_y \mid y \in X_G\}, PO, \{\#''_y, \#_y, \#'_y, \#^b_y \mid b \in U, y \in X_G\}, \emptyset, \{\#_{x,y} \mid (x,y) \in E_G\})$, $\alpha'(x_6) = (s)$, $\beta'(x_6) = *$.

Let us assume that w is a word that was sent towards the node x in a communication step of Γ and the word $w_1\#_x w_2$, with $w_1 w_2 = w$, was communicated by some node in the network Γ'. If the word w can pass the input filters of x then $w_1\#_x w_2$ can also enter x_0, and vice versa. In the node x of Γ we obtain from w a word w' by deleting all the symbols a from the left end of the word, and this word leaves the node if it verifies the output conditions. In the network Γ' the word is processed as follows. In the node x_0 it becomes $w_1\#_x w_2\#'_x$ and is sent out. It can only enter the node x_1 where it becomes $w\#'_x$. Now it can only go to node x_2. Here it is transformed into $w'_1 b' w'_2 \#'_x$ for all $b \in U$ and $w'_1, w'_2 \in U^*$ such that $w'_1 b w'_2 = w$. Now these words enter node x_3 and the network obtains from them the words $w'_1 b' w'_2 \#^c_x$ with $c \in U$. From these only the words $w'_1 b' w'_2 \#^b_x$ are further processed. More precisely, the node x_b permits these words to enter and transforms them into $w'_2\#^b_x$ if and only if $w'_1 = \lambda$. Next, the obtained words can only go to node x^b. If $b \neq a$ it means that we simulated a deletion that should not have happened, so we remake the word into $b'' w'_2 \#^b_x$; otherwise, the deletion was correct and we get in x^a the word $w'_2\#''_x$. In the first case, the words can enter node x_5 (if they do not contain any of the forbidden output symbols of node x) where they are transformed into $w'\#_{x,y}$, with $(x,y) \in E_G$, in two processing steps and sent out; finally, they can enter only node x_6 but if and only if w' verifies the permitting output filters of the original node x, and here the words are transformed into $w'\#_y$ and are sent to the other nodes of the network (and can only enter the nodes of the subnetwork constructed for the node y). In the second case, the word $w_2'\#_x''$ can either enter x_5, and be processed as above, or, if it still contains symbols a, which are forbidden output symbols for x, it goes to node x_4. In this node they are transformed into $w_2'\#_x'$ (in 5 steps), go back to node x_2, and the whole process described above is repeated. It is not hard to see now that the action of the node x in Γ was correctly simulated by the subnetwork $s(x)$ in Γ'. More precisely, one processing and one communication step of Γ are simulated, again, in exactly 9 processing and 9 communication steps of Γ'.

In the case when $a \notin FO$ we simply remove the node x_4 from the above construction, and the simulation goes on just the same. To adapt the construction to the case of right deletion nodes, we should switch the working mode of nodes x_0, x_b and x^b; that is, if one of these nodes was a node where left (right) operations were applied, it must become a node where right (respectively, left) operations are applied. In the case of arbitrary deletion nodes, we simply redefine all the node such that they apply the rules arbitrary. Finally, when the node x has weak filters, we set the filters of x_1 and x_6 to be weak and redefine their permitting output filters as being equal to U', such that all the words that do not contain any forbidden output symbols are allowed to leave these nodes.

The output node of the network is the node Out_0 from the subnetwork $s(Out)$, associated with the node Out.

From the way the subnetworks $s(x)$, with $x \in X_G$, work we see that the following statements are equivalent:

i. In Γ: w is a word that entered the node x, was transformed by this node into w' in one step, and w' was communicated to node y (when w' cannot exit x we assume that $y = x$);

ii. In Γ': the word $w_1\#_x w_2$, with $w_1 w_2 = w$, entered the node x_0, from the subnetwork $s(x)$, it was transformed, by the nodes of $s(x)$, into $w_1'\#_y w_2'$, with $w_1' w_2' = w'$, in exactly 9 evolutionary and 9 communication steps, and was communicated to the nodes of the network.

Note, also, that Γ' accepts a word w if and only if a word $w_1\#_{Out}w_2$, with $w_1w_2 \in U^*$, can be derived from it; but such a word can be derived only in a number of steps divisible by 9. According to the above, the computation of Γ on w ends (and accepts) in t steps if and only if the computation of Γ' on w ends (and, respectively, accepts) in $9t$ steps. Therefore, $L(\Gamma) = L(\Gamma')$. □

We can now derive the main result of this section.

Theorem 3. *Given an ANEP $\Gamma = (V, U, \mu, G, \mathcal{N}, \alpha, \beta, In, Out)$, one can construct a complete ANEP $\Gamma' = (V, U', \mu, G', \mathcal{N}', \alpha', \beta', In', Out')$ such that the network Γ' accepts (decides) the same language as Γ accepts (respectively, decides). Moreover, two consecutive steps of Γ (an evolutionary and a communication step) are simulated in exactly 54 consecutive steps (27 evolutionary and 27 communication steps) of Γ'.*

Proof. First, the given ANEP $\Gamma = (V,\ U,\ \mu,\ G,\ \mathcal{N},\ \alpha,\ \beta,\ x_I,\ x_O)$, is transformed into an ANEP $\Gamma' = (V,\ U,\ \mu,\ G',\ \mathcal{N}',\ \alpha',\ \beta',\ x_I',\ x_O')$ such that Γ' accepts (decides) the same language as Γ accepts (respectively, decides), each node of Γ' has at most one rule and x_I' has no rules. Moreover, two consecutive steps of Γ (an evolutionary and a communication step) are simulated in exactly in exactly 3 evolutionary steps and 3 communication steps of Γ'.

Second, by Lemma 2, we are able to transform the network Γ' into a complete ANEP $\Gamma'' = (V,\ U'',\ \mu,\ G'',\ \mathcal{N}'',\ \alpha'',\ \beta'',\ x_I'',\ x_O'')$ such that Γ' accepts (decides) the same language as Γ accepts (respectively, decides). Moreover, two consecutive steps of Γ (an evolutionary and a communication step) are simulated in exactly 9 evolutionary steps and 9 communication steps ofΓ'.

This shows that the statement of Theorem 3 hold. □

It is worth mentioning that the problem of simulating an arbitrary NEP by a complete NEP was approached also in the case of generating networks of evolutionary processors [6]. Note, though, that the things were quite different in that setting: the filters in generating networks were regular sets, and this allowed us to control easier the communication; no weak/strong filtering conditions existed; each node contains a set of axioms, at the beginning of the computation. The most important differences are that: the rules were applied in a different fashion (i.e., if at least one rule can be applied to a word w, we obtain all the words that are derived from the word w by applying exactly one of the possible rules at exactly one feasible position in the word w) and in the case of generating networks we don't need to worry about halting computations, acceptance or rejection (and synchronizing the different derivations of a word, in order to obtain the same outcome). All these differences lead to much less complicated constructions in that case.

Let us now consider the afore mentioned construction of a complete ANEP simulating an arbitrary ANEP via an intermediary Turing machine. We also propose a 2-steps construction, but the only computational model we use is the ANEP model. Also, the both steps rely on the same idea: we replace the nodes of the initial network with a group of nodes that simulate its job; this makes the construction simpler to apply, and easier to follow. Moreover, the complete network simulates in linear time the arbitrary network.

The result in Theorem 3 has a series of immediate consequences. First, it provides a canonical topology for ANEPs, allowing us to specify all the results and definitions in an uniform manner, without taking into account the particular topology of the network; in this respect, we may see this result as Normal-Form-result for ANEPs. Second, the transformation from an arbitrary network into a complete one is algorithmic, and can be applied automatically. Further, the number of nodes in the complete networks is greater only by a constant factor than the number of nodes of the simulated one. Finally, our simulation preserves the computational properties of the initial ANEP, thus, complete networks can be used to prove lower bounds for the time needed to solve a given problem: the most time-efficient ANEP-based solution of a problem can be implemented on a complete ANEP.

The proof of Lemma 2 is an example on how one can design an ANEP by putting together multiple subnetworks; this approach seems close to that of procedural programming. Theorem 3 shows that designing a greater ANEP from smaller subnetworks is an approach that can be used without being afraid that such a solution is no longer uniform (i.e., the network has very specific properties, and cannot be transformed efficiently to a general network, such as a complete one). Moreover, if an ANEP constructed from subnetworks works efficiently, so will do the complete variant of that ANEP.

5. Computational Complexity

We begin this section by recalling the definition of one-tape nondeterministic Turing machines. Such a machine is a construct $M = (Q, V, U, \delta, q_0, B, F)$, where Q is a finite set of states, V is the input alphabet, U is the tape alphabet, $V \subset U$, q_0 is the initial state, $B \in U \setminus V$ is the "blank" symbol, $F \subseteq Q$ is the set of final states, and δ is the transition mapping, $\delta : (Q \setminus F) \times U \to 2^{Q\times(U\setminus\{B\})\times\{R,L\}}$. In this paper, we assume without loss of generality that any Turing machine we consider has a semi-infinite tape (bounded to the left) and makes no stationary move. An instantaneous description (ID for short) of a Turing machine is a word that encodes the state of the machine, the content of its tape (that is, the finite string of non-blank symbols that exists on the tape), and the position of the tape-head, at a given moment of the computation. An ID is said to be initial if the tape contains a string from V^*, the tape-head is positioned on the first symbol of the tape, and the state is q_0; an ID is said to be final if the state encoded in it is the accepting or the rejecting state of the machine. A computation of a Turing machine on an input word consists of a sequence of IDs, starting with an initial ID, encoding the input word, such that each ID of the sequence can be transformed into the next one by executing a transition of the machine (i.e., rewriting the content of the tape and changing the state and the position of the tape-head according to the transition defined by the current state and the symbol scanned by the tape-head). A computation is said to be finite when it consists of a finite sequence of IDs that ends with a final ID; a computation is said to be accepting (respectively, rejecting), if and only if it is a finite computation and the final ID encodes the accepting state (respectively, rejecting state). Clearly, a nondeterministic machine may have more than one computation on an input string. All the possible computations of a nondeterministic machine on an input word can be described as a (potentially infinite) tree of IDs: each ID is transformed into its sons by simulating the possible moves of the machine; this tree is called computation-tree.

An input word is accepted by a nondeterministic Turing machine if and only if there exists an accepting computation of the machine on that word. The language accepted by the Turing machine is a set of all accepted words. We say a Turing machine M *decides* a language L if it accepts L and moreover halts on every input.

The length of a finite computation of a machine M on a given word w is the number of IDs that occur in that computation. The *shortest computations* of M on w are those computations of M on w whose length is minimal among all such computations. If M has no finite computation on w, the set of the shortest computations of M on w is empty; however, in this paper we discuss only about nondeterminitic polynomial machines, thus, machines that have only finite computations on every input. For such a machine, the shortest computations of M on w may be seen as the shortest paths from the root to the leaves in the tree of the computations of M on w.

The reader is referred to [7, 8, 17] for the classical definitions regarding

Turing machines, and the time and space complexity classes defined for them. However, here we focus on a different way of using nondeterministic Turing machines, that work in polynomial time, defined in [11].

Definition 1. *Let M be a nondeterministic polynomial Turing machine and let w be a word over the input alphabet of M. The word w is accepted by M* with respect to the shortest computations *if one of the shortest computations of M on w is accepting; w is rejected by M* w.r.t. the shortest computations *if all the shortest computations of M on w are rejecting. We denote by $L_{sc}(M)$ the language decided by M w.r.t. the shortest computations and by $PTime_{sc}$ the class of all the languages decided in this manner.*

It is not hard to see that the class of languages decided w.r.t. shortest computations by nondeterministic polynomial Turing machines with a single tape equals $PTime_{sc}$. In [11] the following result is shown:

Theorem 4. $PTime_{sc} = \mathbf{P}^{\mathbf{NP}[\log]}$.[2]

Further, one can show that a language is decided by ANEPs as efficiently as nondeterministic Turing machines deciding w.r.t. shortest computations, and vice versa.

Theorem 5.
i. *For every ANEP Γ, deciding a language L and working in polynomial time $P(n)$, there exists a nondeterministic polynomial single-tape Turing machine M, deciding L w.r.t. shortest computations; M can be constructed such that it makes at most $P^2(n)$ steps in a halting computation on a word of length n.*
ii. *For every nondeterministic polynomial single-tape Turing machine M, deciding a language L w.r.t. shortest computations, there exists a complete ANEP Γ, deciding the same language L. Moreover, Γ can be constructed such that $Time_{\Gamma}(n) \in \mathcal{O}(P(n))$, provided that M makes at most $P(n)$ steps in a halting computation on a word of length n.* $\square$

Proof. First we show that **i.** holds. The idea is quite simple. Since Γ works in polynomial time it follows that any computation of this ANEP on

[2]$\mathbf{P}^{\mathbf{NP}[\log]}$ is the class of problems solvable by a deterministic polynomial machine, that can make $\mathcal{O}(\log n)$ queries to an $\mathbf{NP}$ oracle on an input word of length n.

a word of length n has at most $P(n)$ steps. A nondeterministic Turing machine chooses nondeterministically a computation and simulates it, exactly like in the proofs from [13]. The only restriction is that the machine first computes $n+P(n)$ and, then, starts simulating, one by one, the evolutionary steps and communication steps of Γ, until at most $n+P(n)$ computational steps are performed. A computation halts when the word obtained can be accepted in the output node of Γ, and it accepts when the word contains the special accepting symbol, and rejects, otherwise. In this way, the shortest computation of the Turing machine on an input word w simulates the shortest possible derivation of Γ starting from that word and ending with a word accepted in the output node. The computation of the machine accepts (rejects), w.r.t. shortest computations, if and only if the computation of the ANEP accepts (rejects). Clearly, M makes at most $nP(n)+P^2(n)$ steps in a halting computation. This concludes the proof of **i.**.

Now we move on to show **ii.**. We start by constructing an incomplete network Γ' that simulates M. Then we just have to apply Theorem 3 and we obtain the desired result.

We assume, without losing generality, that M has exactly one accepting state q_{acc} and a blocking state (i.e., a state in which it enters and rejects) q_r. We assume that there are no transitions that can occur in these states.

The main idea is to construct Γ' from several subnetworks. The working alphabet of Γ' is denoted by $U_{\Gamma'}$.

First we have two nodes x_1 and x_2 that work as follows.

- $\mathcal{N}(x_1) = (\{\lambda \to B\}, \emptyset, \emptyset, U, \emptyset)$, $\alpha(x_1) = (w)$, $\beta(x_1) = r$.
- $\mathcal{N}(x_2) = (\{\lambda \to q_0\}, U, \emptyset, U \cup \{q_0\}, \emptyset)$, $\alpha(x_2) = (w)$, $\beta(x_2) = r$.
- We have the edge (x_1, x_2)

Then we have a node x_{center}, which acts as a central node of the network, as it basically controls the way the computation is conducted.

- $\mathcal{N}(x_{center}) = (\{q \to [q, a, q', b, X], q_{acc} \to q_{acc}, q_r \to q_r \mid q, q' \in Q,\ a, b \in U,\ X \in \{R, L\}$, and $(q', b, X) \in \delta(q, a)\}$, $U \cup Q, \emptyset, U_{\Gamma'}, Q)$, $\alpha(x_{center}) = (w)$, $\beta(x_{center}) = r$.
- We have the edge (x_2, x_{center}).

We also have an output node:

- $\mathcal{N}(x_{out}) = (\emptyset, \{q_{acc}, q_r\}, \emptyset, \emptyset, \emptyset)$, $\alpha(x_{out}) = (w)$.
- We have the edge (x_{center}, x_{out}).

For a transition $(q', b, L) \in \delta(q, B)$, let t denote the tuple (q, B, q', b, L). We have a subnetwork Γ'_t. This subnetwork has the following nodes and edges:

- $\mathcal{N}(x_1^t) = (\{B \to \bot\}, \{B\}, Q \cup (Q' \setminus \{[q, B, q', b, L]\}), \{\bot\}, \emptyset)$, $\alpha(x_1^t) = (w)$.
- $\mathcal{N}(x_2^t) = (\{\bot \to \lambda\}, \{\bot\}, \emptyset, U_{\Gamma'}, \emptyset)$, $\alpha(x_2^t) = (w)$, $\beta(x_2^t) = l$.
- $\mathcal{N}(x_3^t) = (\{\lambda \to B\}, U_{\Gamma'}, \{\bot\}, U_{\Gamma'}, \emptyset)$, $\alpha(x_3^t) = (w)$, $\beta(x_3^t) = l$.
- $\mathcal{N}(x_4^t) = (\{\lambda \to b\}, U_{\Gamma'}, \emptyset, U_{\Gamma'}, \emptyset)$, $\alpha(x_4^t) = (w)$, $\beta(x_4^t) = l$.
- $\mathcal{N}(x_5^t) = (\{c \to c' \mid c \in U\}, U_{\Gamma'}, \emptyset, \{c' \mid c \in U\}, \emptyset)$, $\alpha(x_5^t) = (w)$.
- $\mathcal{N}(x_{c,1}^t) = (\{\lambda \to c\}, \{c'\}, \emptyset, U_{\Gamma'}, \emptyset)$, $\alpha(x_{c,1}^t) = (w)$, $\beta(x_{c,1}^t) = l$, for $c \in U \setminus \{B\}$.
- $\mathcal{N}(x_6^t) = (\{c' \to \lambda \mid c \in U\}, \{c' \mid c \in U\}, \emptyset, U_{\Gamma'}, \emptyset)$, $\alpha(x_6^t) = (w)$, $\beta(x_6^t) = r$.
- $\mathcal{N}(x_7^t) = (\{[q, B, q', b, L] \to q'\}, U_{\Gamma'}, \{c' \mid c \in U\}, Q, \emptyset)$, $\alpha(x_7^t) = (w)$.
- We have the edges (x_i^t, x_{i+1}^t), for $i \in \{1, 2, 3, 4, 6\}$, and $\{(x_5^t, x_c^t), (x_c^t, x_6^t) \mid c \in U \setminus \{B\}\}$. Furthermore, we have the edges (x_{center}, x_1^t) and (x_7^t, x_{center})

For a transition $(q', b, R) \in \delta(q, B)$, let t denote the tuple (q, B, q', b, R). We have a subnetwork Γ'_t. This subnetwork has the following nodes and edges:

- $\mathcal{N}(x_1^t) = (\{B \to \bot\}, \{B\}, Q \cup (Q' \setminus \{[q, B, q', b, L]\}), \{\bot\}, \emptyset)$, $\alpha(x_1^t) = (w)$.
- $\mathcal{N}(x_2^t) = (\{\bot \to \lambda\}, \{\bot\}, \emptyset, U_{\Gamma'}, \emptyset)$, $\alpha(x_2^t) = (w)$, $\beta(x_2^t) = l$.
- $\mathcal{N}(x_3^t) = (\{\lambda \to B\}, U_{\Gamma'}, \{\bot\}, U_{\Gamma'}, \emptyset)$, $\alpha(x_3^t) = (w)$, $\beta(x_3^t) = l$.
- $\mathcal{N}(x_4^t) = (\{\lambda \to b\}, U_{\Gamma'}, \emptyset, U_{\Gamma'}, \emptyset)$, $\alpha(x_4^t) = (w)$, $\beta(x_4^t) = r$.
- $\mathcal{N}(x_5^t) = (\{[q, a, q', b, L] \to \bot_1, \bot_1 \to \bot_2, \bot_2 \to \bot_3, \bot_3 \to q'\}, U_{\Gamma'}, \{c' \mid c \in U\}, Q, \{\bot_i \mid i \in \{1, 2, 3\}\})$, $\alpha(x_5^t) = (w)$.

- We have the edges (x^t_i, x^t_{i+1}), for $i \in \{1, 2, 3, 4\}$. Also, we have the edges (x_{center}, x^t_1) and (x^t_4, x_{center})

For a transition $(q', b, L) \in \delta(q, a)$, with $a \neq B$, let t denote the tuple (q, a, q', b, L). We have a subnetwork Γ'_t. This subnetwork has the following nodes and edges:

- $\mathcal{N}(x^t_1) = (\{a \to \bot\}, \{a\}, Q \cup (Q' \setminus \{[q, B, q', b, L]\)\}, \{\bot\}, \emptyset)$, $\alpha(x^t_1) = (w)$.
- $\mathcal{N}(x^t_2) = (\{\bot \to \lambda\}, \{\bot\}, \emptyset, U_{\Gamma'}, \emptyset)$, $\alpha(x^t_2) = (w)$, $\beta(x^t_2) = l$.
- $\mathcal{N}(x^t_3) = (\{\lambda \to b\}, U_{\Gamma'}, \{\bot\}, U_{\Gamma'}, \emptyset)$, $\alpha(x^t_3) = (w)$, $\beta(x^t_3) = l$.
- $\mathcal{N}(x^t_4) = (\{c \to c' \mid c \in U\}, U_{\Gamma'}, \emptyset, \{c' \mid c \in U\}, \emptyset)$, $\alpha(x^t_5) = (w)$.
- $\mathcal{N}(x^t_{c,1}) = (\{\lambda \to c\}, \{c'\}, \emptyset, U_{\Gamma'}, \emptyset)$, $\alpha(x^t_{c,1}) = (w)$, $\beta(x^t_{c,1}) = l$, for $c \in U \setminus \{B\}$.
- $\mathcal{N}(x^t_5) = (\{c' \to \lambda \mid c \in U\}, \{c' \mid c \in U\}, \emptyset, U_{\Gamma'}, \emptyset)$, $\alpha(x^t_5) = (w)$, $\beta(x^t_5) = r$, for $c \in U \setminus \{B\}$.
- $\mathcal{N}(x^t_6) = (\{[q, a, q', b, L] \to \bot, \bot \to q'\}, U_{\Gamma'}, \{c' \mid c \in U\}, Q, \{\bot\})$ and $\alpha(x^t_6) = (w)$.
- We have the edges (x^t_i, x^t_{i+1}), for $i \in \{1, 2, 3, 4, 5\}$, and $\{(x^t_4, x^t_c), (x^t_c, x^t_5) \mid c \in U \setminus \{B\}\}$. Furthermore, we have the edges (x_{center}, x^t_1) and (x^t_6, x_{center})

For a transition $(q', b, R) \in \delta(q, a)$, with $a \neq B$, let t denote the tuple (q, a, q', b, R). We have a subnetwork Γ'_t. This subnetwork has the following nodes and edges:

- $\mathcal{N}(x^t_1) = (\{a \to \bot\}, \{a\}, Q \cup (Q' \setminus \{[q, B, q', b, L]\)\}, \{\bot\}, \emptyset)$, $\alpha(x^t_1) = (w)$.
- $\mathcal{N}(x^t_2) = (\{\bot \to \lambda\}, \{\bot\}, \emptyset, U_{\Gamma'}, \emptyset)$, $\alpha(x^t_2) = (w)$, $\beta(x^t_2) = l$.
- $\mathcal{N}(x^t_3) = (\{\lambda \to b\}, U_{\Gamma'}, \{\bot\}, U_{\Gamma'}, \emptyset)$, $\alpha(x^t_3) = (w)$, $\beta(x^t_3) = r$.
- $\mathcal{N}(x^t_4) = (\{[q, a, q', b, L] \to \bot_1, \bot_4 \to q'\} \cup \{\bot_i \to \bot_{i+1} \mid 1 \leq i \leq 3\}$, $U_{\Gamma'}$, $\{c' \mid c \in U\}$, Q, $\{\bot_i \mid i \in \{1, 2, 3, 4\}\})$, $\alpha(x^t_4) = (w)$.
- We have the edges (x^t_i, x^t_{i+1}), for $i \in \{1, 2, 3\}$. Furthermore, we have the edges (x_{center}, x^t_1) and (x^t_3, x_{center})

The way the network Γ' works is quite simple: the nodes x_1 and x_2 transform the input word w into wBq_0, and sends this word to x_{center}. Then, we can assume that the node x_{center} contains a word $w'Bqw''$, meaning that the machine M is in state q and the tape-head points to the first symbol of $w'B$ (assumption that holds after the first step). This node chooses which transition of M should be simulated next. Let us assume that the machine chooses to simulate a move given by the transition $(q, b, L) \in \delta(q, B)$. In this case, the word becomes $w'B[q, B, q', b, L]w''$ and is sent to the subnetwork Γ'_t. Here, in the nodes x_1^t and x_2^t we check if B is the leftmost symbol, and it is deleted; if B was not the leftmost symbol, the word is blocked, and this derivation stops. Then, in the nodes x_3^t and x_4^t another B is inserted (the machine-tape contains an infinite number of Bs, but we always keep just the first one in our words, and, in this case, this B was deleted), and the symbol b that replaced the deleted B is inserted at the leftmost end of the word. Now we should simulate the movement of the head, that is we should move one symbol from the rightmost end to the leftmost end. And this is done in the nodes x_5^t, x_c^t and x_6^t. The word first becomes $bB[q, B, q', aL]xc'y$, where $xcy = w''$, and, then, $cbB[q, B, q', b, L]xc'y$. Now c' is deleted if and only if it is the rightmost symbol, and we obtain the word $cbB[q, B, q', b, L]x$; otherwise, the word is lost. Finally, in x_7^t the word becomes $cbBq'x$, and it is sent back to x_{center}, where a new move of the machine will be simulated.

The other types of moves are simulated in a very similar manner by the respective subnetworks (actually, the case described above was the most involved one). Note that each move is simulated in exactly 9 evolutionary and 9 communication steps by the ANEP: 1 evolutionary and 1 communication step for choosing which move is simulated, and, then, 8 evolutionary and 8 communication steps for actually simulating the chosen move. In some of the cases, we needed to do some extra dummy-moves, just to be able to synchronize all the possible derivations.

The node x_{center} can also leave the word unchanged, in the case when it contains q_{acc} or q_r. Such words go to x_{out}. The special accepting symbol of the network is q_{acc}.

It is not hard to see that a halting computation of M, with t steps, is simulated by Γ' in $6 + 18t$ consecutive steps (evolutionary and communication), and it ends with a word in the node x_{out}. Also, the only words that can enter in the output node of Γ' have the form $w'Bq_cw''$, with $c \in \{acc, r\}$, provided that the configuration $w''q_cw'B$ can be reached by M. Also, such a word enters in that node only after $6 + 18k$ steps, with $k \in I\!N$. So the words that will be accepted the first in the node x_{out} are exactly those encoding the final configurations reached in the shortest computations of M.

Therefore, the words that are accepted (respectively, rejected) by Γ' are exactly those that are accepted (respectively, rejected) by M, w.r.t. shortest computations. $\square$

It is worth noting that this proof is different from other similar proofs. like those from [16, 13], as here every move of the Turing machine is simulated in exactly 9 evolutionary and 9 communication steps by the ANEP; in the previous proofs different moves of the machine were simulated in different number of ANEP-steps.

As a consequence of Theorem 5 we obtain the following Theorem.

Theorem 6. $\mathbf{PTime}_{ANEP} = \mathbf{P}^{\mathbf{NP}[\log]}$. $\square$

This result seems interesting to us as we are not aware of any other characterization of the class $\mathbf{P}^{\mathbf{NP}[\log]}$ by computational complexity classes defined for bio-inspired computing models. Note, once more, that all the problems in $\mathbf{P}^{\mathbf{NP}[\log]}$ can be solved by networks with the same topology, so, in a way, the solution to a problem is not based on the topology of the network but on the algorithm implemented by that network.

Finally, we show that, in fact, one can design complete ANEPs working faster than nondeterministic Turing machines. This shows that solving a problem by nondeterministic Turing machines and then simulate such machines by NEPs does not lead to an optimal ANEP-based solution to that problem. Also, this shows that the mechanism implemented by ANEPs does more than replacing the nondeterminism by massive parallelism.

Example 1. *Let $L = \{a^n b \mid n \in I\!N, n \geq 1\}$. It is not hard to see that any nondeterministic Turing machine, deciding L in the classical way, or w.r.t. shortest computations, or using oracles, and with an arbitrary number of tapes, makes at least a linear number of moves before it stops on an input word. However, L can be accepted in constant time by a complete ANEP.*

- *We construct an ANEP Γ with an underlying graph with 6 nodes x_1, x_2, x_3, x_4, x_5, and x_{out}, and edges (x_1, x_2), (x_1, x_4), (x_2, x_3), (x_2, x_5), (x_3, x_{out}), (x_4, x_5), (x_5, x_{out}).*
- *This ANEP works as follows: in the node x_1 (which is also the input node) a symbol b from the input word is replaced with an X, and all the*

words obtained exist the node. They can go to x_2, where they enter only if they have an X (so only words that initially contained at least a symbol b can be further processed in this node); all the other words go to x_4. In x_2 the rightmost symbol of any words deleted if and only if it is an X. Further, the words can go to x_3, if they have no X and still have an a, or to x_5, otherwise. Clearly, at least a word enters x_3 if and only if the input word had the form $a^n b$, with $n \in I\!N, n \geq 1$. In this node a special symbol μ is inserted in the words. The nodes x_4 and x_5 do not process the words in any way. All the words obtained in x_3 and x_5 go to x_{out}, and the input word is accepted if and only if one of the words in x_{out} contains μ.

- *From the explanations above it follows clearly that Γ accepts L, and any word is decided in at most 3 evolutionary steps and communication steps.*

- *By Theorem 3 we obtain that there exists a complete network Γ' accepting L in constant time.*

6. Yet Another Model and Conclusions

During the writing of this paper we considered several variants of definitions for the decision of a language by ANEPs. The variant presented so far seemed to us quite motivated and lead to simpler proofs. However, there is an equally motivated variant that we would like to present, informally, in the following.

The idea behind this new definition is the same as in the previous case: we first define a halting condition and then we define a method to take the decision, once the computation halted. In this case, instead of having a special symbol, a network will contain an additional distinguished node x_A. The computation of such a network on an input word works exactly as in the case discussed before. A computation halts as soon as a word enters the node x_O; once the computation halts, the input word is accepted if the node x_A contains at least a word and rejected otherwise.

The main difference between this definition and the one previously discussed is that in this case the halting condition and the method of deciding the input are of similar nature: we simply check whether the configuration of a node is empty or not. However, this model is a little more complicated and is farther away from the initial definition of the NEPs [16], as now we have more than one node that is involved in the termination of a computation. This model seems also harder to use; in fact, we were able to show results similar to the ones discussed previously in this paper only by simulations that basically rely on the idea of introducing in the processed words, at a given point of the computation, a symbol that signals the acceptance of the input word (thus, has the same meaning as μ from the previous definition) and then halting somehow the computation.

It is not hard to see that this new model simulates efficiently the model introduced in Section 2. Indeed, assume that we are given an ANEP that decides a language according to the method previously discussed and has an output node x_O; we add to this network two new nodes x'_O and x'_A and the edges (x_O, x'_O) and (x_O, x'_A). The node x'_O allows any string to enter and is used to signal the halting of the computation as soon as a string enters in it, while x'_A allows in only the strings that have the symbol μ and is used in order to take the correct decision on the input word (that is, to accept the input string when it is not empty in the moment when the computation stops).

To show that the new model can be simulated efficiently by the model introduced in Section 2 we make, once more, use of the Turing machines deciding w.r.t. shortest computations. Indeed, let Γ be an ANEP and x_O be its node that is used to halt the computations and x_A its node that it is used to take the decision, as described above. We construct a Turing machine M that simulates, step by step, two derivations of Γ starting with the same word. The machine halts as soon as one of these derivations reaches a word that can enter x_O and accepts if the other derivations produced (in the same number of steps) a word that can enter x_A, rejecting otherwise. Let us see which is the language decided with respect to the shortest computations by M. The shortest computations of M are exactly those in which one of the shortest derivations of Γ from the input word, ending with a word that can enter in x_O, is simulated; the halting computations of Γ have exactly as many steps as such a derivation. Also, Γ accepts if and only if besides the shortest derivation that produces the word accepted in x_O there is another derivation of the input word that produces, in the same number of steps, a word accepted in x_A. Thus, a word is accepted by Γ if there exist two derivations with minimum number of steps that produce from the input word a word accepted in x_O and, respectively, a word accepted in x_A; but this is equivalent to the fact that M accepts the input word (as one of its shortest computations will consist in simulating exactly these two computations). Similarly, a word is rejected by Γ if there are no two derivations with minimum number of steps that produce from the input word a word accepted in x_O and, respectively, a word accepted in x_A; once more, this is equivalent to the fact that all the shortest computations of M are rejecting. In conclusion, M decides exactly the same language as Γ, and the shortest computations of M on an input word have as many steps as Γ does in its computation on that word. By Theorem 5 we get that the new model of deciding NEPs can be simulated efficiently by the model introduced in Section 2.

These last results, as well as the results presented before them in this paper, make us think that nondeterministic machines deciding with respect to shortest computation may become useful when defining complexity measures for other bio-inspired computational models, as well. It seems appealing to us to study how the computation of other such models can be expressed in terms of shortest computations of classical machines, and in what measure the complexity classes defined with respect to shortest computations have corresponding complexity classes defined for bio-inspired models. This new approach of bio-inspired computational models could provide a deeper understanding of their real computational capacities and could lead to a simplification of their definitions, in such a manner that the natural restrictions are kept and the artificial ones are discarded (as it was the case of deciding networks of evolutionary processors).

7. Acknowledgements

The author thanks Professor Jürgen Dassow for his advices in writing this paper. Nevertheless, the author would also like to thank to the anonymous referees for their comments and suggestions that led to a better presentation of this paper.

The author's work was partially supported by the *Alexander von Humboldt Foundation*, through a two-years Research Fellowship at the Otto-von-Guericke University Magdeburg, Germany, that ended in June 2011 and a subsequent Return Fellowship at the University of Bucharest, Romania, between July 2011 and October 2011. His work is currently supported by *DFG - Deutsche Forschungsgemeinschaft* through the grant 582014.

References

[1] A. Alhazov, E. Csuhaj-Varjú, C. Martín-Vide, Y. Rogozhin, On the size of computationally complete hybrid networks of evolutionary processors, Theor. Comput. Sci. 410 (2009) 3188–3197.

[2] P. Bottoni, A. Labella, F. Manea, V. Mitrana, J.M. Sempere, Filter position in networks of evolutionary processors does not matter: A direct proof, in: R.J. Deaton, A. Suyama (Eds.), DNA, volume 5877 of *Lecture Notes in Computer Science*, Springer, 2009, pp. 1–11.

[3] E. Csuhaj-Varjú, V. Mitrana, Evolutionary systems: A language generating device inspired by evolving communities of cells, Acta Inf. 36 (2000) 913–926.

[4] E. Csuhaj-Varjú, A. Salomaa, Networks of parallel language processors, in: G. Paun, A. Salomaa (Eds.), New Trends in Formal Languages, volume 1218 of *Lecture Notes in Computer Science*, Springer, 1997, pp. 299–318.

[5] J. Dassow, F. Manea, Accepting hybrid networks of evolutionary processors with special topologies and small communication, in: I. McQuillan, G. Pighizzini (Eds.), DCFS, volume 31 of *EPTCS*, pp. 68–77.

[6] J. Dassow, F. Manea, B. Truthe, On normal forms for networks of evolutionary processors, in: C.S. Calude, J. Kari, I. Petre, G. Rozenberg (Eds.), UC, volume 6714 of *Lecture Notes in Computer Science*, Springer, 2011, pp. 89–100.

[7] M.R. Garey, D.S. Johnson, Computers and Intractability: A Guide to the Theory of NP-Completeness, W. H. Freeman, 1979.

[8] J. Hartmanis, R.E. Stearns, On the Computational Complexity of Algorithms, Trans. Amer. Math. Soc. 117 (1965) 533–546.

[9] W.D. Hillis, The Connection Machine, MIT Press, Cambridge, MA, USA, 1986.

[10] R. Loos, F. Manea, V. Mitrana, Small universal accepting hybrid networks of evolutionary processors, Acta Inf. 47 (2010) 133–146.

[11] F. Manea, Deciding according to the shortest computations, in: B. Löwe, D. Normann, I.N. Soskov, A.A. Soskova (Eds.), CiE, volume 6735 of *Lecture Notes in Computer Science*, Springer, 2011, pp. 191–200.

[12] F. Manea, Deciding networks of evolutionary processors, in: G. Mauri, A. Leporati (Eds.), Developments in Language Theory, volume 6795 of *Lecture Notes in Computer Science*, Springer, 2011, pp. 337–349.

[13] F. Manea, M. Margenstern, V. Mitrana, M.J. Pérez-Jiménez, A new characterization of np, p, and pspace with accepting hybrid networks of evolutionary processors, Theory Comput. Syst. 46 (2010) 174–192.

[14] F. Manea, C. Martín-Vide, V. Mitrana, On the size complexity of universal accepting hybrid networks of evolutionary processors, Mathematical Structures in Computer Science 17 (2007) 753–771.

[15] F. Manea, C. Martín-Vide, V. Mitrana, Accepting networks of evolutionary word and picture processors: A survey, in: C. Martín-Vide (Ed.), Scientific Applications of Language Methods, volume 2 of *Mathematics, Computing, Language, and Life: Frontiers in Mathematical Linguistics and Language Theory*, World Scientific, 2010, pp. 525–560.

[16] M. Margenstern, V. Mitrana, M.J. Pérez-Jiménez, Accepting hybrid networks of evolutionary processors, in: C. Ferretti, G. Mauri, C. Zandron (Eds.), DNA, volume 3384 of *Lecture Notes in Computer Science*, Springer, 2004, pp. 235–246.

[17] C.M. Papadimitriou, Computational Complexity, Addison-Wesley, Reading, Massachusetts, 1994.

[18] G. Rozenberg, A. Salomaa, Handbook of Formal Languages, Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1997.