Implementation of an NEP in Java

David Batard, Víctor Martínez

Abstract
TheNetworks of Evolutionary Processors (NEPs) are computing mechanisms directly inspired from the behavior of cell populations more specifically the point mutations in DNA strands.These mechanisms are been used for solving NP-complete problems by means of a parallel computation postulation.This paper describes an implementation of the basic model of NEP and includes the possibility of designing some of the most common variants of it by means of a graphic user interface which eases the configuration of a given problem. It is a system designed to be used in a multicore processor in order to benefit from the multi thread use.
Keywords: NEP, Evolutionary processors, natural computing, Implementation.

I. Introduction

Networks of Evolutionary Processors (NEP) are a rather new computing mechanism directly inspired from the behavior of cell populations. Every cell is described by a set of words, evolving by mutations, which are represented by operations on these words, resembling the manner carried out by DNA strings [Păun, 1998]. At the end of the process, only the cells with correct strings will survive. The main potential in this model is the simultaneous way it develops for which a basic architecture for parallel and distributed computing is required consisting on several processors, each of them placed in a node of a virtual complete graph, which are able to handle data associated with the respective node. Each node processor acts on the local data in accordance with some predefined rules. Local data is then sent through the network according to well-defined protocols. Only data which is able to pass a filtering process can be communicated. This filtering process may be required to satisfy some conditions imposed by the sending processor, by the receiving processor, or by both of them. All the nodes simultaneously send their data and the receiving nodes also simultaneously handle all the arriving messages, according to specific strategies. In addition, the data in the nodes is organized in the form of large multiset of words where each word could appear in an arbitrarily large number of copies and all the copies are processed in parallel so that every possible action takes place.

This basic model has evolved to others which extend not only the definition but the applications. In this case we consider the hybrid networks of evolutionary processors (HNEP) where the rules in every processor could be applied differently opposed to the basic model as described in [Martín-Vide, 2003].Also other variants can be considered as they all share the same general characteristics.

In this paperwe describe the initial work of implementation of a general NEP which can be thought to represent the most common variations of the basic model, considering the concurrent way it was conceived to perform and having a graphic user interface for an easier way of defining it and getting the outcomes.

II. Basic concepts

A *network of evolutionary processors* of size n is a construct:
$\Gamma = (V, N_1, N_2, \dots, N_n, G)$,
where V is an alphabet of symbols and for each $1 \leq i \leq n$, $N_i = (M_i, A_i, PI_i, FI_i PO_i, FO_i)$ is the i-th evolutionary node processor of the network. The parameters of every processor are:
M_i is a finite set of evolution rules of one of the following forms only:

– $a \rightarrow b$ $a, b \in V$ (substitution rules),
– $a \rightarrow \varepsilon$ $a \in V$ (deletion rules),
– $\varepsilon \rightarrow a$ $a \in V$ (insertion rules),

In this case for the hybrid NEP we are considering each deletion node or insertion node having its own working mode (performs the operation at any position, in the left-hand end, or in the right-hand end of the word) and different nodes are allowed to use different ways of filtering. Thus, the same network may have nodes where the deletion operation can be performed at arbitrary position and nodes where the deletion can be done only at the right-hand end of the word.

A_i is a finite set of strings over V. The set A_i is the set of initial strings in the i-th node. We consider that each string appearing in any node at any step has an arbitrarily large number of copiesin that node.

$PI, FI \subseteq V$are the input permitting/forbidding contexts of the processor, while $PO, FO \subseteq V$ are the output permitting/forbidding contexts of the processor. These filters can work in four different way as described below:

For two disjoint subsets P and F of an alphabet V and a word over V, we define the predicates $\varphi^{(1)}$ and $\varphi^{(2)}$ as follows:

$$\varphi^{(1)}(w;P,F) \equiv P \subseteq alph(w) \quad \wedge \quad F \cap alph(w) = \emptyset$$

$$\varphi^{(2)}(w;P,F) \equiv alph(w) \cap P \neq \emptyset \quad \wedge \quad F \cap alph(w) = \emptyset.$$

$$\varphi^{(3)}(w;P,F) \equiv alph(w) \subseteq P$$

$$\varphi^{(4)}(w;P,F) \equiv P \subseteq alph(w) \quad \wedge \quad F \nsubseteq alph(w)$$

The construction of these predicates is based on random-context conditions defined by the two sets P (permitting contexts) and F (forbidding contexts). For every language $L \subseteq V^*$ and $\beta \in \{(1),(2),(3),(4)\}$, we define:

$$\varphi^{\beta}(L,P,F) = \{w \in L| \quad \varphi^{\beta}(w;P,F)\}.$$

Finally, G= $(\{N_1,N_2,\dots,N_n\}$, E) is an undirected graph called the *underlying graph* of the network.The edges of G, that is the elements of E, are given in the form of sets of two nodes.

By a configuration (state) of a NEP as above we mean an n-tuple$C = (L_1,L_2,\dots,L_n)$, with $L_i \subseteq V^*$for all $1 \leq i \leq n$. A configuration represents the sets of strings which are present in any node at a given moment. The initial configuration of the network is $C_0 = (A_1,A_2,\dots,A_n)$. A configuration can change either by an evolutionary step or by a communicating step. When changing by an evolutionary step, each component L_i of the configuration is changed in accordance with the evolutionary rules associated with the node i.

Formally, we say that the configuration $C_1 = (L_1,L_2,\dots,L_n)$, directly changesinto the configuration $C_2 = (L_1',L_2',\dots,L_n')$ by an evolutionary step, written as $C_1 \Longrightarrow C_2$ if $L_i'$is the set of strings obtained by applying the rules of $R_i$to the strings in $L_i$as follows:

(*i*) If the same substitution or deletion rule may replace different occurrences of the same symbol within a string, all these occurrences must be replaced within different copies of that string. The result is a multiset in which every string that can be obtained appears in an arbitrarily large number of copies.

(*ii*) An insertion rule is applied at any position in a string. Again, the result is a multiset in which every string, that can be obtained by application of an insertion rule to an arbitrary position in an existing string, appears in an arbitrarily large number of copies.

(*iii*) If more than one rule, no matter its type, applies to a string, all of them must be used for different copies of that string.

When changing by a communication step, each node processor $N_i$sends all copies of the strings it has which are able to pass its output filter to all the node processors connected to N_i and receives all copies of the strings sent by any node processor connected with N_i providing that they can pass its input filter.

Formally, we say that the configuration $C_1 = (L_1,L_2,\dots,L_n)$directly changes into the configuration $C_2 = (L_1',L_2',\dots,L_n')$ by a communication step, written as $C_1 \vdash C_2$if

$$L_i' = L_i \backslash \{w \mid w \in L_i \cap PO_i\} \cup \bigcup_{\{N_I,N_j\}\in E} \{x \mid x \in L_j \cap PO_j \cap P_i\}$$

for every $1 \leq i \leq n$.

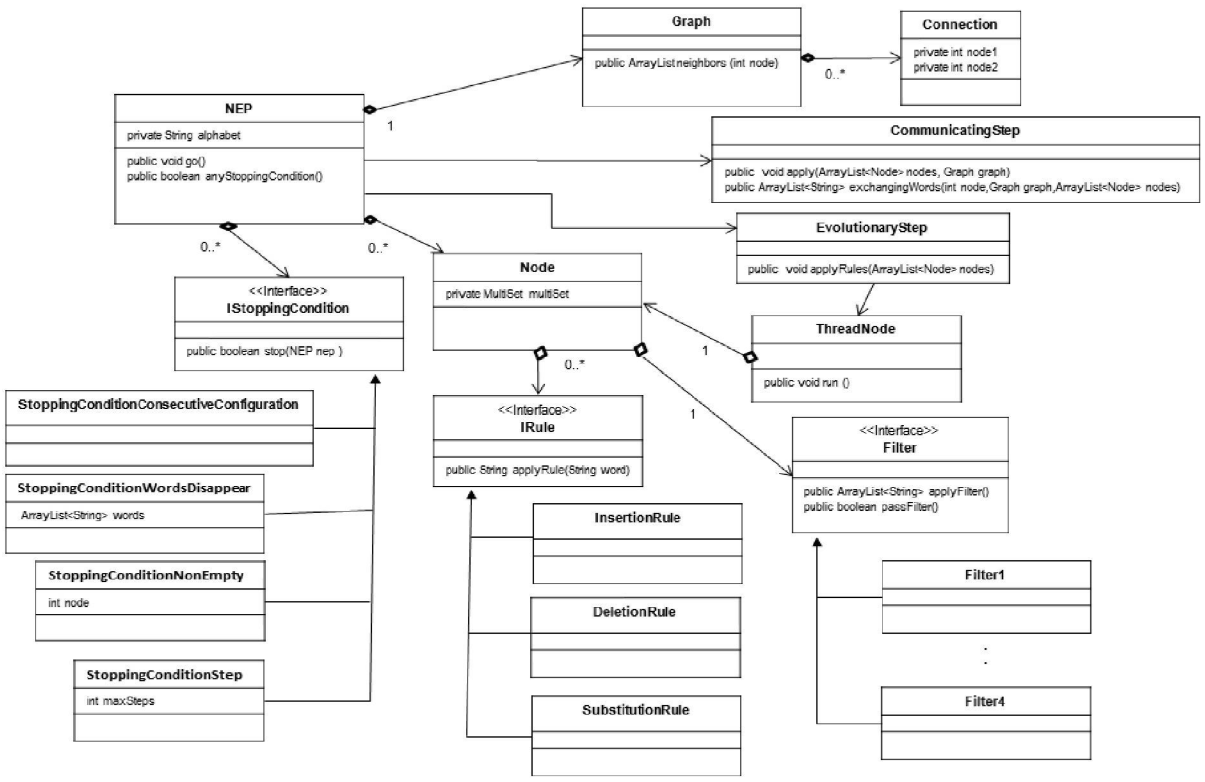
Let $\Gamma = (V, N_1,N_2,\dots,N_n)$ be an NEP. By a computation in Γwe mean a sequence of configurations $C_0,C_1,C_2,\dots$, where $C_0$is the initial configuration, $C_{2i} \Longrightarrow C_{2i+1}$and $C_{2i+1} \vdash C_{2i+2}$for all $i \geq 0$.

If the sequence is finite, we have a finite computation. The result of any finite or infinite computation is a language which is collected in a designated node called the output node of the network. If one considers the output node of the network as being the node k, and if C_0, C_1, . . . is a computation, then all strings existing in the node k at some step t - the k-th component of C_t- belong to the language generated by the network. Let us denote this language by $L_k(\Gamma)$. The time complexity of computing a finite set of strings Z is the minimal number of steps t in a computation C_0, C_1, . . . , C_t. . . such that Z is a subset of the k-th component of C_t.

III. Implementation

This NEP implementation is thought to be a base for future additions and adaptations as the discoveries in this field are moving forward, so the class structure is considered to be lithe by means of the use of interfaces along with abstract classes which gather the common and required features of the formal definition.

The relation and dependencies of classes for simulating the NEP model are showed in Figure 1, in a simplified way.



Graph
public ArrayListneighbors (int node)
Connection
private int node1
private int node2
0..*
NEP
private String alphabet
public void go()
public boolean anyStoppingCondition()
1
CommunicatingStep
public void apply(ArrayList<Node> nodes, Graph graph)
public ArrayList<String> exchangingWords(int node,Graph graph,ArrayList<Node> nodes)
EvolutionaryStep
public void applyRules(ArrayList<Node> nodes)
0..*
0..*
<<Interface>>
IStoppingCondition
public boolean stop(NEP nep)
Node
private MultiSet multiSet
ThreadNode
public void run ()
1
0..*
1
StoppingConditionConsecutiveConfiguration
StoppingConditionWordsDisappear
ArrayList<String> words
StoppingConditionNonEmpty
int node
StoppingConditionStep
int maxSteps
<<Interface>>
IRule
public String applyRule(String word)
InsertionRule
DeletionRule
SubstitutionRule
<<Interface>>
Filter
public ArrayList<String> applyFilter()
public boolean passFilter()
Filter1
Filter4

Fig. 1. Class diagram of the main design

We can see in this diagram the relation among all the involved classes. We thought the main class **NEP** should be in charge of keeping reference to the rest of the well-known components of a NEP such as the alphabet in a form of a **String**in which every character is standing for a symbol, the graph, the list of nodes or processors and the stopping conditions. This class it is also in charge of initiating and controlling the evolutionary and communication processes trough the method **go()**which is in charge of doing this rotation of steps in accordance with the established model and controlling with the **anyStoppingCondition()** the possibility of stopping the processing due to the occurrence of any of the required conditions for stopping the computation. This class also interacts with others devoted to the data management and NEP configuration procedure as well as for retrieving the outcomes of a calculation.
For the **Graph** class we have an array list of Connection which is a class describing a connection between two nodes by keeping the names of the nodes related in a form of**int** values,which are also the numbers of the positions of each node in the list of nodes. The method **neighbors(int node)**of this class was thought to be of use in communication stepsfor retrieving the list of nodes connected to a given one for next exchange of words among them.
As we mentioned, the NEP stops when at least one of the stopping condition is met. In this case we have considered covering the most common ones as in jNEP in[Rosal, 2008].The Figure 1 shows how from an interface it was conceived the general structure of the stopping condition by means of the **stop(NEP nep)**method allowing future variant to be considered without extended variation since each of the four already in the implementing classes(**Stopping Condition Consecutive Configuration, Stopping Condition Words Disappear, Stopping Condition Non Empty, Stopping Condition Step**) of interface **IStopping Condition** share the same method but differing in theirs attributes. Explaining each one of them we say that for the Stopping **Condition Consecutive Configuration**to succeed stopping the computation if two consecutiveidentical configurations are found once communication and evolutionary steps were performed. For the **Stopping Condition Words Disappear**to trigger the stop if none of the words listed are in the NEP. The **Stopping Condition Non Empty**if one of the nodesis non-empty and the **Stopping Condition Step** for stopping after a given amount of steps.

The processors are other key components of a NEP simulation, in this they are referred as nodes. The Node class which has a **MultiSet**reference, standing for the group of words of the processor treated as an **ArrayList<String>**and implementing a group of methods useful for the filtering and rules application processes. In the nodes we have a list of rules defined by the**IRule** interface and instantiated buy a group of the classes: **InsertionRule, DeletionRule, SubstitutionRule**the ones are meant to cover the basic model of NEP,implementing the **applyRule(String word)**method of the interface and inheriting from the **Rule** class, not considered in the class diagram for space reasons.The common attributes come from **Rule**as **symbol,** torepresent the symbolto apply the rule to, coming as a **String**but so far considering the only character it carries and the attribute **how**, also a **String** referring the way it has to be done as in the position the rule has to be used, having one of this values: **left, right, any.**This also reworks the basic model which was conceived for applying rules at the end of the word.

Nodes have two filters as attributes, the **inputfilter** and the **outputfilter**which are instantiated from one of the four filtering classes (**Filter1, Filter2, Filter3, Filter4**) according to the level of strength in the filtering processes described in [Martín-Vide, 2003], each one implementing the **IFilter**interface where the **applyFilter()**returns a list of words able to pass the filter and the **passFilter()**for considering a single word.

EvolutionaryStepis the class conceived to manage the list of nodes for the purpose of performing an evolutionary step; for doing that and by means of generating a new thread by each node in every step. For that, it requires the **ThreadNode** class which uses a node reference to access the node multiset and rules. The **run()** method in every thread applies randomly the rules to every word and every copy till no more can be applied. Once the evolutionary step is finished the NEP commands to the method **apply(ArrayList<Node>nodes,Graph graph)**of the **CommunicatingStep**class to proceed with the exchange of words using the defined filters of each node.

The need of a way for nicely defining and storing the different designs of NEP is something considered in this implementation. Some previous application for this models do not present a solution for this matter but for only for storing and reading from a configuration file which the user has to learn how to create. In the Figure 2 we show the class diagram for the Input/Output of the different configurations.

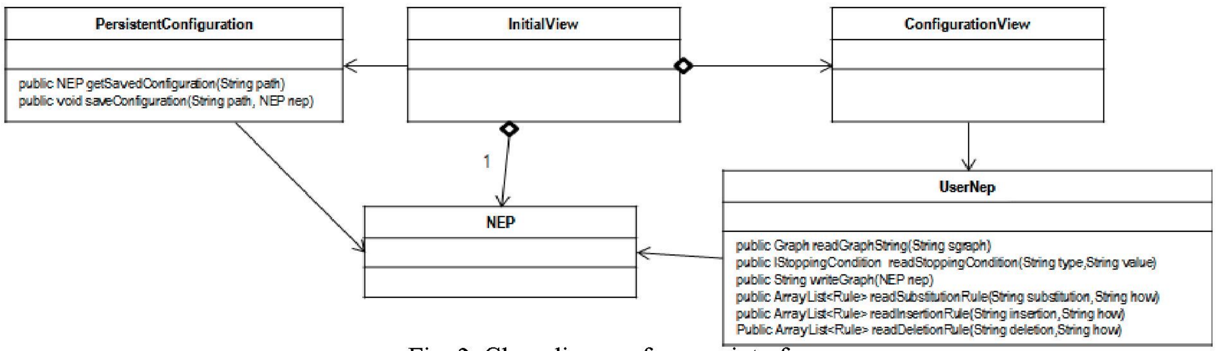
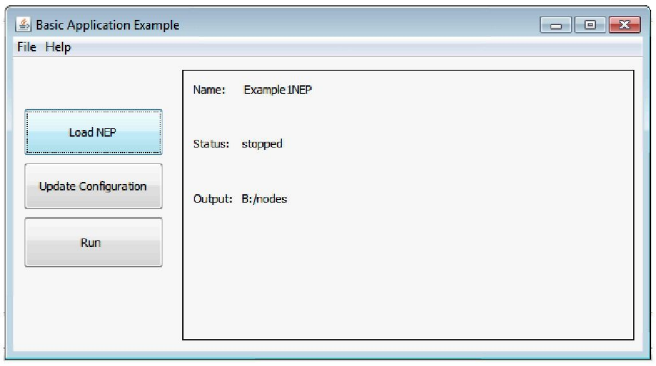

PersistentConfiguration
public NEP getSavedConfiguration(String path)
public void saveConfiguration(String path, NEP nep)
InitialView
ConfigurationView
1
NEP
UserNep
public Graph readGraphString(String sgraph)
public IStoppingCondition readStoppingCondition(String type,String value)
public String writeGraph(NEP nep)
public ArrayList<Rule> readSubstitutionRule(String substitution, String how)
public ArrayList<Rule> readInsertionRule(String insertion, String how)
Public ArrayList<Rule> readDeletionRule(String deletion, String how)

Fig. 2. Class diagram for user interface.

Once the **InitialView** starts it is necessary to upload the file containing the NEP description, specified in a jsonsyntax, to the NEP instance referenced by this view and by means of the **getSavedConfiguration(String path)**method of the **PessistentConfiguration** class, accessible through the **Load NEP** button as showed in Figure 3 which allows us to locate y select the desired configuration.



Name: Example1NEP

Status: stopped

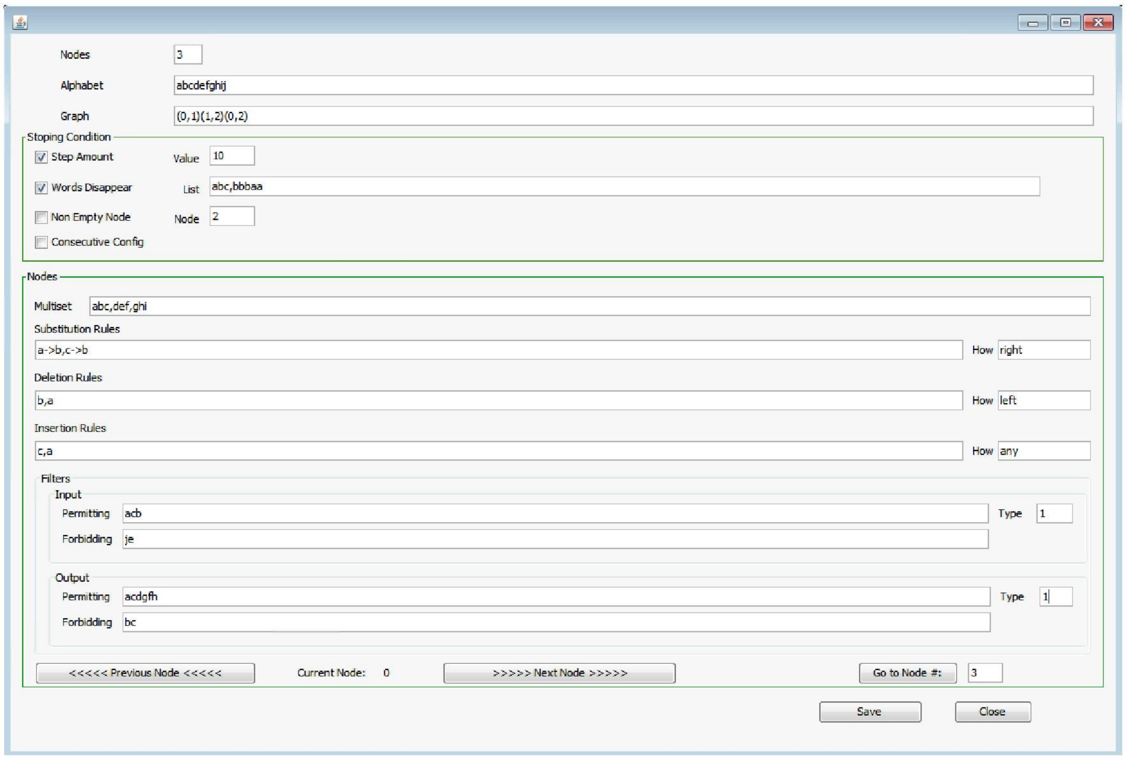
Output: B:/nodes

Fig. 3. Initial View of NEP system.

The **ConfigurationView**responsible for the main interaction with the user permitting the creation and modification, of a current NEP. The use of the **UserNep** class it is given as a translation mechanism between the form of writing in the visual components and the NEP object oriented structure. For example in Figure 4 for defining the graph, a structure in the way of tuples of nodes numbersassociated with a comma and delimited through brackets as follows: (0,1)(1,2)(0,2). For that purpose the **readGraphString(String sgraph)** is in charge of the translation of a string representation of the graph to a **Graph** class form, in the opposite direction the **writeGraph(NEP nep)**method is responsible of putting in a string form the **Graph** content required for visualizing it in the **ConfigurationView.**

In this view the comma is used for separating the individual elements as words in the multiset, also the symbolization -> for describing the rules having consequent, not for the deletion or insertion ruleswhich only requires the one different from the empty symbol.

The Node panel also in Figure 4 allows us to move through the different nodes using the <<<<<Previous Node<<<<<, and >>>>>Next Node >>>>> buttons as well as going directly to the desired one if the configuration is too long.



Nodes: 3
Alphabet: abcdefghij
Graph: (0,1)(1,2)(0,2)

Stoping Condition
- [✓] Step Amount — Value: 10
- [✓] Words Disappear — List: abc,bbbaa
- [ ] Non Empty Node — Node: 2
- [ ] Consecutive Config

Nodes
Multiset: abc,def,ghi
Substitution Rules: a->b,c->b — How: right
Deletion Rules: b,a — How: left
Insertion Rules: c,a — How: any

Filters
Input
Permitting: acb — Type: 1
Forbidding: je
Output
Permitting: acdgfh — Type: 1
Forbidding: bc

Fig. 4.ConfigurationView

IV. Conclusion

In this paper we have described the design and implementation of an abstract computer devise called NEP, aiming to achieve a solution for fitting the most common variants. The use of a graphic user interface is also one of the first attempts of this type of simulations allowing a fast and pleasant configuration of the NEP.

This work plays to be a starting tool for future analysis of the different variants the NEP family as it will be submitted to forthcoming developments in order complete a better and more complete solution.

Bibliography

[1] [Bel-Enguix, 2008] Bel-Enguix G., Jiménez M.: A BioInspired Model for Parsing of Natural Languages.Studies in Computational Intelligence, Springer Verlag, Berlin, 2008, Vol. 129/2008, 369-378.

[2] [Bottoni, 2011] Bottoni, P., Labella, A., Manea, F., Mitrana, V., Petre, I., Sempere, J.: Complexity-preserving simulations among three variants of accepting networks of evolutionary processors, Springer Science+Business Media B.V. 2011.

[3] [Castellanos, 2001] Castellanos, J., Martin-Vide, C., Mitrana, V., Sempere, J.: Solving NP-complete problems with networks of evolutionary processors. Proceedings of IWANN 2001, LNCS 2084, Springer-Verlag, 2001, 621–628.

[4] [Castellanos, 2003] Castellanos, J., C. Martin-Vide, V. Mitrana& J.M. Sempere, Networks of Evolutionary processors, ActaInformatica. 39 (2003): 517-529.

[5] [Manea, 2004] Manea, F., Martin-Vide, V., &Mitrana, V., Solving 3CNF-SAT and HPP in linear time using WWW, Proc. of MCU 2004, LNCS, in press.

[6] [Manea, 2006] Manea F., Martín-Vide C., Mitrana V.: A Universal Accepting Hybrid Network of Evolutionary Processors, Electronic Notes in Theoretical Computer Science,2006, Vol. 135, 15-23.

[7] [Martín-Vide, 2003] Martin-Vide, C., Mitrana, V., Perez-Jimenez, M., & Sancho-Caparrini, F., Hybrid networks of evolutionary processors. In: Proc. of GECCO 2003, LNCS 2723, Springer Verlag, Berlin, 2003.

[8] [Păun,1998] Păun, Gh.,Rozenberg, G., &Salomaa, A., DNA Computing. New ComputingParadigms, Berlin, Springer,1998.

[9] [Rosal, 2008] Rosal, E., Nuñez, R., Casteñeda, C., Ortega, A. Simulating NEPs in a cluster with jNEP.Proceedings of ICCCC, 2008.