

# TRIANGLE

LLENGUATGE, LITERATURA, COMPUTACIÓ  
LANGUAGE, LITERATURE, COMPUTATION

7

## Networks of Bioinspired Processors

Alfonso Ortega de la Puente & Marina de la Cruz Echeandía (eds.)

# TRIANGLE 7

March 2012

## Networks of Bioinspired Processors

Alfonso Ortega de la Puente & Marina de la Cruz Echeandía (eds.)

Manuel Alfonseca, Eloy Anguiano Rey, Fernando Arroyo Montoro,  
Carlos Castañeda, Juan Castellanos, Miguel Cuéllar, Juan de Lara, Emilio del Rosal,  
Antonio Jiménez Martínez, Robert Mercaş, Victor Mitrana, Carmen Navarrete Navarrete,  
Rafael Nuñez Hervás, Diana Pérez, Alexander Perekrestenko, José Miguel Rojas Siles,  
Eugenio Santos, José M. Sempere

LLENGUATGE, LITERATURA, COMPUTACIÓ  
LANGUAGE, LITERATURE, COMPUTATION



Tarragona, 2012

**Revista TRIANGLE**

**President:** Antonio Garcia Español

**Consell editorial:** M. Angeles Caamaño, Natalia Català,  
M. Dolores Jiménez López, M. José Rodríguez Campillo.

Gemma Bel-Enguix per la Sèrie Linguistics, Biology and Computation,  
i Esther Forgas per la Sèrie Español Lengua Extranjera.

Edita: Publicacions URV

ISSN: 2013-939X

DL: T-1492-2010

Per a més informació de la revista consulteu la pàgina  
<http://revista-triangle.blogspot.com/>

Publicacions de la Universitat Rovira i Virgili:

Av. Catalunya, 35 - 43002 Tarragona

Tel. 977 558 474 - Fax: 977 558 393

[www.urv.cat/publicacions](http://www.urv.cat/publicacions)

[publicacions@urv.cat](mailto:publicacions@urv.cat)

Arola Editors: Polígon Francolí, parcel·la 3, nau 5 - 43006 Tarragona

Tel. 977 553 707 - Fax 977 542 721

[arola@arolaeditors.com](mailto:arola@arolaeditors.com)

Cossetània Edicions: C. de la Violeta, 6 - 43800 Valls

Tel. 977 602 591 - Fax 977 614 357

[cossetania@cossetania.com](mailto:cossetania@cossetania.com)

Aquesta obra està subjecta a una llicència Attribution-NonCommercial-NoDerivs 3.0 Unported de Creative Commons. Per veure'n una còpia, visiteu <http://creativecommons.org/licenses/by-nc-nd/3.0/> o envieu una carta a Creative Commons, 171 Second Street, Suite 300, San Francisco, California 94105, USA.

This work is licensed under the Creative Commons Attribution-NonCommercial-NoDerivs 3.0 Unported License. To view a copy of this license, visit <http://creativecommons.org/licenses/by-nc-nd/3.0/> or send a letter to Creative Commons, 171 Second Street, Suite 300, San Francisco, California, 94105, USA.

# NETWORKS OF BIOINSPIRED PROCESSORS

Alfonso Ortega de la Puente  
Marina de la Cruz Echeandía

Editors



# NETWORKS OF BIOINSPIRED PROCESSORS

**Alfonso Ortega de la Puente** (Editor)

Departamento de Ingeniería Informática  
Escuela Politécnica Superior  
Universidad Autónoma de Madrid  
alfonso.ortega@uam.es

**Marina de la Cruz Echeandía** (Editor)

Departamento de Ingeniería Informática  
Escuela Politécnica Superior  
Universidad Autónoma de Madrid  
marina.cruz@uam.es

**Manuel Alfonseca**

Departamento de Ingeniería Informática  
Escuela Politécnica Superior  
Universidad Autónoma de Madrid  
manuel.alfonseca@uam.es

**Eloy Anguiano Rey**

Departamento de Ingeniería Informática  
Escuela Politécnica Superior  
Universidad Autónoma de Madrid  
eloy.anguiano@uam.es

**Fernando Arroyo Montoro**

Departamento Lenguajes, Proyectos y Sistemas Informáticos  
Escuela Universitaria de Informática  
Universidad Politécnica de Madrid  
farroyo@eui.upm.es

**Carlos Castañeda**

Departamento de Ingeniería Informática  
Escuela Politécnica Superior  
Universidad Autónoma de Madrid  
carlos.castanneda@uam.es



**Juan Castellanos**

Departamento de Inteligencia Artificial  
Universidad Politécnica de Madrid  
jcastellanos@fi.upm.es

**Miguel Cuéllar**

Departamento de Ingeniería Informática  
Escuela Politécnica Superior  
Universidad Autónoma de Madrid  
miguel.cuellar@uam.es

**Juan de Lara**

Departamento de Ingeniería Informática  
Escuela Politécnica Superior  
Universidad Autónoma de Madrid  
juan.delara@uam.es

**Emilio del Rosal**

Departamento de Ingeniería Informática  
Escuela Politécnica Superior  
Universidad Autónoma de Madrid  
emilio.delrosal@uam.es

**Antonio Jiménez Martínez**

Departamento de Ingeniería Informática  
Escuela Politécnica Superior  
Universidad Autónoma de Madrid  
antonio.jimenez@uam.es

**Robert Mercas**

GRLMC-Research Group on Mathematical Linguistics  
Universitat Rovira i Virgili  
robertgeorge.mercas@estudiants.urv.cat

**Victor Mitrana**

Departamento Lenguajes, Proyectos y Sistemas Informáticos  
Escuela Universitaria de Informática  
Universidad Politécnica de Madrid  
victor.mitrana@upm.es



**Carmen Navarrete Navarrete**

Departamento de Ingeniería Informática  
 Escuela Politécnica Superior  
 Universidad Autónoma de Madrid  
 carmen.navarrete@uam.es

**Rafael Nuñez Hervás**

Escuela Politécnica Superior  
 Universidad San Pablo CEU  
 rnhervas@ceu.es

**Diana Pérez**

Departamento de Ingeniería Informática  
 Escuela Politécnica Superior  
 Universidad Autónoma de Madrid  
 diana.perez@urjc.es

**Alexander Perekretenko**

GRLMC-Research Group on Mathematical Linguistics  
 Universitat Rovira i Virgili  
 alexander.perekrestenko@estudiants.urv.cat

**José Miguel Rojas Siles**

Departamento de Lenguajes y Sistemas Informáticos e Ingeniería de Software  
 Universidad Politécnica de Madrid  
 josemiguel.rojas@upm.es

**Eugenio Santos**

Departamento Lenguajes, Proyectos y Sistemas Informáticos  
 Escuela Universitaria de Informática  
 Universidad Politécnica de Madrid  
 esantos@eui.upm.es

**José M. Sempere**

Departamento de Sistemas Informáticos y Computación  
 Universitat Politècnica de València  
 jsempere@dsic.upv.es







---

# Contents

## Preface

*Alfonso Ortega de la Puente, Marina de la Cruz Echeandía* . . . . . XI

---

## Part I Models

---

### Networks of Bio-inspired Processors

<i>Fernando Arroyo Montoro, Juan Castellanos, Victor Mitrana, Eugenio Santos, José M. Sempere</i> . . . . .		3
1	Introduction . . . . .	3
2	Basic Definitions . . . . .	5
3	Three Variants of Accepting Networks of Evolutionary Processors . .	7
3.1	Computational power of [U]ANEP[FC]s . . . . .	10
4	Accepting Networks of Splicing Processors . . . . .	12
4.1	Computational power of ANSP[FC]s . . . . .	13
5	Problem Solving with [U]ANEP[FC]s/ANSP[FC]s . . . . .	14
6	Accepting Networks of Genetic Processors . . . . .	15
7	Towards an Unifying Model . . . . .	17
	References . . . . .	19

---

## Part II Tools

---

### Developing Tools for Networks of Processors

<i>Alfonso Ortega de la Puente, Marina de la Cruz Echeandía, Emilio del Rosal, Carmen Navarrete Navarrete, Antonio Jiménez Martínez, Juan de Lara, Eloy Anguiano Rey, Miguel Cuéllar, José Miguel Rojas Siles ..</i>		25
1	Motivation .....	25
2	Simulation of NEPs .....	27
	jNEP: a Java NEP simulator .....	27
	jNEPview: a graphical viewer for the simulations of jNEP .....	34
	First steps of the simulation of NEPs on massively parallel platforms .....	35
2.1	Hamiltonian path problem solution by NEPs .....	39
2.2	Family of graphs .....	40
2.3	Multithread platform architecture .....	41
2.4	Parallel platform architecture .....	41
2.5	Programming languages for NEPs .....	45
	NEPvl .....	45
	NEPsLingua .....	49
References .....		57

---

## Part III Applications

---

### NEPs Applied to Solve Specific Problems

<i>Alfonso Ortega de la Puente, Marina de la Cruz Echeandía, Emilio del Rosal, Rafael Nuñez Hervás, Antonio Jiménez Martínez, Carlos Castañeda, José Miguel Rojas Siles, Diana Pérez, Robert Mercas, Alexander Perekrestenko, Manuel Alfonseca .....</i>		63
1	Solving NP-problems with Lineally Bounded Resources .....	64
1.1	Solving the SAT problem with jNEP .....	64
1.2	Solving an instance of the Hamiltonian path problem with jNEP .....	69
1.3	Solving a graph coloring problem with jNEP .....	71
2	Some Applications of NEPs to Language Processing .....	76
2.1	PNEPs: top-down parsing for natural languages .....	76
2.2	PNEPs for shallow parsing .....	92
References .....		98



---

## Preface

Alfonso Ortega de la Puente and Marina de la Cruz Echeandía

Departamento de Ingeniería Informática  
Escuela Politécnica Superior  
Universidad Autónoma de Madrid  
E-mail: {alfonso.ortega, marina.cruz}@uam.es

This volume aims to provide a state-of-the-art of the work recently done, by some relevant Spanish Research Groups, in the area of nets of processors.

It could be interesting for a wide spectrum of audience: from mathematicians and linguists to computer scientists that are looking for efficient new models of computation to apply to their problems. There is actually no previous knowledge needed. In a first reading, formal definitions and results could be skipped. The rest of the volume is written to be independent and fully understandable.

The structure of this family of bio-inspired models of computation contains very simple nodes (processors) able to perform very simple operations on their contents connected in a predefined net topology. Although the structure and operations are very simple (formally, with a bounded expressive power), it is easy to find instances of the model equivalent to Turing machine. This circumstance allows considering nets of processors as general purpose computers to solve any computable task. Their intrinsic parallelism makes possible to write versions of algorithms for classical intractable problems that improve the (at least temporal) performance.

This volume is structured as follows: firstly, it introduces the reader in the formal definition and properties of the family, showing, for the first time, a possible full and integrated (Meta) model for the different kind of nets of

processors currently used. Then, it shows together (again for the first time) results about the design of tools to consider these models as computers (simulators, programming languages, and some examples of problems solved with them).



## Part I

---

## Models



---

# Networks of Bio-inspired Processors <sup>★</sup>

Fernando Arroyo Montoro<sup>1</sup>, Juan Castellanos<sup>2</sup>, Victor Mitrana<sup>1</sup>, Eugenio Santos<sup>1</sup>, José M. Sempere<sup>3</sup>

<sup>1</sup> Departamento Lenguajes, Proyectos y Sistemas Informáticos  
Escuela Universitaria de Informática  
Universidad Politécnica de Madrid  
Madrid, Spain  
E-mail: {farroyo,esantos}@eui.upm.es, victor.mitrana@upm.es

<sup>2</sup> Departamento de Inteligencia Artificial  
Universidad Politécnica de Madrid  
Madrid, Spain  
E-mail: jcastellanos@fi.upm.es

<sup>3</sup> Departamento de Sistemas Informáticos y Computación  
Universitat Politècnica de València  
Valencia, Spain  
E-mail: jsempere@dsic.upv.es

## 1 Introduction

The goal of this work is twofold. Firstly, we propose a uniform view of three types of accepting networks of bio-inspired processors: networks of evolutionary processors, networks of splicing processors and networks of genetic processors. And, secondly, we survey some features of these networks: computational

---

<sup>★</sup> Work partially supported by the Spanish Ministry of Science and Innovation under coordinated research project TIN2011-28260-C03-00 and research projects TIN2011-28260-C03-01, TIN2011-28260-C03-02 and TIN2011-28260-C03-03.



power, computational and descriptonal complexity, the existence of universal networks, efficiency as problem solvers and the relationships among them.

These networks are based on a rather common architecture for parallel and distributed symbolic processing, related to the Connection Machine [28] and the Logic Flow paradigm [24], and they consist of several processors, each of which is placed in a node in a virtual complete graph, which can handle data associated with the respective node. Each node processor acts on the local data in accordance with some predefined rules, and then the local data become mobile agents which can navigate in the network following a given protocol. Only that data which is able to pass a filtering process can be communicated. This filtering process may require that some conditions imposed by the sending processor be satisfied by the receiving processor or by both processors. All the nodes simultaneously send their data and the receiving nodes use a variety of strategies to handle, also simultaneously, all the arriving messages (see [25, 28]).

The general idea briefly presented above is modified here using a method inspired by cell biology. Each processor in a node is very simple, either an evolutionary, a splicing or a genetic processor. The three types of processors differ from each other by the operation they carry out.

By an evolutionary processor we mean a processor which can perform very simple operations: namely, point mutations in a DNA sequence (insertion, deletion or substitution of a pair of nucleotides). More generally, each node may be viewed as a cell that contains genetic information encoded in DNA sequences which may evolve by local evolutionary events: that is, point mutations. Each node is specialized for just one of these evolutionary operations.

By a splicing processor we mean a processor that can perform the splicing operation which is one of the basic mechanisms by which the DNA sequences are recombined under the effect of enzymatic activities.

By a genetic processor we mean a processor that can perform two different types of operations: either a pure mutation operation (i.e. the substitution operation in the evolutionary processors) or a full and massive crossover operation between strings which can be considered as a splicing operation with null contexts between strings.

Furthermore, the data in each node is organized in the form of multisets of words (each word appears 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.



A series of papers was devoted to different variants of this model viewed as language generating devices (see [2, 3, 4, 5, 6, 12, 13, 17, 19]). The paper [42] is an early survey in this area. Similar ideas may be found in other bio-inspired models: for example, *tissue-like membrane systems* [51] or models from Distributed Computing area like *parallel communicating grammar systems* [47].

## 2 Basic Definitions

We start by summarizing the notions used throughout the paper. An *alphabet* is a finite and nonempty set of symbols. The cardinality of a finite set  $A$  is written  $\text{card}(A)$ . 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  $\varepsilon$ . The length of a word  $x$  is denoted by  $|x|$  while  $\text{alph}(x)$  denotes the minimal alphabet  $W$  such that  $x \in W^*$ .

In the course of its evolution, the genome of an organism mutates by different processes. At the level of individual genes the evolution proceeds by local operations (point mutations) which substitute, insert and delete nucleotides of the DNA sequence. In what follows, we define some rewriting operations that will be referred to as *evolutionary operations* since they may be viewed as linguistic formulations of local gene mutations. We say that a rule  $a \rightarrow b$ , with  $a, b \in V \cup \{\varepsilon\}$  is a *substitution rule* if both  $a$  and  $b$  are not  $\varepsilon$ ; it is a *deletion rule* if  $a \neq \varepsilon$  and  $b = \varepsilon$ ; it is an *insertion rule* if  $a = \varepsilon$  and  $b \neq \varepsilon$ . The set of all substitution, deletion, and insertion rules over an alphabet  $V$  are denoted by  $\text{Sub}_V$ ,  $\text{Del}_V$ , and  $\text{Ins}_V$ , respectively.

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

- If  $\sigma \equiv a \rightarrow b \in \text{Sub}_V$ , then  $\sigma^*(w) = \begin{cases} \{ubv : \exists u, v \in V^* (w = uav)\}, \\ \{w\}, \text{ otherwise.} \end{cases}$

Note that a rule such as the one above is applied to all occurrences of the letter  $a$  in different copies of the word  $w$ . An implicit assumption is that arbitrarily many copies of  $w$  are available.



- If  $\sigma \equiv a \rightarrow \varepsilon \in Del_V$ , then  $\sigma^*(w) = \begin{cases} \{uv : \exists u, v \in V^* (w = uav)\}, \\ \{w\}, \text{ otherwise} \end{cases}$   
 $\sigma^r(w) = \begin{cases} \{u : w = ua\}, \\ \{w\}, \text{ otherwise} \end{cases}$   
 $\sigma^l(w) = \begin{cases} \{v : w = av\}, \\ \{w\}, \text{ otherwise} \end{cases}$
- If  $\sigma \equiv \varepsilon \rightarrow a \in Ins_V$ , then  $\sigma^*(w) = \{uav : \exists u, v \in V^* (w = uv)\}$ ,  
 $\sigma^r(w) = \{wa\}$ ,  $\sigma^l(w) = \{aw\}$ .

$\alpha \in \{*, l, r\}$  expresses the way a deletion or insertion rule is applied to a word, namely at any position ( $\alpha = *$ ), in the left ( $\alpha = l$ ), or in the right ( $\alpha = r$ ) end of the word, respectively. The note for the substitution operation mentioned above remains valid for insertion and deletion at any position. For every rule  $\sigma$ , action  $\alpha \in \{*, l, r\}$ , and  $L \subseteq V^*$ , we define the  $\alpha$ -action of  $\sigma$  on  $L$  by  $\sigma^\alpha(L) = \bigcup_{w \in L} \sigma^\alpha(w)$ . Given a finite set of rules  $M$ , we define the  $\alpha$ -action of  $M$  on the word  $w$  and the language  $L$  by:

$$M^\alpha(w) = \bigcup_{\sigma \in M} \sigma^\alpha(w) \quad \text{and} \quad M^\alpha(L) = \bigcup_{w \in L} M^\alpha(w),$$

respectively.

For two disjoint and nonempty subsets  $P$  and  $F$  of an alphabet  $V$  and a word  $z$  over  $V$ , we define the following two predicates

$$rc_s(z; P, F) \equiv P \subseteq alph(z) \wedge F \cap alph(z) = \emptyset$$

$$rc_w(z; P, F) \equiv alph(z) \cap P \neq \emptyset \wedge F \cap alph(z) = \emptyset.$$

The construction of these predicates is based on *context conditions* defined by the two sets  $P$  (*permitting contexts/symbols*) and  $F$  (*forbidding contexts/symbols*). Informally, both conditions require that no forbidding symbol be present in  $w$ ; furthermore the first condition requires all permitting symbols to appear in  $w$ , while the second one requires at least one permitting symbol to appear in  $w$ . It is plain that the first condition is stronger than the second one.

For every language  $L \subseteq V^*$  and  $\beta \in \{s, w\}$ , we define:

$$rc_\beta(L, P, F) = \{z \in L \mid rc_\beta(z; P, F)\}.$$

An *evolutionary processor over  $V$*  is a 5-tuple  $(M, PI, FI, PO, FO)$ , where:



– Either  $(M \subseteq Sub_V)$  or  $(M \subseteq Del_V)$  or  $(M \subseteq Ins_V)$ . The set  $M$  represents the set of evolutionary rules of the processor. As can be seen, a processor is “specialized” in one evolutionary operation only.

–  $PI, FI \subseteq V$  are the *input* permitting/forbidding contexts of the processor, while  $PO, FO \subseteq V$  are the *output* permitting/forbidding contexts of the processor (with  $PI \cap FI = \emptyset$  and  $PO \cap FO = \emptyset$ ).

An evolutionary processor such as the one above with  $PI = PO = P$  and  $FI = FO = F$  is called a *uniform evolutionary processor* and is defined as the triple  $(M, P, F)$ . We denote the set of (uniform) evolutionary processors over  $V$  by  $(U)EP_V$ . Clearly, the (uniform) evolutionary processor described here is a mathematical concept similar to that of an evolutionary algorithm, both being inspired by Darwinian evolution. As we have mentioned above, the rewriting operations we have considered might be interpreted as mutations and the filtering process described might be viewed as a selection process. Recombination is missing but evolutionary and functional relationships between genes can be captured by taking only local mutations into consideration [57]. However, another type of processor based on recombination only, called a splicing processor, has been the focus of a series of studies which will be surveyed in the sections below.

### 3 Three Variants of Accepting Networks of Evolutionary Processors

An *accepting network of evolutionary processors* (ANEP for short) is an 8-tuple  $\Gamma = (V, U, G, N, \alpha, \beta, x_I, x_O)$ , where:

- $V$  and  $U$  are the input and network alphabets, respectively,  $V \subseteq U$ .
- $G = (X_G, E_G)$  is an undirected graph without loops with the set of vertices  $X_G$  and the set of edges  $E_G$ .  $G$  is called the *underlying graph* of the network.
- $N : X_G \longrightarrow EP_U$  is a mapping which associates each node  $x \in X_G$  with the evolutionary processor  $N(x) = (M_x, PI_x, FI_x, PO_x, FO_x)$ .
- $\alpha : X_G \longrightarrow \{*, l, r\}$ ;  $\alpha(x)$  gives the action mode of the rules of node  $x$  on the words existing in that node.
- $\beta : X_G \longrightarrow \{s, w\}$  defines the type of the *input/output filters* of a node. More precisely, for every node,  $x \in X_G$ , the following filters are defined:



$$\begin{aligned} \text{input filter: } \rho_x(\cdot) &= rc_{\beta(x)}(\cdot; PI_x, FI_x), \\ \text{output filter: } \tau_x(\cdot) &= rc_{\beta(x)}(\cdot; PO_x, FO_x). \end{aligned}$$

That is,  $\rho_x(w)$  (resp.  $\tau_x$ ) indicates whether or not the word  $w$  can pass the input (resp. output) filter of  $x$ . Moreover,  $\rho_x(L)$  (resp.  $\tau_x(L)$ ) is the set of words of  $L$  that can pass the input (resp. output) filter of  $x$ .

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

An *Accepting Network of Uniform Evolutionary Processors* (UANEP for short) is an ANEP with uniform evolutionary processors only.

We say that  $\text{card}(X_G)$  is the size of  $\Gamma$ . If  $\alpha$  and  $\beta$  are constant functions, then the network is said to be *homogeneous*. In the theory of networks some types of underlying graphs are common (for example *rings*, *stars*, *grids*, etc.). In most of the cases considered here, we focus on *complete* networks (i.e., networks having a complete underlying graph). The last section is an exception, as we discuss an incomplete [U]ANEP that simulates a given ANEPFC (see the meaning of the abbreviation ANEPFC in the next subsection).

A *configuration* of an [U]ANEP  $\Gamma$  as above is a mapping  $C : X_G \longrightarrow 2^{V^*}$  which associates 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. Given a word  $w \in V^*$ , the initial configuration of  $\Gamma$  on  $w$  is defined by  $C_0^{(w)}(x_I) = \{w\}$  and  $C_0^{(w)}(x) = \emptyset$  for all  $x \in X_G - \{x_I\}$ .

When changed for an evolutionary step, each component  $C(x)$  of configuration  $C$  is changed in accordance with the set of evolutionary rules  $M_x$  associated with node  $x$  and the way the rules  $\alpha(x)$  are applied. Formally, we say that configuration  $C'$  is obtained in *one evolutionary step* from configuration  $C$ , written as  $C \Longrightarrow C'$ , iff

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

When changed for a communication step, each node processor  $x \in X_G$  sends one copy of each word it has which can pass the output filter of  $x$  to all the node processors connected to  $x$ . And it receives all the words sent by any node processor connected with  $x$  providing that they can pass its input filter. Formally, we say that configuration  $C'$  is obtained in *one communication step* from configuration  $C$ , written as  $C \vdash C'$ , iff

$$C'(x) = (C(x) - \tau_x(C(x))) \cup \bigcup_{\{x,y\} \in E_G} (\tau_y(C(y)) \cap \rho_x(C(y)))$$

for all  $x \in X_G$ . Note that words which leave a node are removed from that node. If they cannot pass the input filter of any node, they are lost.



A model closely related to that of ANEPs, introduced in [23] and further studied in [22, 31], is that of *accepting networks of evolutionary processors with filtered connections* (ANEPFCs for short). An ANEPFC may be viewed as an ANEP in which the filters are shifted from the nodes on the edges. Therefore, instead of having a filter at both ends of an edge in each direction, there is only one filter independently of the direction.

An ANEPFC is a 9-tuple

$$\Gamma = (V, U, G, \mathcal{R}, \mathcal{N}, \alpha, \beta, x_I, x_O),$$

where:

- $V, U, G = (X_G, E_G)$ , have the same meaning as for ANEP,
- $\mathcal{R} : X_G \longrightarrow 2^{Sub_U} \cup 2^{Del_U} \cup 2^{Ins_U}$  is a mapping which associates each node with *the set of evolutionary rules* that can be applied in that node. Note that each node is associated only with one type of evolutionary rules: namely, for every  $x \in X_G$  either  $\mathcal{R}(x) \subset Sub_U$  or  $\mathcal{R}(x) \subset Del_U$  or  $\mathcal{R}(x) \subset Ins_U$  holds.
- $\alpha : X_G \longrightarrow \{*, l, r\}$ ;  $\alpha(x)$  gives *the action mode of the rules* of node  $x$  on the words existing in that node.
- $\mathcal{N} : E_G \longrightarrow 2^U \times 2^U$  is a mapping which associates each edge  $e \in E_G$  with *the permitting and forbidding filters of that edge*; formally,  $\mathcal{N}(e) = (P_e, F_e)$ , with  $P_e \cap F_e = \emptyset$ .
- $\beta : E_G \longrightarrow \{s, w\}$  defines *the filter type of an edge*.
- $x_I, x_O \in X_G$  are *the input and the output node* of  $\Gamma$ , respectively.

Note that every ANEPFC can be immediately transformed into an equivalent ANEPFC with a complete underlying graph by adding the edges that are missing and associating with them filters that do not allow any words to pass. Note that such a simplification is not always possible for ANEPs.

A configuration of an ANEPFC is defined in the same way as the configuration of an ANEP (see above). An evolutionary step is also defined in the same way as above.

Otherwise, when changed for a communication step, in an ANEPFC, each node-processor  $x \in X_G$  sends one copy of each word it contains to every node-processor  $y$  connected to  $x$ , provided they can pass the filter of the edge between  $x$  and  $y$ . It keeps no copy of these words but receives all the words sent by any node processor  $z$  connected with  $x$  providing that they can pass the filter of the edge between  $x$  and  $z$ . In this case, no word is lost.

Let  $\Gamma$  be an [U]ANEP[FC], the computation of  $\Gamma$  on the input word  $w \in V^*$  is a sequence of configurations  $C_0^{(w)}, C_1^{(w)}, C_2^{(w)}, \dots$ , where  $C_0^{(w)}$  is



the initial configuration of  $\Gamma$  defined by  $C_0^{(w)}(x_I) = w$  and  $C_0^{(w)}(x) = \emptyset$  for all  $x \in X_G$ ,  $x \neq x_I$ ,  $C_{2i}^{(w)} \implies C_{2i+1}^{(w)}$  and  $C_{2i+1}^{(w)} \vdash C_{2i+2}^{(w)}$ , for all  $i \geq 0$ . Note that the configurations are changed by alternative evolutionary and communication steps. By the previous definitions, each configuration  $C_i^{(w)}$  is uniquely determined by configuration  $C_{i-1}^{(w)}$ .

A computation *halts* (and it is said to be *halting*) if one of the following two conditions holds:

(i) There exists a configuration in which the set of words existing in the output node  $x_O$  is non-empty. In this case, the computation is said to be an *accepting computation*.

(ii) There exist two identical configurations obtained either in consecutive evolutionary steps or in consecutive communication steps.

The *language accepted* by the [U]ANEP[FC]  $\Gamma$  is  $L_a(\Gamma) = \{w \in V^* \mid \text{the computation of } \Gamma \text{ on } w \text{ is an accepting one}\}$ . We denote by  $\mathcal{L}([U]ANEP[FC])$  the class of languages accepted by [U]ANEP[FC]s.

We say that an [U]ANEP[FC]  $\Gamma$  decides the language  $L \subseteq V^*$ , and write  $L(\Gamma) = L$  iff  $L_a(\Gamma) = L$  and the computation of  $\Gamma$  on every  $x \in V^*$  halts.

### 3.1 Computational power of [U]ANEP[FC]s

The results obtained so far ([40, 38, 22, 23]) state that non-deterministic Turing machines can be simulated by ANEPs and ANEPFCs.

Therefore we have:

**Theorem 1.** *Both  $\mathcal{L}(ANEP)$  and  $\mathcal{L}(ANEPFC)$  equal the class of recursively enumerable languages.*

It is clear that filters associated with each node of an ANEP allow the computation to be closely controlled. However, by moving the filters from the nodes to the edges, the possibility of controlling the computation seems to be diminished. For instance, data cannot be lost during the communication steps. In spite of this, we have seen that ANEPFCs are still computationally complete. This means that moving the filters from the nodes to the edges does not decrease the computational power of the model. Although the two variants are equivalent from the point of view of computational power, a direct proof would have been worthwhile. In [8] it was shown that the two models can efficiently simulate each other: namely, each computational step in one model is simulated in a constant number of computational steps in the other. This



is particularly useful when the solution of a problem needs to be translated from one model to the other. Note that a translation via a Turing machine, by the constructions shown in [40, 38, 22, 23] squares the time complexity of the new solution. A natural question arises: What is the computational power of UANEPs? The answer was given in [9] where the time complexity preserving simulation between ANEPs and ANEPFCs was extended to UANEPs. More precisely, it was shown that each pair of networks among the three variants efficiently simulates each other. Consequently, we can state the first main result of this section:

**Theorem 2.**

1. *Each class  $\mathcal{L}([U]ANEP[FC])$  equals the class of recursively enumerable languages.*
2. *Each pair of networks among the three variants efficiently simulates each other.*

These results can be improved by showing that each recursively enumerable language can be accepted by an ANEP[FC] of constant size. More precisely:

**Theorem 3.** [32, 31, 6]

1. *Every recursively enumerable language can be accepted by an ANEP of size 7.*
2. *Every recursively enumerable language can be accepted by an ANEPFC of size 16.*

The second result can be extended to characterize the class **NP**. Although the first result cannot be extended to a similar succinct characterization of **NP**, as the proof in [6] is based on the simulation of a phrase-structure grammar, such a succinct characterization of **NP** is proposed in [32, 31].

**Theorem 4.**

1. *A language is in **NP** if and only if it is accepted by an ANEP of size 10 in polynomial time.*
2. *A language is in **NP** if and only if it is accepted by an ANEPFC of size 16 in polynomial time.*

We do not know whether similar results like those in Theorems 3 or 4 holds for UANEPs.





## 4 Accepting Networks of Splicing Processors

In the case of Accepting Networks of Splicing Processors (ANSP for short), the point mutations associated with each node are replaced by the missing operation (recombination), which is present here in the form of splicing. This computing model is to some extent similar to the test tube distributed systems based on splicing introduced in [16] and further explored in [48]. However, there are several differences: first, the model proposed in [16] is a language generating mechanism while ours is an accepting one; second, we use a single splicing step, while every splicing step in [16] is actually an infinite process consisting of iterated splicing steps; third, each splicing step in our model is reflexive; fourth, the filters of our model are based on random context conditions while those in [16] are based on membership conditions; fifth, at every splicing step a set of auxiliary words, always the same and particular to every node, is available for splicing. Along the same lines, we should stress the differences between this model and the time-varying distributed H systems, a generative model introduced in [50] and further studied in [41, 49, 46]. The computing strategy of such a system is that the passing of words from a set of rules to another one is specified by a cycle. Only those words that are obtained at one splicing step by using a set of rules are passed in a circular way to the next set of rules. This means that words which cannot be spliced at some step disappear from the computation while words produced at different splicing steps cannot be spliced together. Now, the differences between time-varying distributed H systems and ANSPs are evident: each node of an ANSP has a set of auxiliary words, words obtained at different splicing steps in different nodes can be spliced together, words are not communicated in a circular way, since identical copies of the same word are sent out to all the nodes, the communication is controlled by filters.

A *splicing rule* over a finite alphabet  $V$  is a word of the form  $u_1\#u_2\$v_1\#v_2$  such that  $u_1, u_2, v_1$ , and  $v_2$  are in  $V^*$  and such that  $\$$  and  $\#$  are two symbols not in  $V$ .

For a splicing rule  $r = u_1\#u_2\$v_1\#v_2$  and for  $x, y, w, z \in V^*$ , we say that  $r$  produces  $(w, z)$  from  $(x, y)$  (denoted by  $(x, y) \vdash_r (w, z)$ ) if there exist some  $x_1, x_2, y_1, y_2 \in V^*$  such that  $x = x_1u_1u_2x_2$ ,  $y = y_1v_1v_2y_2$ ,  $z = x_1u_1v_2y_2$ , and  $w = y_1v_1u_2x_2$ .

For a language  $L$  over  $V$  and a set of splicing rules  $R$  we define

$$\sigma_R(L) = \{z, w \in V^* \mid (\exists u, v \in L, r \in R)[(u, v) \vdash_r (z, w)]\}.$$



A *splicing processor* over  $V$  is a 6-tuple  $(S, A, PI, FI, PO, FO)$ , where  $S$  a finite set of splicing rules over  $V$ ,  $A$  a finite set of auxiliary words over  $V$ , and all the other parameters have the same meaning as in the definition of evolutionary processors. Now an ANSP can be defined in the same way as an ANEP except that the processors associated with nodes are splicing processors.

A *configuration* of an ANSP  $\Gamma$  is a mapping  $C : X_G \rightarrow 2^{U^*}$  which associates a set of words to every node of the graph. By convention, the auxiliary words do not appear in any configuration.

There are two ways to change a configuration: by a splicing step or by a communication step. When a splicing step is used, each component  $C(x)$  of the configuration  $C$  is changed according to the set of splicing rules  $S_x$ , whereby the words in set  $A_x$  are available for splicing. Formally, configuration  $C'$  is obtained in one splicing step from configuration  $C$ , written as  $C \Rightarrow C'$ , iff for all  $x \in X_G$

$$C'(x) = \sigma_{S_x}(C(x) \cup A_x).$$

Since each word present in a node, as well as each auxiliary word, appears in an arbitrarily large number of identical copies, all possible splittings are assumed to be done in one splicing step. If the splicing step is defined as  $C \Rightarrow C'$ , iff

$$C'(x) = S_x(C(x), A_x) \text{ for all } x \in X_G,$$

then all processors of  $\Gamma$  are called *restricted* and  $\Gamma$  itself is said to be restricted.

A communication step and the language accepted/decided by an ANSP are defined in the same way as those for ANEP. The definitions of the complexity classes defined on ANEPs can be straightforwardly carried over ANSPs. On the other hand, accepting networks of splicing processors with filtered connections (ANSPFC) are defined similarly to ANEPFCs.

#### 4.1 Computational power of ANSP[FC]s

The main result in [37, 36] is:

**Theorem 5.**

1. Each recursively enumerable language  $L$  is accepted by a restricted ANSP of size 7.
2. Each NP language  $L$  is accepted by a restricted ANSP of size 7 in polynomial time.



We should point out that only the rules in the node input node depend on the language  $L$ , and the encoding that we use for its symbols while the parameters of the other nodes do not depend in any way on language  $L$ . If we allow all the parameters of the networks to depend on the given language, we have

**Theorem 6.** [30]

1. *All recursively enumerable languages are accepted by ANSPs of size 2.*
2. *All languages in **NP** can be accepted by ANSPs of size 3 working in polynomial time.*

Note that the ANSPs in the last theorem are not necessarily restricted. Since, by definition, ANSPs need at least two nodes to accept any non-trivial language, these results go a long way to settling this issue, although they do leave one problem unsolved: the efficient simulation of non-deterministic Turing machines by ANSPs with two nodes.

As far as the computational power of ANSPFCs is concerned, a complete characterization is reported in [14]:

**Theorem 7.**

1. *A language is recursively enumerable if and only if it is accepted by a restricted ANSPFC of size 4.*
2. *A language is in **NP** if and only if it is accepted by a restricted ANSPFC of size 4 in polynomial time.*

## 5 Problem Solving with [U]ANEP[FC]s/ANSP[FC]s

Although the results in the previous sections state that every problem in **NP** can be solved in polynomial time using different variants of accepting networks, the results are obtained by simulating a nondeterministic Turing machine; thus we still have to obtain a classic solution to a problem, and then translate it in terms of [U]ANEP[FC]s/ANSP[FC]s. To overcome this drawback, a series of papers discussed how [U]ANEP[FC]s and ANSP[FC]s can be viewed as problem solvers.

Recall that a possible correspondence between decision problems and languages can be made via an encoding function which transforms an instance of a given decision problem into a word (see, e.g., [26]). We say that a decision problem  $P$  is solved in time  $\mathcal{O}(f(n))$  by [U]ANEP[FC]s/ANSP[FC]s if there exists a family  $\mathcal{G}$  of [U]ANEP[FC]s/ANSP[FC]s such that the following conditions are satisfied:



1. The encoding function of any instance  $p$  of  $P$  with size  $n$  can be computed by a deterministic Turing machine in time  $\mathcal{O}(f(n))$ .
2. For each instance  $p$  of size  $n$  of the problem one can effectively construct, in time  $\mathcal{O}(f(n))$ , an  $[U]ANEP[FC]/ANSP[FC]$   $\Gamma(p) \in \mathcal{G}$  which decides, again in time  $\mathcal{O}(f(n))$ , the word encoding the given instance. This means that the word is decided if and only if the solution to the given instance of the problem is “YES”. This effective construction is called an  $\mathcal{O}(f(n))$  time solution to the problem.

If the  $[U]ANEP[FC]/ANSP[FC]$   $\Gamma \in \mathcal{G}$  constructed above decides the language of words encoding all instances of the same size  $n$ , then the construction of  $\Gamma$  is called a uniform solution. Intuitively, a solution is uniform if for problem size  $n$ , we can construct a unique  $[U]ANEP[FC]/ANSP[FC]$  that solves all instances of size  $n$  taking the (reasonable) encoding of instance as “input”.

The paper [34] proposes using ANEPs to provide uniform linear time solutions to the 3-CNF-SAT and Hamiltonian Path; in [38] a uniform linear solution to the Vertex-Cover problem is proposed. And [23] proposes another uniform linear time solution to the Vertex-Cover problem, solved this time by ANEPFCs. Uniform linear time solutions to the SAT and Hamiltonian Path problems with ANSPs and ANSPFCs are discussed in [33].

## 6 Accepting Networks of Genetic Processors

The third case that we refer to in this work is the Accepting Networks of Genetic Processors (ANGP). Here, there are two sources of inspiration: the classical paradigm of Genetic Algorithms and Evolutionary Computation [43], and the models of Evolutionary or Splicing processors mentioned above. A genetic processor can perform one of the following two operations: (1) Mutation between symbols (here, the substitution operation in the evolutionary processors can be considered), and (2) Pure and massive crossover (which can be considered as the splicing operation by taking empty contexts). Observe that both operations were considered in the past as the main ingredients of genetic algorithms. Despite this, ANGP differs from classical Genetic Algorithms in two aspects: first, ANGP consists of a finite number of processors that run in parallel independently, so they should be considered as a full parallel scheme for genetic algorithms [1]; and second, the model is an acceptance model not an optimization one (like genetic algorithms). Nevertheless, ANGP could be modified to tackle optimization problems instead of acceptance ones.



For any alphabet  $V$ , the *mutation* rules take the form  $a \rightarrow b$ , with  $a, b \in V$ , and they can be applied over the string  $xay$  to produce the new string  $xbay$ . The *crossover operation* is defined as follows: Let  $x$  and  $y$  be two strings, then  $x \bowtie y = \{x_1y_2, y_1x_2 : x = x_1x_2 \text{ and } y = y_1y_2\}$ . Observe that  $x, y \in x \bowtie y$  given that we can take  $\varepsilon$  to be a part of  $x$  or  $y$ . In addition, the crossover operation can be extended over languages in the usual form.

A *genetic processor* over  $V$  is a tuple  $(M_R, A, PI, FI, PO, FO, \alpha, \beta)$ , where  $M_R$  is a finite set of mutation rules over  $V$ ,  $A$  is a multiset of strings over  $V$  with a finite support and an arbitrary large number of copies of every string,  $PI, FI \subset V^*$  are the input permitting/forbidding contexts,  $PO, FO \subset V^*$  are the output permitting/forbidding contexts,  $\alpha \in \{1, 2\}$  defines the working mode with the following values

- If  $\alpha = 1$  the processor applies mutation rules
- If  $\alpha = 2$  the processor applies crossover rules and  $M_R = \emptyset$

and  $\beta \in \{(s), (w)\}$  defines the type of the input/output filters of the processor. Here,  $s$  means the strong predicate  $rc_s(\cdot; P, F)$  as defined in the evolutionary case, and  $w$  the weak predicate denoted by  $rc_w(\cdot; P, F)$ . Nevertheless, given that  $P, F \subset V$ , the previous predicates will be defined over the segments of a given string instead of its symbols.

An Accepting Network of Genetic Processors is defined as in the previous models of ANEPs and ANSPs. The acceptance criterion, the configuration of the network and the alternation between communication steps and *genetic* steps are defined as in the previous models.

With respect to the completeness of the ANGP model, we have the following result.

**Theorem 8.** [10] *Every recursively enumerable language can be accepted by an ANGP.*

The proof of the previous result is approached in a non-uniform manner. Hence, one can construct in polynomial time an ANGP that simulates the computation of an arbitrary Turing machine with an arbitrary input string (no matter its length). Given that the previous simulation works in polynomial time depending on the length of the input string (provided that we take into account the number of genetic and communication steps), the following result comes easily by simulating a nondeterministic Turing machine.



**Theorem 9.** [10] *Every language in NP can be accepted/decided in polynomial time by an ANGP.*

Observe that no results have been obtained to define the description complexity of this model. Nevertheless, a formal proof that 16 genetic processors are sufficient to generate any recursively enumerable language is provided in [11]. So, it is expected that further results of the descriptive complexity of ANGPs will be provided shortly.

## 7 Towards an Unifying Model

We have presented three different models of Accepting Networks of Bio-inspired processors. They have common characteristics and features that point to a model which can be formally defined. They share the following aspects and, probably, new models will be formulated in the near future:

1. A finite set of processors that apply operations over strings which have been inspired by biomolecular functions and operations in nature. The processors work with a multiset of strings.
2. A connection topology between processors in the form of a network.
3. A set of (input/output) filters which can be attached to the processors or to the connections.

A biologically inspired processor with filters, over an alphabet  $V$ , can be defined as the tuple  $(op, PI, FI, PO, FO)$ , where  $op$  is a biologically inspired operation over strings and the rest of the elements have been defined in the evolutionary processors.

The following table shows some of the operations that we have defined in this study and others which can be used instead of the operations that have been defined previously.



<i>insertion</i>	Insert a symbol into a string
<i>deletion</i>	Delete a symbol from a string
<i>substitution (mutation)</i>	Substitute a symbol into a string
<i>splicing</i>	Splicing rules
<i>crossover</i>	Full massive splicing with empty context
<i>hairpin completion</i>	Hairpin completion from folded strings [15, 52]
<i>superposition</i>	Complementarity completion from double stranded strings [7]
<i>loop and double loop recombination</i>	DNA recombination based on gene assembly [54]
<i>inversion, duplication and transposition</i>	DNA fragments modification as operations over substrings [29, 18]

**Table 1: Some operations which can be inserted into biologically inspired processors**

Once we have introduced a generalization of previously defined processors, an Accepting Network of Bio-inspired Processors (ANBP) can be defined as the tuple  $\Gamma = (V, U, G, N, \alpha, \beta, x_I, x_O)$ , where the difference with respect to ANEPs is that the function  $N$  associates a biologically inspired processor to every vertex in the connection graph.

Here, we describe a new framework that should be studied in depth. In particular the following questions should be addressed:

- Some of the operations shown in Table 1, do not have computational completeness (i.e. they do not characterize recursively enumerable languages). We can combine some of these operations by inserting them into different processors. It is natural to ask whether computational completeness could be achieved for some combinations of operations, and what the minimal combination is to achieve it.
- Filtered connections have been proposed for ANEPs while other models consider only filtered processors. The transformation of filtered processors into filtered connections should be explored in the different combinations of operations. Furthermore, we could provide a pure hybrid network where different types of filters (connections or processors) work together.



## References

1. Alba, E., Troya, J.M. (1999). A survey of parallel distributed genetic algorithms. *Complexity*, 4(4), 31–52.
2. Alhazov, A., Rogozhin, Y. (2008). About Precise Characterization of Languages Generated by Hybrid Networks of Evolutionary Processors with One Node. *it Computer Science Journal of Moldova* 16(3), 364–376.
3. Alhazov, A., Csuhaj-Varjú, E., Martín-Vide, C., Rogozhin, Y. (2008). About Universal Hybrid Networks of Evolutionary Processors of Small Size. In *Language and Automata Theory and Applications (LATA 2008)*, LNCS 5196, 28–39.
4. Alhazov, A., Martín-Vide, C., Truthe, B., Dassow, J., Rogozhin, Y. (2009). On Networks of Evolutionary Processors with Nodes of Two Types. *Fundam. Inform.* 91(1), 1–15.
5. Alhazov, A., Bel Enguix, G., Rogozhin, Y. (2009). Obligatory Hybrid Networks of Evolutionary Processors. In *International Conference on Agents and Artificial Intelligence (ICAART 2009)*, 613–618.
6. Alhazov, A., Csuhaj-Varjú, E., Martín-Vide, C., Rogozhin, Y. (2009). On the Size of Computationally Complete Hybrid Networks of Evolutionary Processors. *Theoretical Computer Science* 410, 3188–3197.
7. Bottoni, P., Labella, A., Manca, V., Mitrana, V. (2004). Superposition Based on Watson-Crick-Like Complementarity *Theory of Computing Systems* 39, 503–524.
8. Bottoni, P., Labella, A., Manea, F., Mitrana, V., Sempere, J.M. (2009). Filter Position in Networks of Evolutionary Processors Does Not Matter: A Direct Proof. In *International Meeting on DNA Computing and Molecular Programming (DNA 15)*, LNCS 5877, 1–11.
9. Bottoni, P., Labella, A., Manea, F., Mitrana, V., Petre, I., Sempere, J.M. (2010). Complexity-Preserving Simulations Among Three Variants of Accepting Networks of Evolutionary Processors *Natural Computing*, in press.
10. Campos, M., Sempere, J.M. (2011). Accepting Networks of Genetic Processors are computationally complete. (*submitted*)
11. Campos, M., Sempere, J.M. (2011). Descriptive Complexity of Generating Networks of Genetic Processors. (*submitted*)
12. Castellanos, J., Martín-Vide, C., Mitrana, V., Sempere, J.M. (2003). Networks of Evolutionary Processors. *Acta Informatica* 39, 517–529.
13. Castellanos, J., Leupold, P., Mitrana, V. (2005). On the Size Complexity of Hybrid Networks of Evolutionary Processors. *Theoretical Computer Science* 330 (2), 205–220.
14. Castellanos, J., Manea, M., Mingo López, L.F., Mitrana, V. (2007). Accepting Networks of Splicing Processors with Filtered Connections. In *Machines, Computations, and Universality (MCU 2007)*, LNCS 4664, 218–229.
15. Cheptea, D., Martín-Vide, C., V. Mitrana, V. (2006) A new operation on words suggested by DNA biochemistry: Hairpin completion, in: *Proc. Transgressive Computing*, 216–228.





16. Csuhaj-Varjú, E., Kari, L., Păun, G. (1996). Test Tube Distributed Systems Based on Splicing. *Computers and AI*, 15, 211–232.
17. Csuhaj-Varjú, E., Martín-Vide, C., Mitrana, V. (2005). Hybrid Networks of Evolutionary Processors are Computationally Complete. *Acta Informatica* 41, 257–272.
18. Dassow, J., Mitrana, V., Salomaa, A. (2002). Operations and language generating devices suggested by the genome evolution *Theoretical Computer Science* 270, 701–738.
19. Dassow, J., Truthe, B. (2007). On the Power of Networks of Evolutionary Processors. In *Machines, Computations, and Universality (MCU 2007)*, LNCS 4667, 158–169.
20. Dassow, J., Mitrana, V. (2008). Accepting Networks of Non-Inserting Evolutionary Processors, In *Proceedings of NCGT 2008: Workshop on Natural Computing and Graph Transformations*, 29–42.
21. Dassow, J., Mitrana, V., Truthe, B. (2008). The Role of Evolutionary Operations in Accepting Networks of Evolutionary Processors. Submitted.
22. Drăgoi, C., Manea, F. (2008). On the Descriptive Complexity of Accepting Networks of Evolutionary Processors with Filtered Connections. *International Journal of Foundations of Computer Science*, 19:5, 1113–1132.
23. Drăgoi, C., Manea, F., Mitrana, V. (2007). Accepting Networks of Evolutionary Processors With Filtered Connections. *Journal of Universal Computer Science*, 13:11, 1598–1614.
24. Errico, L., Jesshope, C. (1994). Towards a New Architecture for Symbolic Processing, In *Artificial Intelligence and Information-Control Systems of Robots '94*, 31–40.
25. Fahlman, S. E., Hinton, G.E., Sejnowski, T.J. (1983). Massively Parallel Architectures for AI: NETL, THISTLE and Boltzmann Machines, In *Proc. of the National Conference on Artificial Intelligence*, 109–113.
26. Garey, M., Johnson, D. (1979). *Computers and Intractability: A Guide to the Theory of NP-completeness*, San Francisco, CA: W. H. Freeman.
27. Hartmanis, J., Stearns, R.E. (1965). On the Computational Complexity of Algorithms, *Trans. Amer. Math. Soc.*, 117, 533–546.
28. Hillis, W.D. (1979). *The Connection Machine*. MIT Press, Cambridge.
29. Leupold, P., Mitrana, V., Sempere, J.M. (2004). Formal Languages Arising from Gene Repeated Duplication, In *Aspects of Molecular Computing LNCS 2950*, 297–308.
30. Loos, R., Manea, F., Mitrana, V. (2009). On Small, Reduced, and Fast Universal Accepting Networks of Splicing Processors, *Theoretical Computer Science* 410:4–5, 417–425.
31. Loos, R., Manea, F., Mitrana, V. (2009). Small Universal Accepting Networks of Evolutionary Processors with Filtered Connections, In *Proceedings of the Descriptive Complexity of Formal Systems Workshop, EPTCS* 3, 173–183.



32. Loos, R., Manea, F., Mitrana, V. (2009). Small Universal Accepting Networks of Evolutionary Processors, submitted.
33. Manea, F., Martín-Vide, C., Mitrana, V. (2005). Accepting Networks of Splicing Processors. In *Computability in Europe (CiE 2005)*, LNCS 3526, 300–309.
34. Manea, F., Martín-Vide, C., Mitrana, V. (2005). Solving 3CNF-SAT and HPP in Linear Time Using WWW, In *Machines, Computations and Universality*, LNCS 3354, 269–280.
35. Manea, F., Martín-Vide, C., Mitrana, V. (2006). On the Size Complexity of Universal Accepting Hybrid Networks of Evolutionary Processors, in *Proceedings of the First International Workshop on Developments in Computational Models, ENTCS 135*, 95–105.
36. Manea, F., Martín-Vide, C., Mitrana, M. (2006). All NP-Problems Can Be Solved in Polynomial Time by Accepting Networks of Splicing Processors of Constant Size. *International Meeting on DNA Computing, DNA12*, LNCS 4287, 47–57.
37. Manea, F., Martín-Vide, C., Mitrana, V. (2007). Accepting Networks of Splicing Processors: Complexity Results, *Theoretical Computer Science*, 371:1-2, 72–82.
38. Manea, F., Martín-Vide, C., Mitrana, V. (2007). On the Size Complexity of Universal Accepting Hybrid Networks of Evolutionary Processors, *Mathematical Structures in Computer Science*, 17:4, 753–771.
39. Manea, F., Mitrana, V. (2007). All NP-problems Can Be Solved in Polynomial Time by Accepting Hybrid Networks of Evolutionary Processors of Constant Size, *Information Processing Letters*, 103:3, 112 – 118.
40. Manea, F., Margenstern, M., Mitrana, V., Pérez-Jiménez, M. J. (2008). A New Characterization of NP, P, and PSPACE With Accepting Hybrid Networks of Evolutionary Processors, in press *Theory of Computing Systems*, doi:10.1007/s00224-008-9124-z.
41. Margenstern, M., Rogozhin, Y. (2001). Time-varying Distributed H Systems of Degree 1 Generate All Recursively Enumerable Languages. In *Words, Semigroups, and Transductions* World Scientific Publishing, Singapore, 329–340.
42. Martín-Vide, C., Mitrana, V. (2005). Networks of Evolutionary Processors: Results and Perspectives, In *Molecular Computational Models: Unconventional Approaches*, 78–114.
43. Michalewicz, Z. (1996). Genetic Algorithms + Data Structures = Evolution Programs, Springer.
44. Minsky, M.L., Size and Structure of Universal Turing Machines Using Tag Systems. In: *Recursive Function Theory, Symp. in Pure Mathematics 5*, 229–238.
45. Papadimitriou C.H. (1994). *Computational Complexity*, Addison-Wesley.
46. Păun, A. (1999). On Time-varying H Systems. *Bulletin of the EATCS*, 67, 157–164.
47. Păun, G., Sântean, L. (1989). Parallel Communicating Grammar Systems: The Regular Case, *Annals of University of Bucharest, Ser. Matematica-Informatica* 38, 55–63.



48. Păun, G. (1998). Distributed Architectures in DNA Computing Based on Splicing: Limiting the Size of Components. In *Unconventional Models of Computation*, Springer-Verlag, Berlin, 323–335.
49. Păun, G. (1996). Regular Extended H Systems are Computationally Universal. *J. Automata, languages, Combinatorics*, 1(1), 27–36.
50. Păun, G. (1997). DNA Computing; Distributed Splicing Systems. In *Structures in Logic and Computer Science, LNCS 1261*, Springer-Verlag, Berlin, 351–370.
51. Păun, G. (2000). Computing with Membranes, *Journal of Computer and System Sciences* 61, 108–143.
52. Păun G., Rozenberg, G., Yokomori, T. (2001). Hairpin languages. *International Journal of Foundations of Computer Science* 12, 837–847.
53. Post, E.L. (1943). Formal Reductions of the General Combinatorial Decision Problem, *Amer. J. Math.* 65, 197–215.
54. Prescott, D.M., Ehrenfeucht, A., Rozenberg, G. (2001). Molecular operations for DNA processing in hypotrichous ciliates. *European Journal of Protistology* 37, 241–260.
55. Rogozhin, Y. (1996). Small Universal Turing Machines, *Theoretical Computer Science* 168, 215–240.
56. Rozenberg, G., Salomaa, A., eds. (1997). *Handbook of Formal Languages*, vol. I–III, Springer-Verlag, Berlin.
57. Sankoff, D., et al. (1992). Gene Order Comparisons for Phylogenetic Inference: Evolution of the Mitochondrial Genome, In *Proceedings of the National Academy of Sciences of the United States of America* 89, 6575–6579.



## Part II

---

### Tools



---

# Developing Tools for Networks of Processors <sup>★</sup>

Alfonso Ortega de la Puente<sup>1</sup>, Marina de la Cruz Echeandía<sup>1</sup>, Emilio del Rosal<sup>1</sup>, Carmen Navarrete Navarrete<sup>1</sup>, Antonio Jiménez Martínez<sup>1</sup>, Juan de Lara<sup>1</sup>, Eloy Anguiano Rey<sup>1</sup>, Miguel Cuéllar<sup>1</sup>, and José Miguel Rojas Siles<sup>2</sup>

<sup>1</sup> Departamento de Ingeniería Informática

Escuela Politécnica Superior

Universidad Autónoma de Madrid

Madrid, Spain

E-mail: {alfonso.ortega, marina.cruz, emilio.delrosal,  
carmen.navarrete, antonio.jimenez, juan.delara, eloy.anguiano,  
miguel.cuellar}@uam.es

<sup>2</sup> Departamento de Lenguajes y Sistemas Informáticos e Ingeniería de Software

Universidad Politécnica de Madrid

Madrid, Spain

E-mail: josemiguel.rojas@upm.es

## 1 Motivation

A great deal of research effort is currently being made in the realm of so called “natural computing”. “Natural computing” mainly focuses on the definition, formal description, analysis, simulation and programming of new models of computation (usually with the same expressive power as Turing Machines) inspired by Nature, which makes them particularly suitable for the simulation of complex systems.

---

<sup>★</sup> Work partially supported by the Spanish Ministry of Science and Innovation under coordinated research project TIN2011-28260-C03-00 and research projects TIN2011-28260-C03-01, TIN2011-28260-C03-02 and TIN2011-28260-C03-03.

Some of the best known natural computers are Lindenmayer systems (L-systems, a kind of grammar with parallel derivation), cellular automata, DNA computing, genetic and evolutionary algorithms, multi agent systems, artificial neural networks, P-systems (computation inspired by membranes) and NEPs (or networks of evolutionary processors). This chapter is devoted to this last model.

There are two main areas in which these models could be useful: as new architectures for computers, other than von Neumann's machine; and as modelling tools to simulate complex systems for which "conventional approaches" (usually based on differential equations) are, in practice, difficult to handle.

Two steps are needed in both scenarios:

1. design a particular instance of the model able to solve the task under study (this step is equivalent to "programming" the model) and
2. "run" the model.

Several attempts have been made to build hardware devices to support these bio-inspired models. Some research groups are currently implementing *in silico* the basic components of P-systems [19]. [38] describes other examples of hardware implementations of cellular automata, CAM-6 and its derivatives, that have been used for the simulation of complex systems (see [36]). But, unfortunately there are no real computers for almost all bio-inspired models. So, step 2 usually involves simulating the model in a "conventional" (von Neumann) computer.

Informally, and assuming that NP (*nondeterministic polynomial time*)  $\neq$  P, NP is a complexity that includes those problems whose solution by means of algorithms run on conventional computers requires *more than polynomial* time. We can informally understand *more than polynomial* as *exponential*. One of the most interesting features of these bio-inspired computers is their intrinsic parallelism. We can design algorithms for them that could improve the exponential performance of their *classic* versions. Nevertheless, when the models have to be simulated on conventional computers, the total amount of space needed to simulate the model and to actually run the algorithm usually becomes exponential. This may be one of the main reasons why natural computers are not widely used to solve real problems. Most of the simulators are not able to handle the size of non trivial problems. Grid, cloud computation and clusters offer an interesting and promising option to overcome the drawbacks of both solutions: "specific" hardware, and simulators run on von Neumann's machines.



There are several research groups interested in programming tools for natural computers. These tools include textual and visual programming languages, compilers, sequential and parallel simulators.

P-Lingua ([21] and <http://www.p-lingua.org>) is a programming language for membrane computing which aims to be a standard to define P systems. One of its main characteristics is to remain as close as possible to the formal notation used in the literature to define P systems. Once he has formalized his P systems, the programmer does not need any additional effort to describe them with P-Lingua. P-Lingua is also the name of a software package that includes several built-in simulators for each supported model, as well as the compilers needed to simulate P-Lingua programs.

One of the current topics of interest of the authors of this chapter is the development of programming tools for NEPs, which will be briefly described in the following paragraphs.

This chapter is structured as follows:

1. We describe our approaches to simulate NEPs:
  - jNEP, a Java multitreaded NEPs simulator
  - The simulation of NEPs on massively parallel platforms
2. We describe some graphical tools for designing NEPs:
  - We describe our graphical viewer for the simulation of jNEP (jNEPView)
  - We also describe our visual programming language for NEPs (NEPsVL)
3. With respect to other tools for designing NEPs, we introduce NEPsLingua, our textual language for NEPs inspired by P-Lingua.

We should point out that this chapter brings together some work previously published earlier. All the references are placed in the corresponding section.

## 2 Simulation of NEPs

### jNEP: a Java NEP simulator

Current research on NEPs focuses mainly on the definition of different families and on the study of their formal properties, such as their computational completeness and their ability to solve NP problems with polynomial performance. However, apart from [26], little effort has been made to develop a NEP simulator for any kind of implementation. Unfortunately, this software hardly





restricts the general model because it only allows one kind of rules and filters and, what is more important, violates two of the main principles of the model:

1. NEP's computation should not be deterministic and
2. Evolutionary and communication steps should alternate strictly.

In addition, the software focuses on solving decision problems in a parallel way, rather than on providing the researchers with a general simulator for any kind of NEPs.

jNEP tries to fill this gap in the literature. It is a program written in Java which can simulate almost any NEP in the literature. In order to be a valuable tool for the scientific community, it has been developed under the following principles:

- a) It rigorously complies with the formal definitions found in the literature.
- b) It serves as a general tool, by allowing the use of the different NEP variants and is ready to adapt to future possible variants, as the research in the area advances.
- c) It exploits as much as possible the inherent parallel/distributed nature of NEPs.

The jNEP code is freely available in <http://jnep.e-delrosal.net>.

### *jNEP design*

jNEP provides an implementation of NEPs as general, flexible and rigorous as has been described in the previous paragraphs. As shown in figure 1, the design of the NEP class mimics the NEP model definition. In jNEP, a NEP is composed of evolutionary processors and an underlying graph (attribute *edges*) to define the net topology and the allowed inter processor interactions. The *NEP* class coordinates the main dynamic of the computation and rules the processors (instances of the *EvolutionaryProcessor* class), forcing them to perform alternate evolutionary and communication steps. It also stops the computation when needed. The core of the model includes these two classes, together with the *Word* class, which handles the manipulation of words and their symbols.

We keep *jNEP* as general and rigorous as possible by means of the following mechanisms: Java interfaces and different versions to widely exploit the parallelism available in the hardware platform.

*jNEP* offers three interfaces:



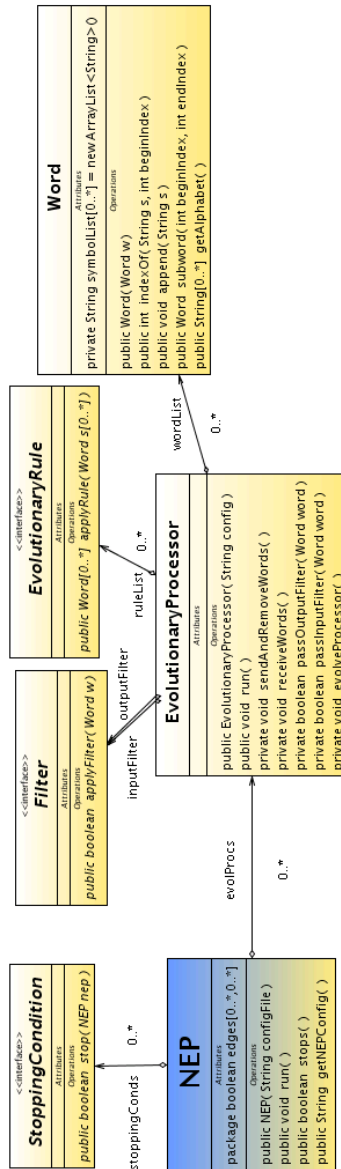


Fig. 1. Simplified class diagram of jNEP



- a) *StoppingCondition*, which provides the method *stop* to determine whether a *NEP* object should stop according to its state.
- b) *Filter*, whose method *applyFilter* determines which objects of class *Word* can pass.
- c) *EvolutionaryRule*, which applies a *Rule* to a set of *Words* to get a new set.

*jNEP* tries to implement a wide set of NEPs' features. The *jNEP user guide* (<http://jnep.e-delrosal.net>) contains the updated list of filters, evolutionary rules and stopping conditions implemented.

Currently *jNEP* has two lists of choices to select the parallel/distributed platform on which it runs (any combination of them is also available in <http://jnep.e-delrosal.net>). Concurrency is implemented by means of two different Java approaches: *Threads* and *Processes*. The first needs more complex synchronization mechanisms. The second uses heavier concurrent threads. The supported platforms are standard JVM and clusters of computers (by means of *JavaParty*).

More precisely, in the case of the *Processes* option each processor in the net is actually an independent program in the operating system. The communication between nodes is carried out through the standard input/output streams of the program. The class *NEP* has access to those streams and coordinates the nodes. The mandatory alternation of communication and evolutionary steps in the computations of NEPs greatly eases their synchronization and communication. The following protocol has been followed for the communication step:

- 1 *NEP* class sends the message to communicate to every node in the graph. Then it waits for responses.
- 2 All node finish their communication step after sending the words that pass their outputs filters. Then, they indicate to the *NEP* class that they have finished the communication step.
- 3 The *NEP* class moves all the words from the net to the input filters of the corresponding nodes.

The evolutionary step is synchronized by means of an initial message sent by the *NEP* class to make all the nodes evolve. Afterwards, the *NEP* class waits until all the nodes finish.

The implementation with *Java Threads* has other implications. In this option, each processor is an object of the *Java Thread* class. Thus, each processor execute its tasks in parallel as independent lines, although they all belong to the same program. Data exchange between them is performed by



direct access to memory. The principles of communication and coordination are the same as in the previous option. The main difference is that, instead of waiting for all the streams to finish or to send a certain message, *Threads* are coordinated by means of basic concurrent programming mechanisms as semaphores, monitors, etc.

In conclusion, jNEP is a very flexible tool that can run in many different environments. Depending on the operating system, the Java Virtual Machine used and the concurrency option chosen, jNEP will work in a slightly different manner. Users should select the best combination for his needs.

Nevertheless, the peculiarities of Java (the JVM can be considered an intermediate layer of middleware between the source code and the operating system) makes it difficult to adjust all the details of the parallel simulation. This is why we have decided to explore other approaches that will be shown in the following sections.

### *Using jNEP*

jNEP is written in Java therefore to run jNEP one needs a Java virtual machine (version 1.4.2 or above) installed in a computer. Then one has to write a configuration file describing the NEP. The *jNEP user guide* (available at <http://jnep.e-delrosal.net>) contains the details concerning the commands and requirements needed to launch jNEP. In this section, we focus on the configuration file which has to be written before running the program, since it has some complex aspects important to be aware of the potentials and possibilities of jNEP.

The configuration file is an XML file specifying all the features of the NEP. Its syntax is described below in BNF format, together with a few explanations. Since BNF grammars are not capable of expressing context-dependent aspects, context-dependent features are not described here. Most of them have been explained informally in the previous sections. Note that the traditional characters `<>` used to identify non-terminals in BNF have been replaced by `[]` to prevent confusion with the use of the `<>` characters in the XML format.

- [configFile] ::= <?xml version="1.0"?> <NEP nodes="[integer]"> [alphabetTag] [graphTag] [processorsTag] [stoppingConditionsTag] </NEP>
- [alphabetTag] ::= <ALPHABET symbols="[symbolList]"/>
- [graphTag] ::= <GRAPH> [edge] </GRAPH>
- [edge] ::= <EDGE vertex1="[integer]" vertex2="[integer]"/> [edge]
- [edge] ::=  $\lambda$
- [processorsTag] ::= <EVOLUTIONARY\_PROCESSORS> [nodeTag] </EVOLUTIONARY\_PROCESSORS>



The above rules show the main structure of the NEP: the alphabet, the graph (specified through its vertices) and the processors. It is worth remembering that each processor is identified implicitly by its position in the processors tag (first is number 0, second is number 1, and so on).

```
- [stoppingConditionsTag] ::= <STOPPING_CONDITION> [conditionTag]
  </STOPPING_CONDITION>
- [conditionTag] ::= <CONDITION type="MaximumStepsStoppingCondition" maximum="[integer]" />
  [conditionTag]
- [conditionTag] ::= <CONDITION type="WordsDisappearStoppingCondition" words="[wordList]" />
  [conditionTag]
- [conditionTag] ::= <CONDITION type="ConsecutiveConfigStoppingCondition" /> [condition-
  Tag]
- [conditionTag] ::= <CONDITION type="NonEmptyNodeStoppingCondition" nodeID="[integer]" />
  [conditionTag]
- [conditionTag] ::=  $\lambda$ 
```

The syntax of the stopping conditions shows that a NEP can have several stopping conditions. The first one which is met causes the NEP to stop. The different types try to cover most of the stopping conditions used in the literature. If needed, more of them can be added to the system.

At this moment jNEP supports 4 stopping conditions, the *jNEP user guide* explains their semantics in detail:

1. **ConsecutiveConfigStoppingCondition:** It stops the NEP if nothing changes after two consecutive complete configurations (a communication and an evolutionary step).
2. **MaximumStepsStoppingCondition:** The NEP stops after a maximum number of steps.
3. **WordsDisappearStoppingCondition:** It stops the NEP if none of the words specified are in the NEP. It is useful for generative NEPs where the lack of non-terminals means that the computation have reached its goal.
4. **NonEmptyNodeStoppingCondition:** The NEP stops if one of the nodes is non-empty. Useful for NEPs with an output node.

```
- [nodeTag] ::= <NODE initCond="[wordList]" [auxWordList]> [evolutionaryRulesTag] [node-
  FiltersTag] </NODE> [nodeTag]
- [nodeTag] ::=  $\lambda$ 
- [auxWordList] ::= auxiliaryWords="[wordList]" |  $\lambda$ 
- [evolutionaryRulesTag] ::= <EVOLUTIONARY_RULES> [ruleTag] </EVOLUTIONARY_RULES>
- [ruleTag] ::= < RULE ruleType="[ruleType]" actionType="[actionType]" symbol="[symbol]"
  newSymbol="[symbol]" /> [ruleTag]
- [ruleTag] ::= < RULE ruleType="splicing" wordX="[symbolList]" wordY="[symbolList]"
  wordU="[symbolList]" wordV="[symbolList]" /> [ruleTag]
- [ruleTag] ::= < RULE ruleType="splicingChoudhary" wordX="[symbolList]" wordY="[sym-
  bolList]" wordU="[symbolList]" wordV="[symbolList]" /> [ruleTag]
- [ruleTag] ::=  $\lambda$ 
- [ruleType] ::= insertion | deletion | substitution
```



```

- [actionType] ::= LEFT | RIGHT | ANY
- [nodeFiltersTag] ::= [inputFilterTag] [outputFilterTag]
- [nodeFiltersTag] ::= [inputFilterTag]
- [nodeFiltersTag] ::= [outputFilterTag]
- [nodeFiltersTag] ::=  $\lambda$ 
- [inputFilterTag] ::= <INPUT [filterSpec]/>
- [outputFilterTag] ::= <OUTPUT [filterSpec]/>
- [filterSpec] ::= type=[filterType] permittingContext="[symbolList]"
    forbiddingContext="[symbolList]"
- [filterSpec] ::= type="SetMembershipFilter" wordSet="[wordList]"
- [filterSpec] ::= type="RegularLangMembershipFilter" regularExpression="[regExpression]"
- [filterType] ::= 1 | 2 | 3 | 4

```

Above, we describe the elements of the processors: their initial conditions, rules, and filters. jNEP treats rules with the same philosophy as in the case of stopping conditions, that is, our system supports almost all kinds found in the literature at the moment and, more important, future types can also be added.

jNEP can work with any of the rules found in the original model [6, 20, 7]. Moreover, we support splicing rules, which are needed to simulate an extension of the original model presented in [8] and [12]. The two splicing rule types are slightly different. It is important to note that if you use Manea's splicing rules, you may need to create an auxiliary word set for those processors with splicing rules.

With respect to filters, jNEP is prepared to simulate nodes with filters based on random context conditions. To be more specific, jNEP supports any of the four filter types traditionally used in the literature since [30]. Besides, jNEP is capable of creating filters based on membership conditions. They are used in such studies as [6]. They are to some extent non-standard and could be defined as follows:

1. **SetMembershipFilter**: It allows only words that are included in a specific set to pass.
2. **RegularLangMembershipFilter**: This filter contains a regular language to which words need to belong. The language has to be defined as a Java regular expression.

We will finish the explanation of the grammar for our xml files with the rules needed to describe some of the pending non-terminals. They are typical constructs for lists of words, list of symbols, boolean and integer data and regular expressions.

```

- [wordList] ::= [symbolList] [wordList]
- [wordList] ::=  $\lambda$ 
- [symbolList] ::= a string of symbols separated by the character '_'

```



```
- [boolean] ::= true | false
- [integer] ::= an integer number
- [regExpression] ::= a Java regular expression
```

The reader may refer to the *jNEP user guide* for further detailed information.

## **jNEPview: a graphical viewer for the simulations of jNEP**

jNEP has been improved with several visualization facilities. jNEPView display the network topology in a friendly manner and shows the complete description of the simulation state in each step. This tool makes it easier to program and study NEPs, which are quite complex, facilitating theoretical and practical advances on the NEP model.

In the following paragraphs we will describe the features of jNEP we have used to implement this graphic viewer, we will also discuss the design of jNEPView, and finally we will show some examples.

### *jNEP logging system*

jNEP produces a sequence of log files while it is running, one for each simulation step. This sequence of files will be read by jNEPView to show the successive configurations of the NEP. These logs are in a very simple format that contains a line for each processor in the same implicit order in which they appear in the configuration file. Each line contains the strings of the corresponding processor. This little extension of jNEP makes it simple to follow the trace of the simulation and manage it.

### *jNEPView design*

To handle and visualize graphs, we have used JGraphT [2] and JGraph [4] which are free Java libraries under the terms of the GNU Lesser General Public License.

JGraphT provides mathematical graph-theory objects and algorithms. It is used by jNEPView to formally represent the NEP underlying graph. Fortunately, JGraphT can also display its graphs using the JGraph library, which is graph visualization library with many utilities.

We use those libraries to show the NEP topology. Once jNEPView is started, a window shows the NEP layout as clear as possible. We have decided to set the nodes in a circle, but the user can freely move each component. In this way, it is easier to interpret the NEP and study its dynamics.



Moreover, several action buttons have been placed to study the NEP state and progress. If the user clicks on a node, a window is open where the words of the node appear. In order to control the simulation development, the user can move throughout the simulation and the contents of the selected nodes are updated in their corresponding windows in a synchronize way.

Before running jNEPView, jNEP should have actually finished the simulation. In this way, jNEPView just reads the jNEP state logs and the user can jump from one simulation step to another, without worrying about the simulation execution times.

### *jNEPView example*

This section describes how jNEPView shows the execution of a NEP solving a particular case of the Hamiltonian path in an undirected graph. This NEP is described in detail in [16] and in the chapter of this publication devoted to some application of NEPs. The jNEP package, that you can freely download from the web, includes the configuration XML file of this NEP.

Firstly, the user has to select the configuration file for jNEP which defines the NEP to simulate. After that, the layout of the NEP is shown as in figure 2.

At this point, the buttons placed in the main window to handle the simulation are activated and the user can select the nodes whose content is to be inspected during the simulation. Besides, the program allows the user to move throughout the simulation timeline by stepping forward and backward. Figures 3 to 6 display the contents of all the nodes in the NEP at three different moments: the three first steps and the final one. The user can also jump to a given simulation step by clicking on the appropriate button.

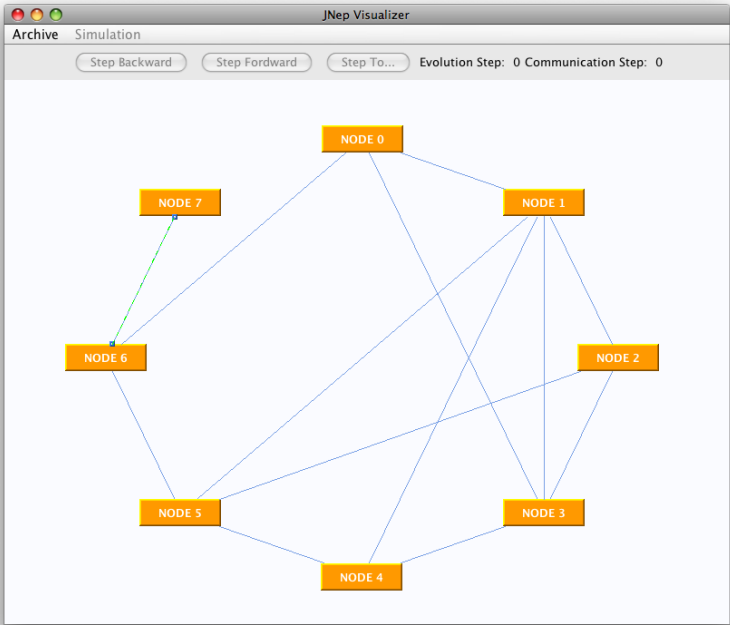
## **First steps of the simulation of NEPs on massively parallel platforms**

### *Introduction to parallel computing*

Parallel computing is a form of computation in which many calculations are carried out simultaneously (by means of multiple processing elements) to solve a problem. The problem is broken up into independent parts (subdomains or partitions) so that each processing element can execute its part of the algorithm simultaneously with the others.



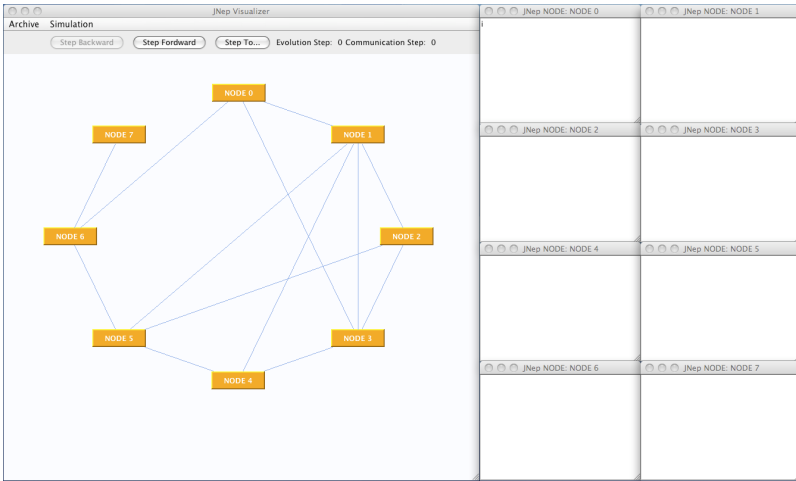




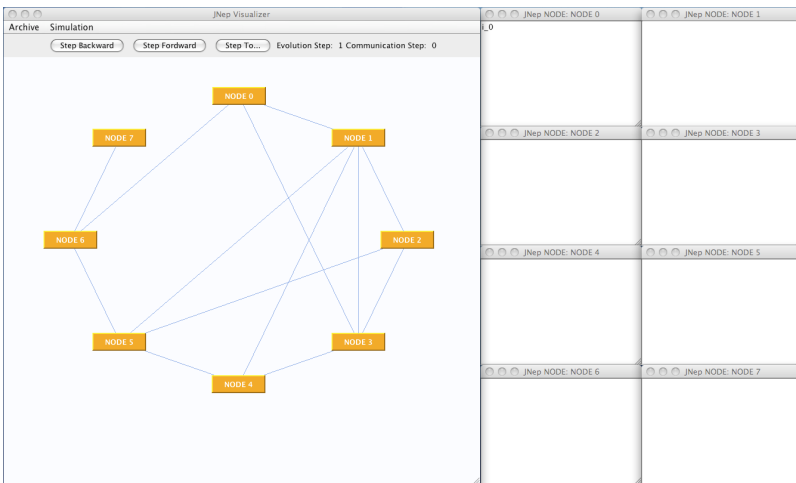
**Fig. 2.** Window that shows the layout of the simulated NEP

Clusters of computers are a popular way of accessing to massively parallel platforms. This is the case of the current work.

Perhaps the most popular general approach to parallel algorithms is the master/slave type of organization. In these multiple-tier applications, a single node (or more) organizes and disseminates the relatively separate tasks of the overall composite problem, and (optionally) collects and/or reassembles the individual results into a single integrated answer or product. The class of nodes actually receiving and processing the smaller component tasks represent another specialized tier of this hierarchical approach. More than two tiers of organization are also possible. A single tier of “slaves”, all simultaneously running serial code with absolutely no inter-communication, can be viewed as a specialized form of this approach. But two levels of organization, often with a single “master” node, is the most common configuration. Strategies for providing and optimizing load-balancing across multiple slave nodes within



**Fig. 3.** Initial simulation step



**Fig. 4.** Next simulation step

heterogeneous parallel environments is of general significance across a wide array of problems.

Because communication and synchronization between the different sub-tasks and nodes are typically one of the greatest obstacles to good parallel



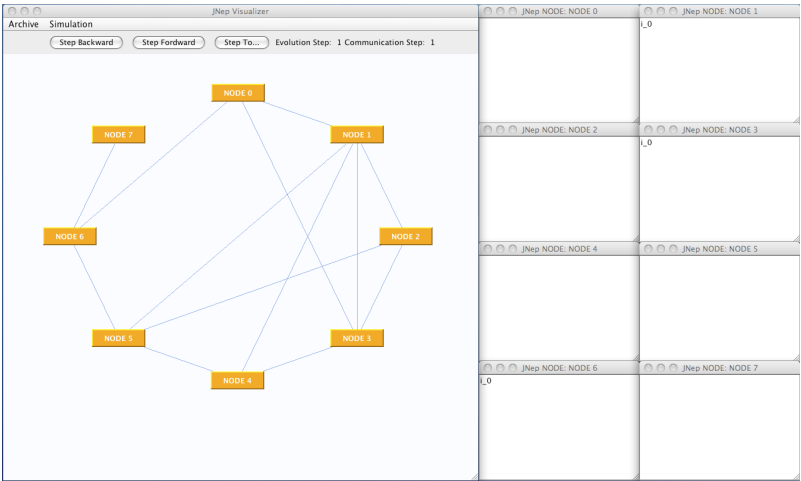


Fig. 5. Second simulation step

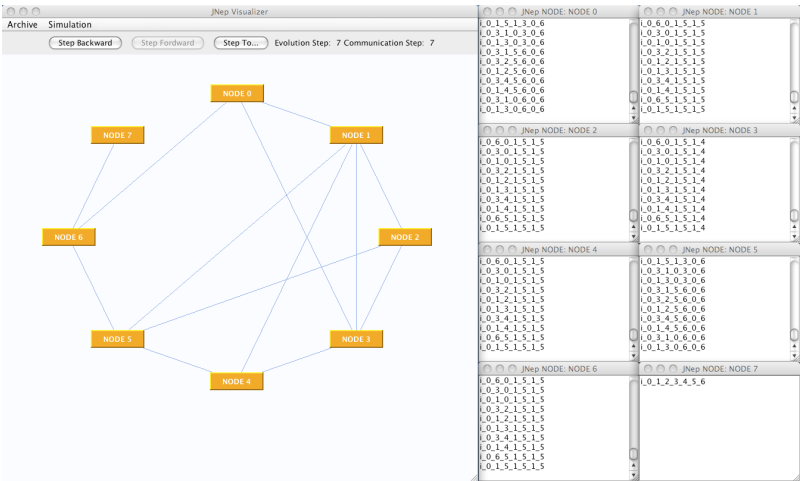


Fig. 6. End of simulation

programm performance, parallel computer programs and algorithms are more difficult to implement than sequential ones. But good management of the communications and synchronization is not sufficient in itself to achieve the



best performance of the parallel algorithm; the loadbalancing and the domain decomposition techniques also have a large role to play.

The most noteworthy idea of parallel computing is to decompose the problem into subproblems that are easier to solve; that is, the *Divide and conquer* philosophy. But, it should be borne in mind that a better or worse performance, and therefore the better use of resources, will depend on the solution taken to decompose the problem into at least as many domains as processes [25]; choosing an inappropriate domain decomposition will affect the speed-up of the parallel solution but the domain decomposition depends on both the problem that we want to execute in the cluster and its symmetries. Thus, our goal is to develop a generic platform to execute existing sequential codes, so that the parameters that optimize the application performance in the cluster (such as network and data topologies or domain decomposition, etc.) will be dynamically obtained while the algorithm is being executed. In general, the domains decomposition algorithm must take into account the problem properties and symmetries and must change them if the speed-up decreases.

Although we have used this framework to run NEPs in parallel, the framework is not limited to this kind of application.

### *Methodology*

In order to test the performance of clusters of computers when they are run in parallel NEPs we have designed a family of graphs to solve several instances of the Hamiltonian Path problem (HPP). [16] shows how the HPP can be solved by means of NEPs with a lineal (temporal) performance. Although our goal is not to reach this bound, this proof will give useful hints on how to improve the performance of the simulation of NEPs on non parallel hardware platforms.

## **2.1 Hamiltonian path problem solution by NEPs**

This well-known NP-complete problem searches an undirected graph for a Hamiltonian path, that is, one that visits each vertex exactly once.

This problem can be solved by means of the following NEP:

- The NEP graph is very similar to the one studied: an extra node is added to ease the definition of the stopping condition.
- Let  $n$  be the number of nodes of the graph under consideration (see figure 7).



- Let  $\{v_i, 0 \leq i \leq n\}$  be the set of processors of the NEP.
- The set  $\{i, 0, 1, \dots, n\}$  is used as the alphabet. Symbol  $i$  is the initial content of the initial node ( $v_0$ ). Each node (except the final one) adds its number to the string received from the network.
- Input and output filters are defined to allow the communication of all the strings that have not yet visited the node.
- The input filter of the final node excludes any string which is not a solution.
- It is easy to imagine a regular expression for the set of solutions (those words with the proper length, the proper initial and final node and where each node appears only once). The NEP basic model allows filters to be defined by means of regular expressions.

## 2.2 Family of graphs

Our goal is to check the cluster performance when solving the HPP for graphs of increasing difficulty. We have used a family of graphs with  $n$  nodes. 0 is the label of the initial node. Each node is connected with the four closest nodes. That is, node  $i$  is connected with the set of nodes  $\{i-2, i-1, i+1, i+2\}$ . There is a special case. When defining the NEP to solve this instance, we have to add the output node. The highest label is given to this node ( $n+1$ ). The output node is only connected with the final node of the graph under consideration ( $n$ ). The other connections of the output node are removed.

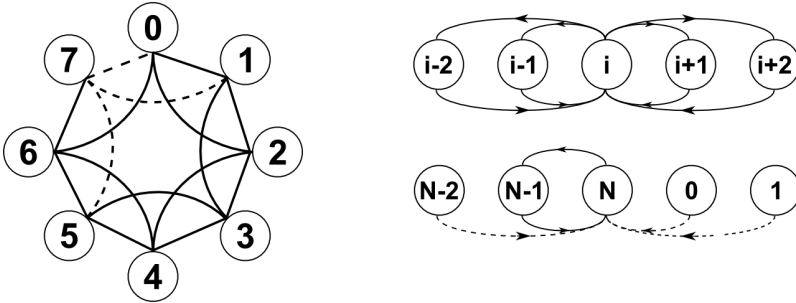


Fig. 7. Example of a NEP with  $n=6$  and the extra one to collect the strings

Figure 7 shows this circumstance and the graph for  $n = 6$ .

These pages compare two approaches that our research group has used to run NEPs on parallel platforms:



1. A multithreaded simulator for desktop computers (possibly parallel)
2. A massively parallel architecture (clusters of computers)

### 2.3 Multithread platform architecture

As we have previously explained, jNEP is a multithreaded Java simulator for NEPs. That is, it could actually be run in parallel if the underlying system is able to distribute the threads among different processors. We have performed a set of experiments in a multicore desktop computer with these characteristics.

The standard Java Virtual Machine is not designed to be run on clusters of computers. To run multithread Java applications on clusters a specific extension must be used. Most of these extensions migrate the threads on the clusters by means of RMI (Remote Method Invocation). In this study we have used JavaParty [3]. This is why it is difficult to compare jNEP with other frequently used libraries to handle parallel code on clusters. We just summarize the results of jNEP and compare them with other implementations.

### 2.4 Parallel platform architecture

We can consider this platform to be a framework that works as both, master and slave; it can also execute sequential code in a cluster, taking advantage of the workload and dynamic domain decomposition concepts, without rewriting the code (NEPs in our case) to adapt it to this parallel platform. This framework is implemented in ANSI C++, uses the MPI-II ([5]) extensions and follows the master-slave model.

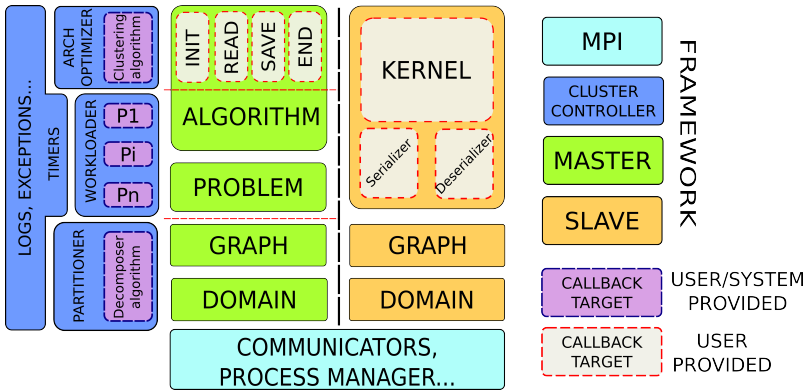
Any problem that will be solved on the cluster, can be modeled as a weighted and directed graph  $G_a$ , denoted by  $G_a(T, D, \omega)$ ;  $T$  denotes a set of vertices of the graph that represents the tasks to be done;  $D$  represents a finite set of edges of the graph; each vertex has a computation weight  $\omega$  that represents the number of computations required by the task to accomplish one step of the algorithm. The existence of an edge between vertex A and vertex B means that, to calculate the value of A at a certain instant in the execution, we need the value of B at the previous step of the algorithm. We say that A has a data dependency on B.

The framework (see figure 8) can be divided into 4 modules:

- Cluster controller: the module that handles the cluster and controls the communication between the master process and the plugins specified by the users. These plugins configure the behaviour of the framework to adapt it to the algorithm it will run on the cluster.



- Master-side procedures: the master creates the data structures, balances the workload  $\omega$  among the different nodes of the cluster (loadbalancing policy), takes care of the domain decomposition (to break the problem into independent domains) and reassembles the results sent by the slave processes. Communication with the user is always through this process.
- Slave-side procedures: slaves execute the sequential algorithm over the received domain as if they were not part of the cluster. Once the calculations have been made, they send the results back to the master process.
- Communication layer: implemented as a layer over Message Passing Interface (MPI), API specification that allows processes to communicate with another one or with any group of processes by sending and receiving messages.



**Fig. 8.** Framework architecture implemented to run sequential code as parallel in a cluster of computers.

Both master and slave processes work with graph structures so, the master translates the information given by the user into a graph and decomposes it into several domains that are sent to the slaves. The slaves receive the domains and translate them again into graphs. To transfer the information through the net, both processes must be able to serialize and deserialize the information of the graph, that is, binarize the user defined data structure that allocates the data to each vertex of the graph.

The kernel method allows the user to execute its specific algorithm on the slave process. Users do not have to worry about the communication and syn-



chronization with the master process and they know neither how many slaves have joined the simulation/resolution nor which loadbalancing and partition algorithm is used.

The behaviour of the cluster (cluster configuration and loadbalancing policies) and of the problem (problem configuration and domain decomposition method) are modelled by means of plugins. The system provides several plugins that can be replaced by the users with their own code.

## Results

We have performed two sets of experiments: on a conventional multicore architecture and on a massively parallel platform.

For the first set of experiments we used a multithreaded multicore platform (a desktop computer running a Linux kernel 2.6.26, with 16Gb of memory and  $4 \star 6$  cores Intel(R) Xeon(R) CPU E7450 2.40GHz) running a Java multithreaded simulator for NEPs, developed by our research group. The jNEP platform succeeded in solving graphs up to 8 nodes whereas the biggest graph solved by our parallel framework had 24 nodes.

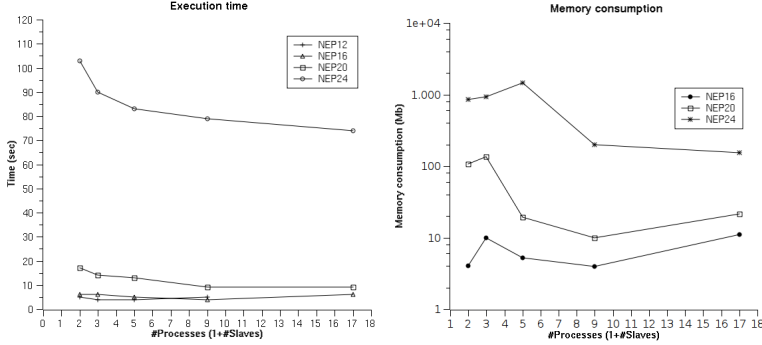
The results for the second set, were obtained by executing a sequential NEP kernel in a parallel environment ([1] HLRB II, 9728 cores, 4Gb memory per core) using the framework described. The simulation has been executed with NEPs of different numbers of nodes, from  $n = 16$  (more or less  $4 \times 10^3$  valid strings) to  $n = 24$  ( $5 \times 10^5$  strings). From  $n = 28$  and higher values, the assigned resources reached the limit. To observe the framework behaviour the number of slaves was changed, from  $2^0$  (equivalent to a single processor) to  $2^4$ . It is not possible to have  $2^5$  or more slaves, because this exceeds the number of vertexes of the NEP. This is the reason for our limited testbench.

Time(s)	Processes					Mem.(Mb)	Processes				
n	2	3	5	9	17	n	2	3	5	9	17
NEP12	5	4	4	5	-	NEP12	-	-	-	-	-
NEP16	6	6	5	4	6	NEP16	4	9.8	5.2	3.9	11
NEP20	17	14	13	9	9	NEP20	106.7	135.5	19.3	9.8	21.3
NEP24	103	90	83	79	74	NEP24	842.9	930.4	1445.5	200.1	153.6

**Table 1.** Execution time vs. number of processes and memory consumption vs. number of processes.







**Fig. 9.** a) Execution time running a sequential NEP algorithm in parallel using the framework described. The performance of the application was better with the NEP algorithm than with the single slave execution. b) Semilog plot for the memory consumption running the NEP algorithm under the framework.

From the point of view of the execution time, the performance of the algorithm is no worse when the framework (see table 1 and fig. 9). It can also be observed that the execution time decreases until a certain value, that depends on the number of processors and on the dimension of the problem, has been reached. Once this point has been exceeded, if the number of processors is still increasing, the execution time will start growing again, just because the master spends more time on the management of the communication, processes and domains than the slaves on the real calculus of the problem.

From the point of view of the memory consumption, behaviour is similar (see table 1); there is an optimal value of memory that depends on the number of processes and on the number of nodes of the NEP. Once this point has been reached, if the number of slaves is increased, the amount of memory needed to solve the HPP will also increase. As long as more slaves join the simulation, the number of domains will grow lineally and therefore, to fulfill the data dependencies between domains, the information will be more and more replicated among the cluster (i.e more memory to allocate the network buffers). On the other hand, increasing the number of domains involves increasing of the number of frames sent by the slaves to the master process. In summary, increasing the number of slaves leads to increasing the number of dataframes and the size of each one.



## 2.5 Programming languages for NEPs

### NEPv1

#### *Introduction to Domain Specific Visual Languages and AToM<sup>3</sup>*

Visual Languages play a central role in many computer science activities. For example, in software engineering, diagrams are widely used in most phases of software construction. They provide intuitive and powerful domain-specific constructs and make it possible to abstract from low-level, *accidental* details, enabling reasoning and improving understandability and maintenance. The term Domain Specific Visual Language (DSVL) [24] refers to languages that are especially oriented to a certain domain, limited but extremely efficient for the task to be performed. DSVLs are extensively used in Model Driven Development, one of the current approaches to Software Engineering. In this way, engineers no longer have to resort to low-level languages and programming, but are able to synthesize code for the final application from high-level, visual models. This increases productivity, and quality, and means that it can be used by non-programmers.

Designing a DSVL involves defining its concepts and the relations between them. This is called the abstract syntax, and is usually defined through a meta-model. Meta-models are normally described through UML class diagrams. Hence, the language spawned by the meta-model is the (possibly infinite) set of models conformant to it. In addition, a DSVL needs to be provided with a concrete syntax. That is, a visualization of the concepts defined in the meta-model. In the simplest case, the concrete syntax just assigns icons to meta-model classes and arrows to associations. The description of the abstract and concrete syntax is enough to generate a graphical modelling environment for the DSVL. Many tools are available that automate this task, and in this chapter we describe AToM<sup>3</sup> [15].

In many scenarios, the description of the DSVL syntax is not enough: manipulations need to be defined that “*breathe life*” into such models. For example, the models can be animated or simulated, “macros” defined for complex editing commands, or code generators built for further processing by other tools. As models and meta-models can be described as attributed, typed graphs, they can be visually manipulated by means of graph transformation techniques [18]. This is a declarative, visual and formal approach to manipulate graphs. Its formal basis, developed in the last 30 years, makes it possible to demonstrate properties of the transformations. A graph grammar is made



of a set of rules and a starting graph. Graph grammar rules consist of a left and a right hand side (LHS and RHS), each with graphs. When a rule is applied to a graph (called host graph), an occurrence of the LHS should be found in the graph, and then it can be replaced by the RHS.

In this chapter, we describe our efforts to apply the aforementioned concepts to build a DSL to design Networks of Evolving Processors (NEPs). For this purpose, we built a meta-model in the ATOM<sup>3</sup> tool and a graphical modelling environment was automatically generated. Then, this environment was enriched by providing rules to automate complex editing commands, and a code generator to synthesize code for jNEPs, in order to perform simulations. The approach has the advantage that the final user does not need to be proficient in the jNEP textual input language, but he can model and simulate NEPs visually.

### *NEPs visual language*

*Designer's viewpoint: how to define the metamodel for NEPs and, thus, the visual language* The system consists of four parts. Two of them are core components and the rest can be considered as tools for increasing the usability of the final system.

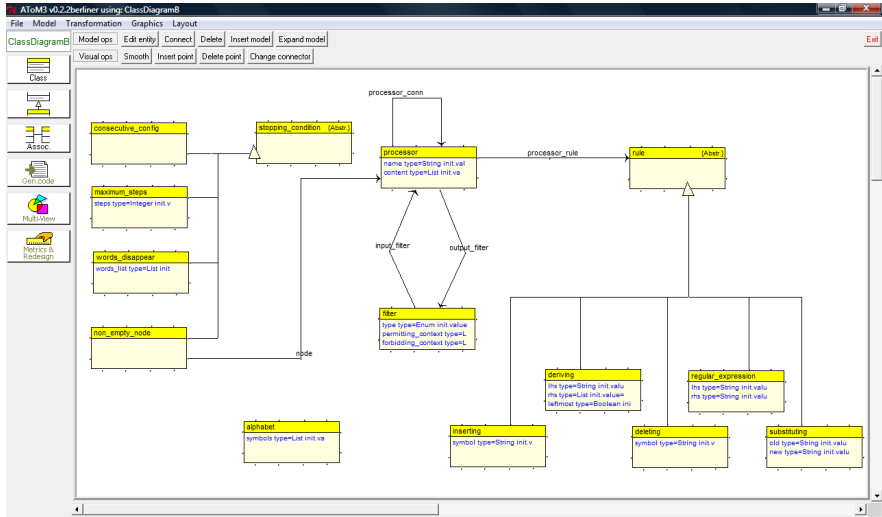
- *Core components*
  - **The meta-model**, which provides the designer with the elements needed to build models.
  - **The code generator**, a program that automatically writes the code used as input by the simulator.
- *Tools that increase usability*
  - **The constraints** included in the meta-model that ensure the syntactic (and possibly semantic) correctness of the models defined.
  - **Graph grammars**. Some graph transformation rules can be specified to automatically modify the models (which are actually ATOM<sup>3</sup> graphs) because several typical transformations might become dull and time-consuming if done manually.

The final programmer just draws his NEP on a canvas of the main window by means of buttons and other typical GUI components. Some special buttons trigger the checker and the code generator, they, finally, they start the execution of the simulator that uses the generated file. The user gets the result of the simulation without taking into account all the low level details of the complete process.



In the following paragraphs, the different components of the system are described with more detail.

Fig. 10 shows the **UML class diagram** of the meta-model that represents the NEP domain for the simulator. We can see several classes for the usual elements of a NEP: alphabet, processors, filters, rules, and stopping conditions. It also shows these subclasses:



**Fig. 10.** The meta-model UML class diagram

- Different rules (found in the literature):
  - inserting rules,
  - deleting rules,
  - substituting rules (replace a symbol),
  - deriving rules (change a symbol by a string),
  - rules that match regular expressions (splicing rules)
- Different stopping conditions (the name used in the diagram is highlighted)
  - *consecutive\_config*, the system stops when no change is detected;
  - *maximum\_steps*, it stops after a given number of steps;
  - *words\_disappear*, when some specific words disappear;
  - *non\_empty\_node*, when something enters a specific node by the first time .



**The code generator** is a set of Python routines responsible for creating the XML file that will be the input for the simulator (jNEP in this case). The algorithm of the code generator follows two steps:

- *Correctness test.* The code generator checks the following properties: there must be exactly one alphabet and one stopping condition; all the symbols in the model have to be contained in the alphabet, and there is a maximum of zero or one connection between each pair of processors.
- *Code generation.* The NEP being programmed is internally represented as a graph between instances of the classes defined in the metamodel. The edges of this graph follow the relationship of the metamodel. After checking the correctness of the model, and only if there is no mistake, the code generator goes across the graph of the model translating each instance and each relation into the corresponding XML code.

**Graph grammars** We have identified two specific tasks that could become boring and time-consuming if a NEP is designed manually. We have decided to automatically implement these tasks by means of graph grammars:

- To create the input and output filters of each processor.
- To create a complete graph among the processors.

In both cases, the final programmer will only push the button associated with the corresponding action.

In *AToM*<sup>3</sup>, each component in the UML diagram of the metamodel is enriched with the graphical representation by means of which the final programmer will draw this component on the canvas of the final system. These graphical representations actually describe the (graphic) basic syntax of the visual language. We have used for NEPv1 the following representations for the main components:

- Big rectangles, for alphabets.
- Small rectangles, for stopping conditions.
- Triangles, for filters.
- Ovals, for rules.
- Those attributes whose values are strings of characters are represented by means of texts.

Figure 11 shows some examples of these graphical representations by means of part of a NEP that contains the alphabet and two processors with their filters; each processor has a rule (P1 has a deleting rule and P2 has an inserting one) and they form a complete graph.



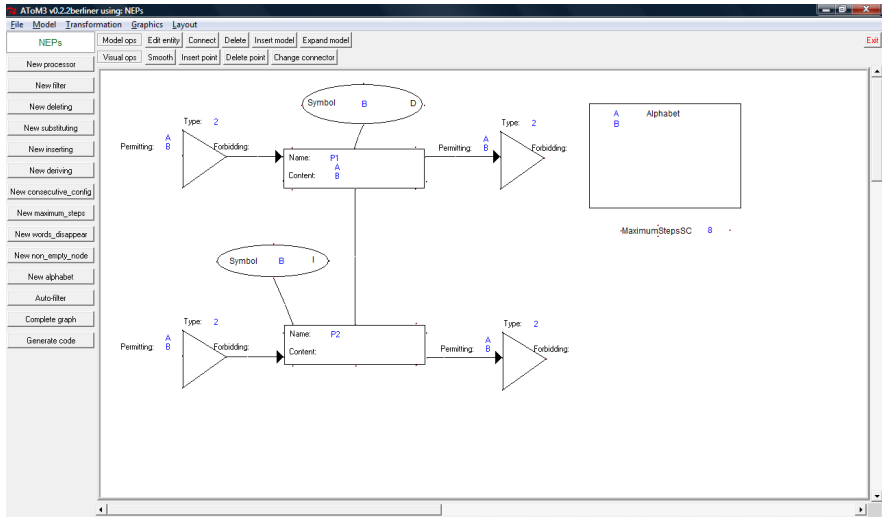


Fig. 11. Example of graphic representation of a NEP

*Programmer's viewpoint: how to graphically design NEPs* It is very simple to design models graphically, because the programmer only has to use different GUI elements (buttons, combo-boxes, pop-up menus, etc.) to draw the NEP on the canvas of the main window of the system.

Figure 11 also shows the main buttons of NEPv1. They are in the left margin of the window

## NEPsLingua

In this chapter we introduce NEPs-Lingua, the first textual programming language for NEPs. It is a first step to extend the P-Lingua approach to other bio-inspired models of computation. Our goal is to provide researchers with homogeneous family of languages for programming natural computers. Programmers who are familiar with a model will not have to learn a very different syntax if they try to use other models. This is why NEPs-Lingua is designed to be similar to P-Lingua. NEPs-Lingua has two main goals that will also be described in detail below:

1. Like P-Lingua, it aims to provide researchers with a syntax as close as possible to the one used to describe NEPs in the literature.



2. It tries to ease some usually boring, mechanical and time-consuming tasks needed to describe NEPs with the input formalisms of the available tools.

### *The NEPs-Lingua syntax*

In the following paragraphs we describe, mainly by examples, the syntax of NEPs-Lingua. A full ANTLR<sup>3</sup> description of the complete grammar may be ordered from the authors. The main components of a NEPs-Lingua program are atomic data, comments, nodes, the alphabet, the initial contents of the nodes, evolutionary rules, filters, the connections of the NEP graph and stopping conditions.

*Atoms* There are two classes of atomic data: alphanumeric strings of symbols (they have to start with an alphabetic character); and integer arithmetic expressions, with the usual mathematical notation, which include the operators in the set  $\{\wedge(\textit{power}), +, -, *, /\}$

*Comments* The typical C++ comments are also available in NEPs-Lingua.

- *Line comments* For example `// Comment`.  
The comment includes every symbol until the end of the line.
- *Multi line comments* For example

```
/*    ... Comment
...          */
```

Where the comment includes everything (even the *end of line* markers) between the symbols “/\*” and “\*/”.

*Alphabet* It is the alphabet of the NEP, a set of strings of symbols. The expression `@A={X,S,a,b,o,O}` defines an alphabet that contains the elements “X”, “S”, “a”, “b”, “O”, and “o”.

*Nodes* This is the most complex type of NEPs-Lingua data. There are two classes of nodes: with and without indexes. There are two kinds of indexes: numeric (defined by a range) and symbolic (defined by a set of strings of symbols). The syntax of indexes with numeric ranges is borrowed from P-Lingua.

---

<sup>3</sup> ANTLR is a Java tool for designing top-down parsers and language processors, developed by Terence Par. Further information can be found at <http://www.antlr.org/>



- *Non indexed nodes* The expression `{initial, final}` defines two nodes without indexes with the names *initial* and *final*.
- *Indexed nodes* The example defines a family of nodes with two indexes. One of them (i) takes its values from the interval  $[0, 10]$ . The values of the other (j) are taken from the set  $\{o, a, b\}$ .

`{m{i,j}: 0<=i<=10, j->{o,a,b}}`

The explicit set of the 33 defined nodes is  $\{m_{0,a}, m_{0,b}, m_{0,c}, \dots, m_{10,a}, m_{10,b}, m_{10,c}\}$ .

Different kinds of nodes can be mixed by means of the union operator. The next example defines a set of nodes that contains the two previous examples.

`@N={initial, final}+{m{i,j}: 0<=i<=10, j->{o,a,b}}`

*Initial content* It describes the set of strings that a given node initially contains. Notice that the node is written as a parameter of the *content directive* `@c`. The expression `@c{n{X}} = {X, S}` sets the initial content of the node  $n_X$  to  $\{X, S\}$

*Rules* Each type of rule has a different notation. Notice that, as in P-Lingua, the symbol `#` stands for the empty string and the string `-->` separates the left and right sides of the rule. The sentences `# -->a`, `a -->#` and `S-->aSb` are examples of insertion, deletion, and substitution (or deriving) rules, respectively.

All the rules for a given node are given together in the same sentence. The sentence `@r{n{S}} = {S-->aSb, S-->ab}` assigns two deriving rules to the node  $n_S$ .

*Filters* Each processor needs an input and an output filter. Several papers mentioned above define three components in the filters: their type and the permitting and forbidding contexts. We have grouped the different filters of the literature into six types (depending on how they are applied): types from 1 to 4 and filters defined by means of regular expressions or by means of sets of strings. Both contexts are just sets of symbols described by means of regular patterns or explicit sets of strings. The following examples define several filters:

`@pif{n{S}}={1, {abc, oo}}`  
`@fof{initial}={@regular_pattern, (((a[]b)+) )[(c*) )][ # ]}`  
`@pif{n{2,a}}={@set, {a,ab,aabb}}`





where `@pif` and `@fof` stand, respectively, for permitting input and forbidding output filter (the same notation is used for forbidding input and permitting output filters). In regular expressions `[]`, `]`, `+`, `*`, `#` represent intersection, union, `+` and `*`, and the empty string, respectively.

*Connections* This element makes it possible to get a compact representation of NEPs. There are two ways of defining connections: the directive `@complete`, which stands for a complete graph; and an explicit set of connections defined by means of pairs of nodes. The following examples show both options:

```
@C=@complete
C={ (final,n{X}), (n{X},m{9,a}) }
```

*Stopping conditions* The stopping conditions are written in a set after the directive `@S`. Each kind of condition is represented by its name and its required parameters. Both names and parameters are easy to identify in the following example:

```
@S={@no_change, @max_steps = 3+4,
    @non_empty_node={n{0}, n{X}} }
```

where `@no_change` stands for two consecutive equal configurations; `@max_steps` requires an expression to define the number of steps (the NEP stops after taking the given number of steps); and `@non_empty_node` includes a set of nodes whose contents are initially empty (the NEP stops when one of these nodes receives some string).

### Examples

In this section we will show some complete NEPs-Lingua programs. Our main goal is to highlight the two main characteristics of NEPs-Lingua: reducing the size and keeping close to the formal notation. For this purpose we will compare several NEPs-Lingua programs with NEPs examples taken from the literature. For reasons of space, we refer to the original papers for the detailed definition of the examples.

*Reducing the size of the representations* We shall first consider a very simple NEP. It has two nodes that delete and insert the symbol *B*. The initial word *AB* travels from one node to the other. The first node removes the symbol *B* from the string before leaving it in the net. The other node receives string *A* and adds symbol *B* again. The resulting string comes back to the initial node and the same process takes place again.

The XML file for jNEP is shown below:



```

<NEP nodes="2">
  <ALPHABET symbols="A_B"/>
  <GRAPH> <EDGE vertex1="0" vertex2="1"/> </GRAPH>
  <EVOLUTIONARY_PROCESSORS>
    <NODE initCond="A_B">
      <EVOLUTIONARY_RULES>
        <RULE ruleType="deletion" actionType="RIGHT"
          symbol="B"
          newSymbol=""/></EVOLUTIONARY_RULES>
      <FILTERS> <INPUT type="2"
        permittingContext="A_B"
        forbiddingContext=""/>
        <OUTPUT type="2"
          permittingContext="A_B"
          forbiddingContext=""/>
      </FILTERS>
    </NODE>
    <NODE initCond="">
      <EVOLUTIONARY_RULES>
        <RULE ruleType="insertion" actionType="RIGHT"
          symbol="B"
          newSymbol=""/> </EVOLUTIONARY_RULES>
      <FILTERS> <INPUT type="2"
        permittingContext="A_B"
        forbiddingContext=""/>
        <OUTPUT type="2"
          permittingContext="A_B"
          forbiddingContext=""/>
      </FILTERS>
    </NODE>
  </EVOLUTIONARY_PROCESSORS>
  <STOPPING_CONDITION>
    <CONDITION type="MaximumStepsStoppingCondition"
      maximum="8"/>
  </STOPPING_CONDITION>
</NEP>

```

(XML configuration file for a simple NEP with just two processors that send the words A and B back and forth)



It is easy to see that the NEPV1 program shown in figure 11 corresponds to this same NEP.

We will show below the NEPs-Lingua program for the previous example. With this simple case we can see that the NEPs-Lingua program is more compact than the other two representations described.

```
@A={A,B}
@N={ n{i}: 0 <= i <= 1}
@c{n{0}}={A,B}
@r{n{0}}={B-->#}
@r{n{1}}={#-->B}
@S={@max_steps = 8 }
@C={@complete}
```

The reduction in size increases as the complexity of the NEP increases. NEPs usually have complete graphs.

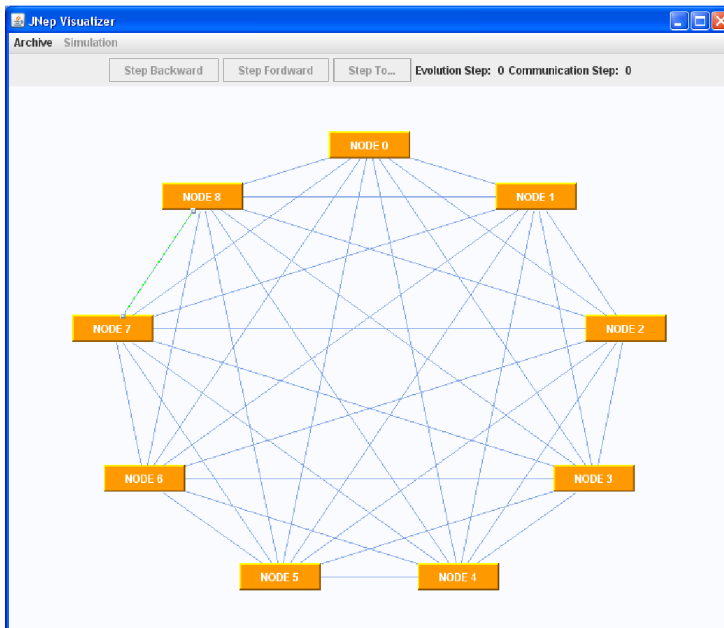


Fig. 12. jNEPview window showing the complete graph of a NEP with 9 processors.



Figure 12 shows the jNEPview window for a NEP with a complete graph with 9 nodes.

The XML configuration file for this NEP is forced to explicitly contain all the nodes and connections while the NEPs-Lingua source has to contain just the following two sentences:

```
@N={ n{i}: 0 <= i <= 8}
@C=@complete
```

[6] shows a NEP that can solve a small instance of the well known graph coloring problem with three different colours. It needs a complete graph with many more nodes than in the previous example.

The jNEPview window for this NEP is not shown here because it is difficult to handle: it looks like a ball of yarn. Once again the NEPs-Lingua program needs just the following two sentences:

```
@N={ n{i}: 0 <= i <= 50 } // Definition of 51 nodes
@C=@complete
```

*Keeping NEPs-Lingua as close as possible to the formal notation used in the literature.* The interested reader can easily see in the references for the last two examples (3-SAT and 3 coloring) that NEPs-Lingua syntax is mainly inspired by the formal notation used in the literature to describe NEPs.

[23] contains another example: a NEP associated with the context free grammar for axiom  $X$  with the derivation rules  $\{X \rightarrow SO, S \rightarrow aSb, S \rightarrow ab, O \rightarrow o, O \rightarrow oO, O \rightarrow Oo\}$

It is easy to see that the following NEPs-Lingua program for this NEP is quite similar to its formal definition.

```
@A={X,S,a,b,o,0} // Alphabet
@N= {final}+ {n{symbol}:symbol->{X,S,0}} /* Nodes associated
with non terminal symbols */
@c{n{X}}={X} // Initial content of the axiom node
@r{n{X}}= {X-->S0} // Deriving rules for the axiom
@r{n{S}}= {S-->aSb, S-->ab}
@r{n{0}}= {0-->o, 0-->o0, 0-->Oo}
@C=@complete // The graph is complete
@S={ @non_empty_node={final} } // Stopping conditions
```



*NEPs Lingua semantics*

The semantic constraints that every NEPs-Lingua program has to satisfy are outlined below:

- It has to contain exactly one alphabet and one set of node declarations.
- It needs at most one of the following elements:
  - Connection declaration set. By default, the graph is considered complete.
  - Set of stopping conditions. `@no_change` is assumed by default.
- Filters, rules and initial contents are optional.
- Nodes have to be defined before they are used.
- Each symbol representing rules, filters and initial contents has to be included in the alphabet.

NEPs-Lingua compilers should ensure these conditions. The last one is usually controlled by means of a symbol table that is filled while processing the declaration sentences and is consulted by the sentences that use nodes and symbols.

We have used different `Hashtable` Java objects to check these constraints. The following example shows some semantic mistakes:

```
@A={A}
@N={ n{i}: 0 <= j <= 1}
@c{n{0}}={A,B}
@r{n{0}}={B-->#}
@r{n{2}}={#-->B}
@S={@max_steps = 8 }
@C={@complete}
```

- The third, fourth and fifth lines contain the symbol B, which is not in the alphabet.
- The second line defines the index j, while the declared one is i
- The fifth line defines the rules for node  $n_2$ , but the value for index (2) is invalid



## References

1. Hochleistungsrechenzentrum bayern, <http://www.lrz.de>.
2. <http://jgrapht.sourceforge.net/>.
3. <http://www.wipd.ira.uka.de/javaparty/>.
4. <http://www.jgraph.com/jgraph.html>.
5. Message passing interface forum, mpi: A message-passing interface standard, university of tennessee, ut-cs-94-230, 1994.
6. E. Alfonseca. *An Approach for Automatic Generation of on-line Information Systems based on the Integration of Natural Language Processing and Adaptive Hypermedia techniques*. PhD thesis, Computer Science Department, UAM, 2003.
7. G. Bel Enguix, M. D. Jiménez-López, R. Mercaş, and A. Perekrstenko. Networks of evolutionary processors as natural language parsers. In *Proceedings ICAART 2009*, 2009.
8. T. Brants. Tnt—a statistical part-of-speech tagger. In *Proceedings of the 6th Conference on Applied Natural Language Processing*, 2000.
9. J. Castellanos, P. Leupold, and V. Mitrana. On the size complexity of hybrid networks of evolutionary processors. *Theoretical Computer Science*, 330(2):205–220, 2005.
10. J. Castellanos, C. Martín-Vide, V. Mitrana, and J. M. Sempere. Networks of evolutionary processors. *Acta Informatica*, 39(6-7):517–529, 2003.
11. Juan Castellanos, Carlos Martín-Vide, Victor Mitrana, and Jose M. Sempere. Solving np-complete problems with networks of evolutionary processors. In *Connectionist Models of Neurons, Learning Processes and Artificial Intelligence : 6th International Work-Conference on Artificial and Natural Neural Networks, IWANN 2001 Granada, Spain, June 13-15, 2001, Proceedings, Part I*, pages 621–, 2001.
12. A. Choudhary and K. Krithivasan. Network of evolutionary processors with splicing rules. *Mechanisms, Symbols and Models Underlying Cognition, Pt 1, Proceedings*, 3561:290–299, 2005.
13. E. Csuhaj-Varjú and V. Mitrana. Evolutionary systems: a language generating device inspired by evolving communities of cells. *Acta Informatica*, 36(11):913–926, May 2000.
14. E. Csuhaj-Varjú and A. Salomaa. *Lecture Notes on Computer Science 1218*, chapter Networks of parallel language processors. 1997.
15. Juan de Lara and Hans Vangheluwe. ATOM<sup>3</sup>: A tool for multi-formalism and meta-modelling. In *FASE'02*, pages 174–188. Springer-Verlag, 2002.
16. Rojas-J.M. Núñez R. Castañeda C. del Rosal, E. and A. Ortega. On the solution of np-complete problems by means of jnep run on computers. In *Proceedings of International Conference on Agents and Artificial Intelligence (ICAART 2009)*, pages 605–612, 2009.



17. J. Earley. An efficient context-free parsing algorithm. *Communications of the ACM*, 13(2):94–102, 1970.
18. Hartmut Ehrig, Karsten Ehrig, Ulrike Prange, and Gabriele Taentzer. *Fundamentals of algebraic graph transformation*. Springer-Verlag, 2006.
19. L. Fernández, V. J. Martínez, and L. F. Mingo. A hardware circuit for selecting active rules in transition p systems. In *Seventh International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC 2005)*, 2005.
20. Carlos Martín-Vide Florin Manea and Victor Mitrană. Accepting networks of splicing processors: Complexity results. *Theoretical Computer Science*, 371(1-2):72–82, 2007.
21. M. García-Quismondo, R. Gutiérrez-Escudero, M. A. Martínez del Amor, E. Orejuela, and I. Pérez-Hurtado. P-lingua 2.0: A software framework for cell-like p systems. *International Journal of Computers, Communications and Control*, IV(3):234–243, 2009.
22. C. Gomez, F. Javier, D. Valle Agudo, J. Rivero Espinosa, and D. Cuadra Fernandez. *Procesamiento del lenguaje Natural*, chapter Methodological approach for pragmatic annotation, pages 209–216. 2008.
23. Z. S. Harris. *String Analysis of Sentence Structure*. Mouton, The Hague, 1962.
24. Steven Kelly and Juha-Pekka Tolvanen. *Domain-Specific Modeling: Enabling Full Code Generation*. Wiley-IEEE Computer Society, 2008.
25. A. Lastovetsky and R. Reddy. Data partitioning with a realistic performance model of networks of heterogeneous computers. *Scientific Programming*, 2:93–112, 2005.
26. E. Santos Menendez R. Gonzalo M. A. Diaz, N. Gomez Blas and F. Gisbert. Networks of evolutionary processors (nep) as decision support systems. In *Fifth International Conference. Information Research and Applications ETHIA, 2007*, volume 1, pages 192–203, 2007.
27. F. Manea. Using ahneps in the recognition of context-free languages. In *In Proceedings of the Workshop on Symbolic Networks ECAI*, 2004.
28. Florin Manea and Victor Mitrană. All np-problems can be solved in polynomial time by accepting hybrid networks of evolutionary processors of constant size. *Information Processing Letters*, 103(3):112–118, July 2007.
29. M. Margenstern, V. Mitrană, and M. J. Perez-Jimenez. Accepting hybrid networks of evolutionary processors. *DNA Computing*, 3384:235–246, 2005.
30. C. Martín-Vide and V. Mitrană. Solving 3cnf-sat and hpp in linear time using www. *Machines, Computations, and Universality*, 3354:269–280, 2005.
31. C. Martín-Vide, V. Mitrană, M. J. Perez-Jimenez, and F. Sancho-Caparrini. Hybrid networks of evolutionary processors. *Genetic and Evolutionary Computation. GECCO 2003, PT I, Proceedings*, 2723:401–412, 2003.
32. Andrei Mikheev. Periods, capitalized words, etc. *Computational Linguistics*, 28(3):289–318, 2002.



33. R. Mitkov. *The Oxford Handbook of Computational Linguistics*. Oxford University Press, 2003.
34. Alfonso Ortega, Emilio del Rosal, Diana Pérez, Robert Mercas, Alexander Perekrestenko, and Manuel Alfonseca. *PNEPs, NEPs for Context Free Parsing: Application to Natural Language Processing*, chapter Bio-Inspired Systems: Computational and Ambient Intelligence, pages 472–479. LNCS. 2009.
35. S. Seifert and I. Fischer. *Parsing String Generating Hypergraph Grammars*. Springer, 2004.
36. M. A. Smith and Y. Bar-Yam. Cellular automaton simulation of pulsed field gel electrophoresis. *Electrophoresis*, 14(1):1522–2683, 1993.
37. TALP. <http://www.lsi.upc.edu/nlp/freeling/>, 2009.
38. T. Toffoli and N. Margolus. *Cellular Automata Machines*. MIT Press, London, 1987.
39. M. Volk. *Introduction to Natural Language Processing*,. Course CMSC 723 / LING 645 in the Stockholm University, Sweden., 2004.
40. W. Weaver. *Translation, Machine Translation of Languages: Fourteen Essays*. 1955.
41. A. Zollmann and A. Venugopal. Syntax augmented machine translation via chart parsing. In *Proceedings of the Workshop on Statistic Machine Translation*. HLT/NAACL, New York, June. 2006.







## Part III

---

### Applications



---

## NEPs Applied to Solve Specific Problems<sup>\*</sup>

Alfonso Ortega de la Puente<sup>1</sup>, Marina de la Cruz Echeandía<sup>1</sup>, Emilio del Rosal<sup>1</sup>, Rafael Nuñez Hervás<sup>3</sup>, Antonio Jiménez Martínez<sup>1</sup>, Carlos Castañeda<sup>1</sup>, José Miguel Rojas Siles<sup>2</sup>, Diana Pérez<sup>1</sup>, Robert Mercas<sup>4</sup>, Alexander Perekrestenko<sup>4</sup>, and Manuel Alfonseca<sup>1</sup>

<sup>1</sup> Departamento de Ingeniería Informática, Escuela Politécnica Superior  
Universidad Autónoma de Madrid  
Madrid, Spain

E-mail: {alfonso.ortega, marina.cruz, emilio.delrosal,  
antonio.jimenez, carlos.castaneda, manuel.alfonseca}@uam.es

<sup>2</sup> Departamento de Lenguajes y Sistemas Informáticos e Ingeniería de Software  
Universidad Politécnica de Madrid  
Madrid, Spain

E-mail: josemiguel.rojas@upm.es

<sup>3</sup> Escuela Politécnica Superior  
Universidad San Pablo CEU  
Madrid, Spain

E-mail: rnhervas@ceu.es

<sup>4</sup> GRLMC-Research Group on Mathematical Linguistics  
Universitat Rovira i Virgili  
Tarragona, Spain

E-mail: {robertgeorge.mercas,  
alexander.perekrestenko}@estudiants.urv.cat

---

<sup>\*</sup> Work partially supported by the Spanish Ministry of Science and Innovation under coordinated research project TIN2011-28260-C03-00 and research projects TIN2011-28260-C03-01, TIN2011-28260-C03-02 and TIN2011-28260-C03-03.

## 1 Solving NP-problems with Lineally Bounded Resources

In the following pages we will use NEPS to solve several small instances of well known NP problems. We will show computational implementations of NEPs.

In previous sections we have shown some results that prove the computational power of NEPs and the possibility of lineally bounding the temporal performance of their algorithms to solve NP problems. Their excellent performance depends on the following facts: NEPs are inherently parallel, and it is assumed that each of their processors has the spatial resources needed to store the results of applying all the possible rules to its contents. All these results are assumed to be generated at the same time.

NEPs have not been implemented in real hardware. So, in practice, NEPs have to be simulated on the architecture of one of the available computers. In practice it is very difficult to achieve lineal temporal performance because all these platforms need to explicitly handle all the possible results of all the processors. If all of them are simultaneously stored to be processed in parallel, the likely exponential temporal complexity turns into exponential spatial complexity. Nevertheless, it seems possible that the overall performance can be improved if we choose the proper platform. It is a trade-off between spatial and temporal needs.

Elsewhere in this volume, we have described different approaches to the simulation of NEPs in different platforms (including their efficient access to clusters of computers).

In this chapter we will not take into account the final platform but only will show how NP-problems can be computationally solved with NEPs and be simulated with jNEP. It is clear that we can improve the likely exponential temporal performance if we choose the proper final platform to run our programs.

### 1.1 Solving the SAT problem with jNEP

Reference [12] describes a NEP with splicing rules (ANSP) which solves the boolean satisfiability problem (SAT) with linear resources, in terms of the complexity classes also present in [12].

We have previously explained in this same volume that ANSP stands for Accepting Networks of Splicing Processors. In short, an ANSP is a NEP in



which the transformation rules of its nodes are *splicing rules*. The transformation performed by those rules is very similar to the genetic crossover. To be more precise, a *splicing rule*  $\sigma$  is a quadruple of words written as  $\sigma = [(x, y); (u, v)]$ . Given this *splicing rule*  $\sigma$  and two words  $(w, z)$ , the action of  $\sigma$  on  $(w, z)$  is defined as follows:

$$\sigma(w, z) = \{t \mid w = \alpha x y \beta, z = \gamma u v \delta \text{ for any words } \alpha, \beta, \gamma, \delta \text{ and } t = \alpha x v \delta \text{ or } t = \gamma u y \beta\}$$

We can use jNEP to actually build and run the ANSP that solves the boolean satisfiability problem (SAT). We will see how the features of NEPs and the *splicing rules* can be used to tackle this problem. The following is a broad summary of the configuration file for such an ANSP, applied to the solution of the SAT problem for three variables. The entire file can be downloaded from [jnep.e-delrosal.net](http://jnep.e-delrosal.net).

```
<NEP nodes="9">
  <ALPHABET symbols="A_B_C_!A_!B_!C_AND_OR_(.)_[A=1]_[B=1]_[C=1]_[A=0]_[B=0]_[C=0]_#_
    UP_{_}_1"/>
  <!-- WE IGNORE THE GRAPH TAG TO SAVE SPACE. THIS NEP HAVE A COMPLETE GRAPH -->
  <STOPPING_CONDITION>
    <CONDITION type="NonEmptyNodeStoppingCondition" nodeID="1"/>
  </STOPPING_CONDITION>
  <EVOLUTIONARY_PROCESSORS>
    <!-- INPUT NODE -->
    <NODE initCond="{_(_A_)_AND_(B_OR_C_)_}"
      auxiliaryWords="{_[A=1]_# {_[A=0]_# {_[B=1]_# {_[B=0]_# {_[C=1]_# {_[C=0]_#}">
      <EVOLUTIONARY_RULES>
        <RULE ruleType="splicing" wordX="{ " wordY="{ " wordU="{_[A=1]" wordV="#"/>
        <RULE ruleType="splicing" wordX="{ " wordY="{ " wordU="{_[A=0]" wordV="#"/>
        <RULE ruleType="splicing" wordX="{ " wordY="{ [A=0]" wordU="{_[B=0]" wordV="#"/>
        <RULE ruleType="splicing" wordX="{ " wordY="{ [A=0]" wordU="{_[B=1]" wordV="#"/>
        <RULE ruleType="splicing" wordX="{ " wordY="{ [A=1]" wordU="{_[B=0]" wordV="#"/>
        <RULE ruleType="splicing" wordX="{ " wordY="{ [A=1]" wordU="{_[B=1]" wordV="#"/>
        <RULE ruleType="splicing" wordX="{ " wordY="{ [B=0]" wordU="{_[C=0]" wordV="#"/>
        <RULE ruleType="splicing" wordX="{ " wordY="{ [B=0]" wordU="{_[C=1]" wordV="#"/>
        <RULE ruleType="splicing" wordX="{ " wordY="{ [B=1]" wordU="{_[C=0]" wordV="#"/>
        <RULE ruleType="splicing" wordX="{ " wordY="{ [B=1]" wordU="{_[C=1]" wordV="#"/>
      </EVOLUTIONARY_RULES>
    </FILTERS>
    <INPUT type="4"
      permittingContext=""
      forbiddingContext="[A=1]_[B=1]_[C=1]_[A=0]_[B=0]_[C=0]_#_UP_{_}_1"/>
    <OUTPUT type="4" permittingContext="[C=1]_[C=0]" forbiddingContext="">
    </FILTERS>
  </NODE>
  <!-- OUTPUT NODE -->
  <NODE initCond="">
    <EVOLUTIONARY_RULES>
    </EVOLUTIONARY_RULES>
  </FILTERS>
```



```

    <INPUT type="1" permittingContext=""
        forbiddingContext="A_B_C_!A_!B_!C_AND_OR_(_)"/>
    <OUTPUT type="1" permittingContext=""
        forbiddingContext="[A=1]_[B=1]_[C=1]_[A=0]_[B=0]_[C=0]_#_UP_{_}_1"/>
</FILTERS>
</NODE>
<!-- COMP NODE -->
<NODE initCond="" auxiliaryWords="#_ [A=0]_}_ #_ [A=1]_}_ #_}_ #_1_}_}">
  <EVOLUTIONARY_RULES>
    <RULE ruleType="splicing" wordX="" wordY="A_OR_1_}_}" wordU="#"
        wordV="1_}_}"/>
    <RULE ruleType="splicing" wordX="" wordY="!A_OR_1_}_}" wordU="#"
        wordV="1_}_}"/>
    <RULE ruleType="splicing" wordX="" wordY="B_OR_1_}_}" wordU="#"
        wordV="1_}_}"/>
    <RULE ruleType="splicing" wordX="" wordY="!B_OR_1_}_}" wordU="#"
        wordV="1_}_}"/>
    <RULE ruleType="splicing" wordX="" wordY="C_OR_1_}_}" wordU="#"
        wordV="1_}_}"/>
    <RULE ruleType="splicing" wordX="" wordY="!C_OR_1_}_}" wordU="#"
        wordV="1_}_}"/>
    <RULE ruleType="splicing" wordX="" wordY="AND_(1_}_}" wordU="#"
        wordV="}"/>
    <RULE ruleType="splicing" wordX="" wordY="[A=1]_(1_}_}" wordU="#"
        wordV="[A=1]_}_}"/>
    <RULE ruleType="splicing" wordX="" wordY="[A=0]_(1_}_}" wordU="#"
        wordV="[A=0]_}_}"/>
  </EVOLUTIONARY_RULES>
  <FILTERS>
    <INPUT type="1" permittingContext="1" forbiddingContext=""/>
    <OUTPUT type="1" permittingContext="" forbiddingContext="#_1"/>
  </FILTERS>
</NODE>
<!-- A=1 NODE -->
<NODE initCond="" auxiliaryWords="#_1_}_}_ #_}_}">
  <EVOLUTIONARY_RULES>
    <RULE ruleType="splicing" wordX="" wordY="A_}_}" wordU="#" wordV="1_}_}"/>
    <RULE ruleType="splicing" wordX="" wordY="( !A_}_}" wordU="#" wordV="UP"/>
    <RULE ruleType="splicing" wordX="" wordY="OR_!A_}_}" wordU="#" wordV="}"/>
    <RULE ruleType="splicing" wordX="" wordY="B_}_}" wordU="#" wordV="UP"/>
    <RULE ruleType="splicing" wordX="" wordY="C_}_}" wordU="#" wordV="UP"/>
  </EVOLUTIONARY_RULES>
  <FILTERS>
    <INPUT type="1" permittingContext="[A=1]" forbiddingContext="[A=0]_1"/>
    <OUTPUT type="1" permittingContext="" forbiddingContext="#_UP"/>
  </FILTERS>
</NODE>
<!-- A=0 NODE -->
<NODE initCond="" auxiliaryWords="#_1_}_}_ #_}_}">
  <EVOLUTIONARY_RULES>
    <RULE ruleType="splicing" wordX="" wordY="OR_A_}_}" wordU="#" wordV="}"/>
    <RULE ruleType="splicing" wordX="" wordY="(A_}_}" wordU="#" wordV="UP"/>
    <RULE ruleType="splicing" wordX="" wordY="!A_}_}" wordU="#" wordV="1"/>
    <RULE ruleType="splicing" wordX="" wordY="B_}_}" wordU="#" wordV="UP"/>
    <RULE ruleType="splicing" wordX="" wordY="C_}_}" wordU="#" wordV="UP"/>
  </EVOLUTIONARY_RULES>
  <FILTERS>

```



```

        <INPUT type="1" permittingContext="[A=0]" forbiddingContext="[A=1]_1"/>
        <OUTPUT type="1" permittingContext="" forbiddingContext="#_UP"/>
    </FILTERS>
</NODE>
<!-- NODES FOR 'B' AND 'C' ARE ANALOGOUS TO THOSE FOR 'A'. WE DO NOT PRESENT
THEM TO SAVE SPACE-->
</EVOLUTIONARY_PROCESSORS>
</NEP>

```

With this configuration file, at the end of its computation, jNEP outputs the interpretation which satisfies the logical formula contained in the file; namely

```
(_A_)_AND_( _B_OR_C_): {_ [C=0]_ [B=1]_ [A=1]_} {_ [C=1]_ [B=1]_ [A=1]_} {_ [C=1]_ [B=0]_ [A=1]_}
```

This ANSP can solve any formula with three variables. The formula to be solved must be specified as the value of the *initCond* attribute for the input node.

```

***** NEP INITIAL CONFIGURATION *****
--- Evolutionary Processor 0 ---
{(_A_)_AND_( _B_OR_C_)}

```

Our ANSP works as follows. Firstly, the first node creates all the possible combinations for the values of the 3 variables. We show below the jNEP output for the first step:

```

***** NEP CONFIGURATION - EVOLUTIONARY STEP ***
***** TOTAL STEPS: 1 *****
--- Evolutionary Processor 0 ---
{_# {_ [A=1]_ (_A_)_AND_( _B_OR_C_)} {_ [A=0]_ (_A_)_AND_( _B_OR_C_)} }

```

As shown, the splicing rules of the initial node have appended the two possible values of *A* to two copies of the logical formula. The rules concerned are:

```

<RULE ruleType="splicing" wordX="{ " wordY="{ " wordU="{_ [A=1] " wordV="#"/>
<RULE ruleType="splicing" wordX="{ " wordY="{ " wordU="{_ [A=0] " wordV="#"/>

```

This kind of rules (Manea's splicing rules) uses some auxiliary words that are never removed from the nodes. In our ANSP we use the following auxiliary words:

```
auxiliaryWords="{_ [A=1]_# {_ [A=0]_# {_ [B=1]_# {_ [B=0]_# {_ [C=1]_# {_ [C=0]_#"
```

The end of this first stage arises after  $2n - 1$  steps, where  $n$  is the number of variables:

```

--- Evolutionary Processor 0 ---
{_#
{[_ [C=0]_ [B=0]_ [A=0]_ (_A_)_AND_( _B_OR_C_)} {_ [C=0]_ [B=0]_ [A=1]_ (_A_)_AND_( _B_OR_C_)} }

```





```
{_[C=1]_[B=0]_[A=0]_( _A_ )_AND_( _B_OR_C_ )_} {_[C=1]_[B=0]_[A=1]_( _A_ )_AND_( _B_OR_C_ )_}
{_[C=0]_[B=1]_[A=0]_( _A_ )_AND_( _B_OR_C_ )_} {_[C=0]_[B=1]_[A=1]_( _A_ )_AND_( _B_OR_C_ )_}
{_[C=1]_[B=1]_[A=0]_( _A_ )_AND_( _B_OR_C_ )_} {_[C=1]_[B=1]_[A=1]_( _A_ )_AND_( _B_OR_C_ )_}
```

We should point out that NEPs take advantage of the fact that all the rules can be applied to one word in the same step. This is because the model states that each word has an arbitrary number of copies in its processor. Therefore, the above task (which is  $\Theta(2^n)$ ) can be completed in  $n$  steps, since each step double the number of words by including in each word a new variable with the value 1 or 0.

After this first stage, the words can leave the initial node and travel to the other nodes. In the net, there is one node per variable and value; in other words, there is one node for  $A = 1$ , another for  $C = 0$  and so on. Each of these nodes reduces, from right to left, the word representing the formula according to the variable values. For example, the sixth node is responsible for  $C = 1$  and, thus, makes the following modification to the word  $\{_[C=1]_[B=1]_[A=1]_( _A_ )_AND_( _B_OR_C_ )_ \}$ :

$$\{_[C=1]_[B=1]_[A=1]_( _A_ )_AND_( _B_OR_C_ )_ \} \implies$$

$$\{_[C=1]_[B=1]_[A=1]_( _A_ )_AND_( _B_OR_1_ )_ \}$$

However, the ninth node is responsible for  $C = 0$  and, therefore, produces the following change:

$$\{_[C=0]_[B=1]_[A=1]_( _A_ )_AND_( _B_OR_C_ )_ \} \implies$$

$$\{_[C=0]_[B=1]_[A=1]_( _A_ )_AND_( _B_ )_ \}$$

In this way, the nodes share the results of their modifications until one of them produces a word in which the formula is empty and only contains the left side with the variable values. This kind of words is allowed to pass through the input filter of the output node and will therefore enter it. At this point the NEP halts, since the stopping condition of the NEP states that a non-empty output node is the signal to stop the computation.

For further details, please refer to [12] and the implementation in `jnep-delrosal.net`.

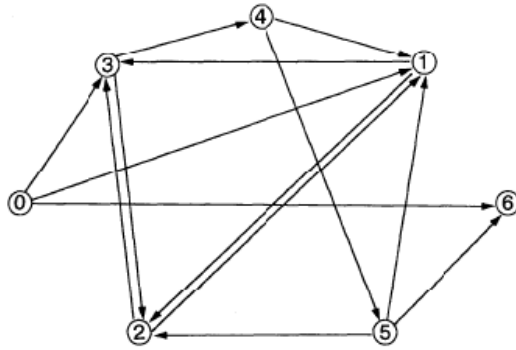


## 1.2 Solving an instance of the Hamiltonian path problem with jNEP

### *Hamiltonian path problem*

This well-known NP-complete problem searches an undirected graph for a Hamiltonian path; that is, one that visits each vertex exactly once.

In [1], Adleman proposed to solve this problem with polynomial resources by means of DNA manipulations in the laboratory. Figure 1 shows the graph he used. In this case, the solution is obvious (path 0-1-2-3-4-5-6). Despite its simplicity, Adleman described a general algorithm applicable to almost any graph with the same performance.



**Fig. 1.** Graph studied by Adleman

Adleman's algorithm can be summarized as follows:

1. Randomly generate all the possible paths.
2. Select those paths that begin and end in the proper nodes.
3. Select only the paths that contain exactly the total number of nodes.
4. Remove those paths that contain some node more than once.
5. The remaining paths are solutions to the problem.

The present study follows a similar approach (we have already introduced it in this same volume). Remember that the NEP graph is very similar to the one studied above: an extra node is added to ease the definition of the stopping condition. The set  $i, 0, 1, 2, 3, 4, 5, 6$  is used as the alphabet. Symbol  $i$  is the initial content of the initial node ( $v_0$ ). Each node (except the final one)



adds its number to the string received from the network. Input and output filters are defined to allow the communication of all the possible words without any special constraint. The input filter of the final node excludes any string which is not a solution. It is easy to imagine a regular expression for the set of solutions (those words with the proper length, the proper initial and final node and where each node appears only once). The NEP basic model defines filters by means of regular expressions. It is also easy to devise a set of additional nodes that performs the previous filter following Adleman's checks (proper beginning and end, proper length, and number of occurrences of each node). For the sake of simplicity we have explicitly used the solution word (i\_0\_1\_2\_3\_4\_5\_6) instead of a more complex regular expression or a greater NEP.

The reader will find at <http://jnep.e-delrosal.net> the complete XML file for this problem (Adleman.xml).

The XML file for this example defines the alphabet with this tag

```
<ALPHABET symbols="i_0_1_2_3_4_5_6" />
```

and the initial content of node 0 as

```
<NODE initCond="i">
```

The rules for adding the number of the node to its string are defined as follows (here for node 2)

```
<RULE ruleType = "insertion" actionType = "RIGHT" symbol = "2"/>
```

There are several ways of defining filters for the desired behavior (to allow the communication of all the possible words without any special constraint). We have used only the permitted input and output filters. A string can enter a node if it contains any of the symbols of the alphabet and no string is forbidden.

```
<FILTERS>
  <INPUT type="2"
    permittingContext="i_0_1_2_3_4_5_6"
    forbiddingContext="" />
  <OUTPUT type="2"
    permittingContext="i_0_1_2_3_4_5_6"
    forbiddingContext="" />
</FILTERS>
```

The behavior of the NEP is sketched as follows:

1. In the initial step the only non empty node is 0 and contains the string i
2. After the first step, 0 is added to this string and, node 0 therefore contains i\_0

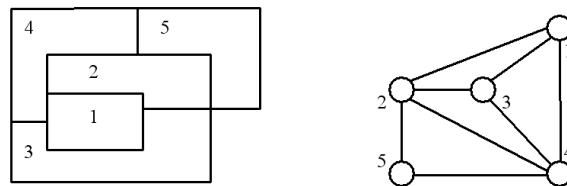


3. This string is moved to the nodes connected to node 0. In the next steps only nodes 1, 3 and 6 contain `i_0`.
4. These nodes add their number to the received string. In the next step their contents are, respectively, `i_0_1`, `i_0_3` and `i_0_6`.
5. This process is repeated as many times as necessary to produce a string that meets the conditions of the solution. In this final step the solution string `i_0_1_2_3_4_5_6` is sent to node 7 and the NEP stops.

Defining filters in the NEP model poses some difficulties to the design of NEPs and, thus, to the development of a simulator. These filters are defined [7] [6] by means of two pairs of filters (forbidden and allowed) to each operation (input and output). There are also several ways of combining and applying the filters to translate them into a set of strings. This mechanism contains obvious redundancies that make it difficult to design NEPs. It could be advisable a more general agreement of the researchers to ease and simplify the development of NEPs simulators.

### 1.3 Solving a graph coloring problem with jNEP

This problem describes a map whose regions have to be colored with only three colors. Adjacent regions must be colored in different colors. We have used the NEP defined in [6]. The map is translated into an undirected graph whose nodes stand for the regions and whose edges represent the adjacency relationship between regions. Figure 2 shows one of the examples we have studied. It is straightforward to prove that there is no solution to this map.



**Fig. 2.** Example of a map and its adjacency graph. In this case, there is no solution for the 3-colorability problem



The NEP has a complete graph with two special nodes (for the initial and final steps) and a set of seven nodes associated to each edge of the adjacency graph. These nodes perform the tasks outlined below.

The initial (final) node is responsible for starting (stopping) the computation. The seven nodes associated with an edge of the map are grouped in three pairs (one for each color). There is, in addition, a special node to communicate with the set of nodes of the next edge. Each pair is responsible for the main operation in the NEP: to check that a coloring constraint is not violated for the current edge. It performs this task in the following way:

Let us suppose that the color red is the one associated with the pair of nodes. The first node in the NEP associates the color red to the first node of the edge in the map. The second node in the NEP simultaneously keeps all the allowed coloring (two, in this case) for the second node of the edge: (blue and green). It is clear that the only acceptable colorings for this edge are red-blue and red-green.

The behavior of the complete NEP could be described as follows:

1. The initial node generates all the possible assignments of colors to all the regions in the map and adds a symbol to identify the first edge to be checked. These strings are communicated to all the nodes of the graph.
2. The set of nodes associated to each edge accepts only the strings marked with the symbol of the edge. These nodes remove all the strings that violate the coloring constraint for the regions of the edge. One special node in the set replaces the edge mark with that which corresponds to the next edge. In this way, the process continues.
3. The final node of the NEP collects the strings that satisfy the constraints of all the edges. It is straightforward to see that these strings are the solutions.

Some fragments of the XML file for this example (3Coloring.xml) are shown below to describe the above behavior in greater detail:

The alphabet of the NEP is defined as follows:

```
<ALPHABET
symbols="b1_r1_g1_b2_r2_g2_b3_r3_g3_b4_r4_g4_b5_r5_g5_B1_R1_G1_B2_R2_G2_B3_R3_G3
_B4_R4_G4_B5_R5_G5_a1_a2_a3_a4_a5_X1_X2_X3_X4_X5_X6_X8_X9"/>
```

This alphabet contains the following subsets of symbols: a1,...,a5 represents the initial situation of the regions (uncolored). b1, r1, g1,..., b5, r5, g5 represents the assignment of the colors to the regions. B1, R1, G1,..., B5,



R5, G5 is a copy of the previous set to be used while checking the constraint associated with a pair of adjacent regions.

The string contained in the initial node at the beginning represents the complete uncolored map and the number of the first edge to be tackled (X1)

```
<NODE initCond="a1_a2_a3_a4_a5_X1">
```

The rules of the initial node assign all the possible colors to all the regions. The following rules refer to the second region:

```
<RULE ruleType = "substitution"
  actionType = "ANY"
  symbol="a2" newSymbol="b2"/>
<RULE ruleType="substitution"
  actionType="ANY"
  symbol="a2" newSymbol="r2"/>
<RULE ruleType="substitution"
  actionType="ANY"
  symbol="a2" newSymbol="g2"/>
```

The node in the NEP that assigns a color (Red, in this case) to the first region (1 in the example) of an edge in the map uses the following rule:

```
<RULE ruleType="substitution"
  actionType="ANY" symbol="r1"
  newSymbol="R1"/>
```

The other node ensures that the adjacent region (2 in this case) has a different color by means of these rules:

```
<RULE ruleType="substitution"
  actionType="ANY"
  symbol="b2"
  newSymbol="B2"/>
<RULE ruleType="substitution"
  actionType="ANY" symbol="g2"
  newSymbol="G2"/>
```

The node used for starting the process in the next edge removes any special (capitalized) color symbol and sets the edge marking to the next one. The following rules correspond to the first edge

```
<RULE ruleType="substitution"
  actionType="ANY" symbol="R1"
  newSymbol="r1"/>
<RULE ruleType="substitution"
  actionType="ANY" symbol="B1"
  newSymbol="b1"/>
<RULE ruleType="substitution"
  actionType="ANY" symbol="G1"
  newSymbol="g1"/>
<RULE ruleType="substitution"
```



```

        actionType="ANY" symbol="R2"
        newSymbol="r2"/>
<RULE ruleType="substitution"
        actionType="ANY" symbol="B2"
        newSymbol="b2"/>
<RULE ruleType="substitution"
        actionType="ANY" symbol="G2"
        newSymbol="g2"/>
<RULE ruleType="substitution"
        actionType="ANY" symbol="X1"
        newSymbol="X2"/>

```

We found it difficult to apply the input and output filters as they are in [6]. In our opinion, greater standardization is advisable to minimize these situations. Notice that nodes associated with the last edge (in this case with number 8) mark their strings with the following number, which does not correspond to any edge in the graph (9 in our example). This is important for the design of the final node that checks the stopping condition (Non Empty Node Stopping Condition). This final node only accepts strings with the corresponding mark (one that does not correspond to any edge in the adjacency graph).

Figure 3 shows another map to be colored with 3 colors. It is generated by splitting region 3 and 4 in figure 2. Figure 3 also summarizes the sequence of steps for one of the possible solutions. It is worth noticing that all the solutions are simultaneously kept in the configurations of the NEP.

The behavior of the NEP for this map could be summarized as follows: the initial content of the initial node is `a1_a2_a3_a4_a5_X1`. This node produces all the possible coloring combinations. In the second step of the computation, for example, it contains the following strings:

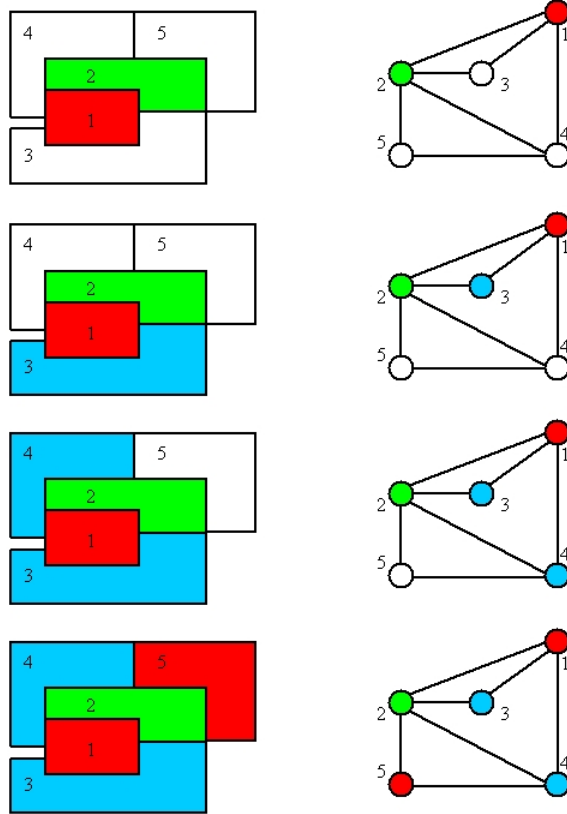
```

b1_a2_a3_a4_a5_X1 r1_a2_a3_a4_a5_X1
g1_a2_a3_a4_a5_X1 a1_b2_a3_a4_a5_X1
a1_r2_a3_a4_a5_X1 a1_g2_a3_a4_a5_X1
a1_a2_b3_a4_a5_X1 a1_a2_r3_a4_a5_X1
a1_a2_g3_a4_a5_X1 a1_a2_a3_b4_a5_X1
a1_a2_a3_r4_a5_X1 a1_a2_a3_g4_a5_X1
a1_a2_a3_a4_b5_X1 a1_a2_a3_a4_r5_X1
a1_a2_a3_a4_g5_X1

```

The NEP still needs a few more steps to get all the combinations. Then, the coloring constraints are applied simultaneously to all the possible solutions and those assignments that violate some constraint are removed. We describe below a sequence of strings generated by the NEP that corresponds to the





**Fig. 3.** Sequence of steps in the solution of a 3-coloring problem by jNEP

solution graphically shown in figure 2:  $r1\_g2\_b3\_b4\_r5\_X1$  is generated in the initial steps. After checking the 1st edge (regions 1 and 2) the NEP contains two strings:  $R1\_g2\_b3\_b4\_r5\_X1$  and  $R1\_G2\_b3\_b4\_r5\_X1$

After checking the 2nd edge (regions 1 and 3)  $R1\_g2\_B3\_b4\_r5\_X2$ . And after checking edges 3, 4, 5, 6 and 8 (remember that edge 7 was removed to make the map colorable) associated, respectively, with the pairs of regions 1-4, 2-3, 2-4, 2-5 and 4-5, the following strings are in the NEP:

$R1\_g2\_b3\_B4\_r5\_X3$

$r1\_G2\_B3\_b4\_r5\_X4$





r1\_G2\_b3\_B4\_r5\_X5                      r1\_G2\_b3\_b4\_R5\_X6  
 r1\_g2\_b3\_B4\_R5\_X8.

Finally, the complete solution is found to be r1\_g2\_b3\_B4\_R5\_X9 and r1\_g2\_b3\_b4\_r5\_X9

This NEP processes all the solutions at the same time. It removes all the coloring combinations that violate any constraint. In the last step the final node contains all the solutions found. [6] describes one of the kinds of NEPs (simple NEPs) that is simulated by jNEPs. As we have briefly mentioned before, we have observed that the authors have used slightly different filters for the 3-coloring problem. We could not use these filters and we had to change some of them (most of the output filters) for the NEP to behave properly. The complete XML file is available at <http://jnep.e-delrosal.net>.

## 2 Some Applications of NEPs to Language Processing

### 2.1 PNEPs: top-down parsing for natural languages

#### *Motivation*

Syntactic analysis is one of the classical problems related to language processing, and applies both to *artificial* languages (formal languages such as, for instance, programming languages) and to *natural* ones (those that people use to write and talk).

There is an ample range of parsing tools that computer scientists and linguists can use. They share a common goal (parsing), but have obvious differences: some are based on the theoretical foundations of Computer Science (automata, Chomsky grammars) while others mix several formal and informal techniques [14]: for example, generalized deterministic parsing, linear-time substring parsing, parallel parsing, parsing as intersection, non-canonical methods or non-Chomsky systems [15].

The characteristics of the particular language determine the suitability of the parsing technique. Two of the main differences between natural and formal languages are ambiguity and the size of the required representation. Ambiguity creates many difficulties for parsing, so programming languages are usually designed to be non ambiguous. On the other hand, ambiguity is an almost implicit characteristic of natural languages, so it should be taken into account by parsing techniques. To compare the size of different representations, the same formalism should be used. Context-free grammars are



widely used to describe the syntax of languages. It is possible to informally compare the sizes of context free grammars for some programming languages (such as C) and for some natural languages (such as Spanish). We conjecture that the representations required to parse natural languages are frequently greater than those required for high level imperative programming languages.

Parsing techniques for programming languages usually restrict the representation (grammar) used in different ways: it must be unambiguous, recursion is restricted, lambda rules must be removed, they must be (re)written according to some specific normal form, etc. These conditions mean that the designer of the grammar has more work to do, and that non-experts in the field of formal languages will have greater difficulty in properly understanding the grammar. This may be one of the reasons why formal representations such as grammars are little used or even unpopular. Natural languages usually do not fulfill these constraints.

These paragraphs focus on formal representations (based on Chomsky grammars) that can be used for syntactic analysis, and specially those which do not comply with these kinds of constraints. In this way, our approach will be applicable to both natural and formal languages.

Formal parsing techniques for natural languages are inefficient. The sentences that these techniques can usually parse are short (usually less than a typical computer program).

This chapter also focus on new models to increase the efficiency of parsing for languages with non-restricted context free grammars: we propose the use of NEPs as efficient parsing tools. In other sections of the current volume we show how we can efficiently access parallel hardware, such as clusters of computers, in order to simulate NEPs. Our goal is to provide the scientific community with efficient parsing tools that can be run on parallel platforms when they are available.

In the paragraphs below we will introduce the peculiarities of the syntactic analysis of natural languages, and PNEPs, an extension to NEPs that makes them suitable for efficient parsing of any kind of context free grammars, particularly those applicable to languages that share characteristics with natural languages (inherent ambiguity, for example). We have designed a top-down parser for context free grammars without additional constraints. Bellow we informally describe the algorithm, formally define it, detail a jNEP implementation and discuss some examples.



*Introduction to analysis of natural languages with NEPs*

Computational Linguistics researches linguistic phenomena that occur in digital data. Natural Language Processing (NLP) is a subfield of Computational Linguistics that focuses on building automatic systems that can interpret or generate information written in natural language [26]. This is a broad area which poses a number of challenges, both for theory and for applications.

Machine Translation was the first NLP application in the fifties [27]. In general, the main problem found in all cases is the inherent ambiguity of the language [22].

A typical NLP system has to cover several linguistic levels:

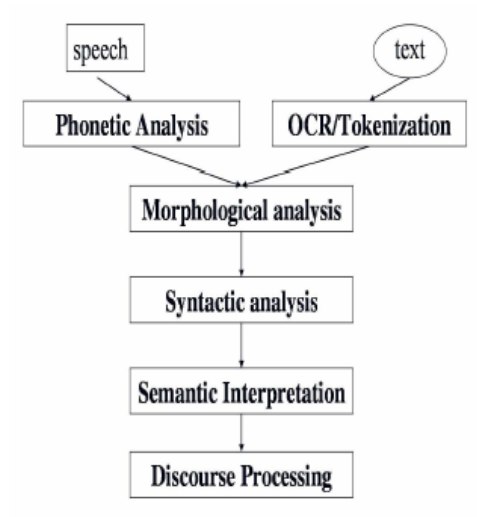
- **Phonological:** Sound processing to detect expression units in speech.
- **Morphological:** Extracting information about words, such as their part of speech and morphological characteristics [21, 2]. The best systems have an accuracy of 97% in this level [4].
- **Syntactical:** Using parsers to detect valid structures in the sentences, usually in terms of a certain grammar. One of the most efficient algorithms is the one described by Earley and its derivatives [11, 24, 28]. It provides parsing in polynomial time, with respect to the length of the input (linear in the average case;  $n^2$  and  $n^3$ , respectively, for unambiguous and ambiguous grammars in the worst case) These sections focus on this step. Syntactical analysis for natural language requires considerable of computational resources. Parsers can usually only completely analyze short sentences. Shallow parsing tries to overcome this difficulty. Instead of a complete derivation tree for the sentence, this parsing technique actually builds partial derivation trees for its elemental components.
- **Semantic:** Finding the most suitable knowledge formalism to represent the meaning of the text.
- **Pragmatic:** Interpreting the meaning of the sentence in a context which makes it possible to react accordingly.

The last two levels are still far from being solved [13].

Figure 4 shows the way in which typical NLP systems usually cover the linguistic levels described above.

A computational model that can be applied to NLP tasks is a network of evolutionary processors (NEPs). NEP as a generating device was first introduced in [10] and [9]. The topic is further investigated in [7], while further different variants of the generating machine are introduced and analyzed in [5, 17, 18, 19, 20].





**Fig. 4.** Typical phases in natural language processing

The first attempt was made to apply NEPs for syntactic NLP parsing in [3]. We have the same goal: to test the suitability of NEPs for this task. We have previously mentioned some performance characteristics of one of the most popular families of NLP parsers (those based on Earley's algorithm). We will conclude that the complexity of our approach is similar.

While [3] outlines a bottom up approach to natural language parsing with NEPs, we suggest a top-down strategy and show its possible use in a practical application.

#### *Top down parsing with NEPs and jNEP*

*Informal description* Other authors have studied the relationships between NEPs, regular, context-free, and recursively enumerable languages [14-18]. [21] shows how NEPs simulate the application of context free rules ( $A \rightarrow \alpha, A \in V, \alpha \in V^*$  for alphabet  $V$ ): a set of additional nodes is needed to implement a rather complex technique to rotate the string and locate  $A$  in one of the string ends, then delete it and add all the symbols in  $\alpha$ . PNEPs use context free rules rather than classic substitution ( $A \rightarrow B, A, B \in V$ ), as well as insertion and deletion NEP rules. In this way, the expressive power of NEP processors is bounded, while providing a more natural and comfortable way to describe the parsed language for practical purposes.



PNEPs implement a top down parser for context free grammars. Like any other parsers, PNEPs have to build the derivation tree of the string being parsed. We have added a simple mechanism to solve this task. We have used indexes to identify the rules added to the sentential form when each rule is applied. The generated string includes these indexes, making it possible to reconstruct the derivation tree. Several examples are shown below. A PNEP for a given grammar can explicitly generate all the possible derivations of each string in the language generated by the grammar. Its temporal complexity is bounded by the length of the analyzed string (actually by the depth of the derivation tree, that is, by the logarithm of the length). This bound can be used to stop the computation when processing incorrect strings, thus avoiding the possibility that the PNEP runs for an infinite number of steps. Nevertheless, this naive approach seems to be spatially inefficient, because of the high number of strings and derivations simultaneously considered which the processors have to store. We have added two additional mechanisms to overcome this inefficiency:

### **Discarding non promising sentential forms**

The first check we have implemented is the lightest, and it is present in almost all the parsers: discard any sentential form that contains a terminal symbol that the analyzed string does not contain. Parsers usually check sequentially for this condition, starting at the right end of the string. NEPs filters make it possible to check the condition regardless of the positions in the sentential form where the incorrect symbols are. We have implemented this feature by means of an additional node which contains just one deletion rule that deletes nothing (no symbol). In this way we can prevent the whole string from being lost when the evolutionary step is executed in the node. The pruning actually happens during the communication step, because it is implemented by the input filters. A string can pass the filter if it contains only non terminals, or terminals that belong to the input string being parsed. PNEPs duplicate the number of steps needed to parse a string, but reduce the number of strings stored by the processors.

### **Forcing a left-most derivation order**

Applying all the possible rules in parallel to a sentential form produces a lot of different derivations that are actually the same derivation tree. They only



differ in the order in which the non terminal symbols of the same sentential form are derived. We extend the NEP model with a new specialized kind of context free evolutive rule that applies only to the left-most non terminal. The symbol  $\rightarrow_l$  will be used to represent this kind of rule. The result of applying the rule  $r : A \rightarrow_l s$ , where  $s \in V^*$  ( $V$  stands for the NEP's alphabet) on a given string  $w$ , can be formally defined as follows:

$$r(w) = t \text{ where } w = w_1Aw_2, t = w_1sw_2, \text{ not\_contains}(w_1, A), s \in V^*, w_1 \in V^*, \text{ and } w_2 \in V^*$$

For example, the rule  $r : A \rightarrow_l s$  will change the following words as shown below:

$Aw_1 \Rightarrow sw_1$  (changes the left-most occurrence of non-terminal A, which is the left-most non-terminal)  
 $BAw_1 \Rightarrow Bsw_1$  (changes the left-most occurrence of non-terminal A, although non-terminal B is on its left)  
 $cdAw_1 \Rightarrow cds w_1$  (now there are terminals to the left of A)

### From context free grammars to PNEPs

The PNEP is built from the grammar in the following way:

1. We assume that each derivation rule in the grammar has a unique index that can be used to reconstruct the derivation tree.
2. There is a node for each non terminal (*deriving* nodes) that applies to its strings all the derivation rules for its left-most non terminal.
3. There is an additional node (*discarding* node) which discards non promising sentential forms. It receives all the sentential forms generated and sends to the net those that just contain non terminal symbols or terminals which are also contained in the input string.
4. The *deriving* nodes are connected only to the *discarding* node.
5. There is an output node, in which the *parsed string* can be found: this is a version of the input, enriched with information that will make it possible to reconstruct the derivation tree (the rules indexes).
6. The output node is connected with all the *deriving* nodes.

Obviously the same task can be performed using a trivial PNEP with only one *deriving* node for all the derivation rules. However, the proposed PNEP is easier to analyze and makes it easier to distribute the work among several nodes.



We will use the following grammar as an example for some of the steps outlined above. Let us consider grammar  $G_{a^n b^n o^m}$  induced by the following derivation rules (notice that indexes have been added in front of the corresponding right hand side):

$$X \rightarrow (1)SO, S \rightarrow (2)aSb|(3)ab, O \rightarrow (4)Oo|(5)oO|(6)o$$

It is easy to prove that the language corresponding to this grammar is  $\{a^n b^n o^m \mid n, m > 0\}$ . Furthermore, the grammar is ambiguous, since every sequence of  $o$  symbols can be generated in at least two different ways: by producing the new terminal  $o$  with rule 4 or rule 5.

The input filters of the output node describe parsed copies of the initial string. In other words, strings whose symbols are preceded by strings of any length (including 0) of the possible rules indexes. As an example, a parsed version of the string  $aabb oo$  would be  $12a3abb5o6o$ .

*Formal description* We will now describe the way in which our PNEP is defined, starting from a certain grammar. Given the context free grammar  $G = \{\Sigma_T = \{t_1, \dots, t_n\}, \Sigma_N = \{N_1, \dots, N_m\}, A, P\}$  with  $A \in \Sigma_N$  its axiom and  $P = \{N_i \rightarrow \gamma_j \mid j \in \{1, \dots, k\}, i \in \{1, \dots, n\} \wedge \gamma_j \in (\Sigma_T \cup \Sigma_N)^*\}$  its set of  $k$  production rules; the PNEP is defined as

$$\Gamma_G = (V = \Sigma_T \cup \Sigma_N \cup \{1, \dots, k\}, node_{output}, N_1, N_2, \dots, N_m, N_1^t, N_2^t, \dots, N_m^t, G)$$

where

1.  $N_i$  is the family of *deriving* nodes. Each node contains the following set of rules:  $\{N_i \rightarrow_l \gamma_j\}$  ( $\{N_i \rightarrow \gamma_j\}$  are the derivation rules for  $N_i$  in  $G$ )
2.  $N^t$  is the *discarding* node. As has been mentioned above it only contains the deletion rule  $\rightarrow$
3.  $node_{output}$  is the output node
4.  $G$  is a graph that contains an edge for
  - Each pair  $(N_i, node_{output})$
  - Each pair  $(N_i, N_i^t)$
5. The *input node*  $A$  is the only one with a non empty initial content ( $A$ )
6. The filters for each node are designed to produce the behavior informally described above. In general, the *deriving* nodes have empty output filters

For example, the PNEP for grammar  $G_{a^n b^n o^m}$  described above has a node for nonterminal  $S$  with the following substitution rules:  $\{S \rightarrow 2aSb, S \rightarrow 3ab\}$ . The input filter for this node allows all strings containing some copy of their non terminal  $S$  to be input in the node.



The input filter for the output node  $node_{output}$  has to describe what we have called *parsed strings*. Parsed strings will contain numbers, corresponding to the derivation rules which have been applied, among the symbols of the initial string. For  $PI_{node_{output}}$ , we can easily create membership conditions. For example, in order to parse the string *aabbo* with the grammar given above, the regular expression can be  $\{1, 2, 3, 4, 5, 6\}^*a\{1, 2, 3, 4, 5, 6\}^*a\{1, 2, 3, 4, 5, 6\}^*b\{1, 2, 3, 4, 5, 6\}^*b\{1, 2, 3, 4, 5, 6\}^*o$ . Our PNEP will stop computing whenever a string enters the output node.

For the discarding node,  $PI_{N^t}$  is a random context filter of type 2, where  $P = \{a, b, o, X, S, O\}$  and  $F = \emptyset$ . The derivation nodes have a random context  $PI_{N_i}$  of type 1, where  $P = \{N_i\}$  and  $F = \emptyset$ . Finally, any other filters are designed to accept any word without additional constraints.

The complete PNEP for our example ( $\Gamma_{a^n b^n o^m}$ ) is defined as follows:

- Alphabet  $V = \{X, O, S, a, b, o, 1, 2, 3, 4, 5, 6\}$
- Nodes
  - $node_{output}$ :
    - $A_{output} = \emptyset$  is the initial content;
    - $M_{output} = \emptyset$  is the set of rules;
    - $PI_{output} = \{ \text{(regular expression membership filter)};$
    - $\{\{1, 2, 3, 4, 5, 6\}^*a\{1, 2, 3, 4, 5, 6\}^*a\{1, 2, 3, 4, 5, 6\}^*b\{1, 2, 3, 4, 5, 6\}^*b\{1, 2, 3, 4, 5, 6\}^*o\}\}$ ;
    - $PO_{output} = \emptyset$  is the output filter
  - $N_X$ :
    - $A_X = \{X\}$ ;
    - $M_X = \{X \rightarrow_l 1SO\}$ ;
    - $PI_X = \{P = \{X\}, F = \emptyset\}$ ;
    - $PO_X = \emptyset$
  - $N_S$ :
    - $A_S = \emptyset$ ;
    - $M_S = \{S \rightarrow_l 2aSb, S \rightarrow_l 3ab\}$ ;
    - $PI_S = \{P = \{S\}, F = \emptyset\}$ ;
    - $PO_S = \emptyset$
  - $N_O$ :
    - $A_O = \emptyset$ ;
    - $M_O = \{O \rightarrow_l 4oO, O \rightarrow_l 5Oo, O \rightarrow_l 5o\}$ ;
    - $PI_O = \{P = \{O\}, F = \emptyset\}$ ;
    - $PO_O = \emptyset$
  - $N^t$ :
    - $A = \{\}$ ;
    - $M = \{\rightarrow\}$ ;





- $PI = \{P = \{X, O, S, a, b, o\}, F = \emptyset\};$
- $PO = \{F = \emptyset, P = \emptyset\}$
- Its graph contains an edge for each pair  $\{(N_X, N^t), (N_S, N^t), (N_O, N^t), (N_X, node_{output}), (N_S, node_{output}), (N_O, node_{output})\}$
- It stops the computation when some string enters  $node_{output}$

The following shows some of the strings generated by all the nodes of the PNEP in successive communication steps, when parsing the string *aboo* (each set corresponds to a different step):

$\{X\} \Rightarrow \{1SO\} \Rightarrow \{\dots, 13abO, \dots\} \Rightarrow \{\dots, 13ab4Oo, 13ab5oO, \dots\} \Rightarrow \{\dots, 13ab46oo, \dots, 13ab5o6o, \dots\}$

The last set contains two different derivations for *aboo* by  $(G_{a^n b^n n o^m})$ , which can enter the output node and stop the computation of the PNEP.

It is easy to reconstruct the derivation tree from the parsed strings in the output node, by following their sequence of numbers. For example, consider the parsed string *13ab6o* and its sequence of indexes 136; *abo* is generated in the following steps:  $X \Rightarrow (\text{rule 1 } X \Rightarrow SO) SO$ ,  $SO \Rightarrow (\text{rule 3 } S \Rightarrow ab) abO$ ,  $abO \Rightarrow (\text{rule 6 } O \Rightarrow o) abo$

*jNEP description of PNEPs* In a previous section we have described the structure of the XML input files for *jNEP*.

In order to keep *jNEP* as general as possible, we have added new xml descriptions for each extension needed in PNEP.

Context free rules are represented in the xml file as follows:

```
<RULE ruleType='contextFreeParsing' symbol='[symbol]' newString='[symbolList]'/>
```

Those applied to the left-most non terminal, however, use this syntax:

```
<RULE ruleType="leftMostParsing" symbol="NON-TERMINAL" string="SUBSTITUTION_STRING" nonTerminals="GRAMMAR_NON-TERMINALS"/>
```

Three of the sections of the xml representation of the PNEP  $\Gamma_{a^n b^n n o^m}$  defined above (the output node, the *derivating* node for axiom *X* and the *discarding* node) are shown below.

```
<NODE initCond="">
  <EVOLUTIONARY_RULES>
    <RULE ruleType="deletion" actionType="RIGHT" symbol=""/>
  </EVOLUTIONARY_RULES>
  <FILTERS>
    <INPUT type="RegularLangMembershipFilter"
      regularExpression="[1-6]*a[1-6]*b[1-6]*o[1-6]*o"/>
    <OUTPUT type="1" permittingContext="" forbiddingContext="a_b_o_o"/>
  </FILTERS>
</NODE>
```



```

<NODE initCond="X">
  <EVOLUTIONARY_RULES>
    <RULE ruleType="leftMostParsing" symbol="X" string="1_S_0" nonTerminals="S_0_X"/>
  </EVOLUTIONARY_RULES>
  <FILTERS>
    <INPUT type="1" permittingContext="X" forbiddingContext=""/>
  </FILTERS>
</NODE>

<NODE initCond="">
  <EVOLUTIONARY_RULES>
    <RULE ruleType="deletion" actionType="RIGHT" symbol=""/>
  </EVOLUTIONARY_RULES>
  <FILTERS>
    <INPUT type="2" permittingContext="a_b_o_o_0_1_2_3_4_5_S_0_X"
      forbiddingContext=""/>
  </FILTERS>
</NODE>

```

The nodes for other non terminal symbols are similar, but with an empty ("" ) *initial condition* and their corresponding derivation rules.

### *An example for natural language processing*

As described in a previous section, the complexity of the grammars used for syntactic parsing depends on the desired target. These grammars are usually very complex, which makes them one of the bottlenecks in NLP tasks.

We will use the grammar deduced from the following derivation rules, whose axiom is the non terminal Sentence. This grammar is similar to grammars devised by other authors in previous attempts to use NEPs for parsing (natural) languages [19]. We have added the index of the derivation rules that will be used later.



Sentence  $\rightarrow$  (1) NounPhraseStandard PredicateStandard  
                   | (2) NounPhrase3Singular Predicate3Singular  
 NounPhrase3Singular  $\rightarrow$  (3) DeterminantAn VowelNounSingular  
                   | (4) DeterminantSingular NounSingular  
                   | (5) Pronoun3Singular  
 NounPhraseStandard  $\rightarrow$  (6) DeterminantPlural NounPlural  
                   | (7) PronounNo3Singular  
 NounPhrase  $\rightarrow$  (8) NounPhrase3Singular | (9) NounPhraseStandard  
 PredicateStandard  $\rightarrow$  (10) VerbStandard NounPhrase  
 Predicate3Singular  $\rightarrow$  (11) Verb3Singular NounPhrase  
 DeterminantSingular  $\rightarrow$  (12) a | (13) the | (14) this  
 DeterminantAn  $\rightarrow$  (15) an  
 VowelNounSingular  $\rightarrow$  (16) apple  
 NounSingular  $\rightarrow$  (17) boy  
 Pronoun3Singular  $\rightarrow$  (18) he | (19) she | (20) it  
 DeterminantPlural  $\rightarrow$  (21) the | (22) several | (23) these  
 NounPlural  $\rightarrow$  (24) apples | (25) boys  
 PronounNo3Singular  $\rightarrow$  (26) I | (27) you | (28) we | (29) they  
 VerbStandard  $\rightarrow$  (30) eat  
 Verb3Singular  $\rightarrow$  (31) eats

As we have described above, NLP syntax parsing usually takes the results of the morphological analysis as input. In this way, the previous grammar can be simplified by removing the derivation rules for the last 9 non terminals (from DeterminantSingular to Verb3Singular): these symbols become terminals for the new grammar.

Notice, also, that this grammar implements grammatical agreement by means of context free rules. For each non terminal, we had to use several different *specialized versions*. For instance, NounPhraseStandard and NounPhrase3Singular are specialized versions of non terminal NounPhrase. These rules increase the complexity of the grammar.

We can build the PNEP associated with this context-free grammar by following the steps described in the corresponding section.

Let us consider the English sentence *the boy eats an apple*. Some of the strings generated by the nodes of the PNEP in successive communication steps while parsing this string are shown below (we show the initials, rather than the full name of the symbols).

A left derivation of the string is highlighted:  $\{ S \} \Rightarrow \{ \dots, 2 \text{ NF3S P3S}, \dots \} \Rightarrow \{ \dots, 2 \text{ 4 DS NS P3S}, \dots \} \Rightarrow \{ \dots, 2 \text{ 4 13 the NS P3S}, \dots \} \Rightarrow \{ \dots, 2 \text{ 4 13 the 17 boy P3S}, \dots \} \Rightarrow \{ \dots, 2 \text{ 4 13 the 17 boy 11 V3S NF}, \dots \} \Rightarrow \{ \dots,$



2 4 13 the 17 boy 11 31 eats NF, ... }  $\Rightarrow$  { ..., 2 4 13 the 17 boy 11 31 eats 8 NF3S, ... }  $\Rightarrow$  { ..., 2 4 13 the 17 boy 11 31 eats 8 3 DA VNS, ... }  $\Rightarrow$  { ..., 2 4 13 the 17 boy 11 31 eats 8 3 15 an VNS, ... }  $\Rightarrow$  { ..., 2 4 13 the 17 boy 11 31 eats 8 3 15 an 16 apple, ... }

The following fragments of the jNEP output for this case show more detail of the contents of some nodes of the PNEP during its execution.

Notice that

- Node 16 is the *discarding* node, node 17 is the output node and the rest are the *deriving* nodes.
- The indexes of the rules added to the string in order to build the derivation tree include two numbers:
  1. The first one identifies their non terminal
  2. The second identifies the right hand side
 For example, index 1-8 refers to the eighth right hand side of the first non terminal.
- The string [...] means that a piece of output is not shown to save space. Comments are also written between square brackets.

```

*****                                NEP INITIAL CONFIGURATION                                *****
--- Evolutionary Processor 0 ---
[...]
--- Evolutionary Processor 9 ---

--- Evolutionary Processor 10 ---
Sentence
--- Evolutionary Processor 11 ---
[...]
--- Evolutionary Processor 16 ---

--- Evolutionary Processor 17 ---

***** NEP CONFIGURATION - EVOLUTIONARY STEP - TOTAL STEPS: 1 *****
[...]
--- Evolutionary Processor 10 ---
10-1_NounPhraseStandard_PredicateStandard 10-0_NounPhrase3Singular_Predicate3Singular
[...]
--- Evolutionary Processor 16 ---

--- Evolutionary Processor 17 ---

***** NEP CONFIGURATION - COMMUNICATION STEP - TOTAL STEPS: 2 *****
--- Evolutionary Processor 0 ---
[...]
--- Evolutionary Processor 15 ---

```



```

--- Evolutionary Processor 16 ---
10-1_NounPhraseStandard_PredicateStandard 10-0_NounPhrase3Singular_Predicate3Singular
--- Evolutionary Processor 17 ---

***** NEP CONFIGURATION - EVOLUTIONARY STEP - TOTAL STEPS: 3 *****
--- Evolutionary Processor 0 ---
[...]
--- Evolutionary Processor 15 ---

--- Evolutionary Processor 16 ---
10-1_NounPhraseStandard_PredicateStandard 10-0_NounPhrase3Singular_Predicate3Singular
--- Evolutionary Processor 17 ---

***** NEP CONFIGURATION - COMMUNICATION STEP - TOTAL STEPS: 4 *****
--- Evolutionary Processor 0 ---

--- Evolutionary Processor 1 ---

--- Evolutionary Processor 2 ---

--- Evolutionary Processor 3 ---
10-0_NounPhrase3Singular_Predicate3Singular
--- Evolutionary Processor 4 ---

--- Evolutionary Processor 5 ---

--- Evolutionary Processor 6 ---

--- Evolutionary Processor 7 ---
10-1_NounPhraseStandard_PredicateStandard
--- Evolutionary Processor 8 ---

--- Evolutionary Processor 9 ---

--- Evolutionary Processor 10 ---

--- Evolutionary Processor 11 ---

--- Evolutionary Processor 12 ---

--- Evolutionary Processor 13 ---
10-0_NounPhrase3Singular_Predicate3Singular
--- Evolutionary Processor 14 ---

--- Evolutionary Processor 15 ---
10-1_NounPhraseStandard_PredicateStandard
--- Evolutionary Processor 16 ---

--- Evolutionary Processor 17 ---

***** NEP CONFIGURATION - EVOLUTIONARY STEP - TOTAL STEPS: 5 *****
[...]
***** NEP CONFIGURATION - COMMUNICATION STEP - TOTAL STEPS: 6 *****
[...]
```



```

***** NEP CONFIGURATION - EVOLUTIONARY STEP - TOTAL STEPS: 7 *****
[...]
***** NEP CONFIGURATION - COMMUNICATION STEP - TOTAL STEPS: 8 *****
[...]
***** NEP CONFIGURATION - EVOLUTIONARY STEP - TOTAL STEPS: 9 *****
--- Evolutionary Processor 0 ---

--- Evolutionary Processor 1 ---

--- Evolutionary Processor 2 ---
10-0_3-0_2-2_he_Predicate3Singular 10-0_3-0_2-1_she_Predicate3Singular
10-0_3-0_2-0_it_Predicate3Singular
--- Evolutionary Processor 3 ---
10-0_3-1_DeterminantSingular_NounSingular_Predicate3Singular
10-0_3-0_Pronoun3Singular_Predicate3Singular
10-0_3-2_DeterminantAn_VowelNounSingular_Predicate3Singular
--- Evolutionary Processor 4 ---
10-1_7-1_4-1_several_NounPlural_PredicateStandard
10-1_7-1_4-2_the_NounPlural_PredicateStandard
10-1_7-1_4-0_these_NounPlural_PredicateStandard
--- Evolutionary Processor 5 ---
10-0_3-2_5-0_an_VowelNounSingular_Predicate3Singular
--- Evolutionary Processor 6 ---

--- Evolutionary Processor 7 ---

--- Evolutionary Processor 8 ---
10-0_3-1_DeterminantSingular_NounSingular_Predicate3Singular
--- Evolutionary Processor 9 ---
10-1_7-0_9-2_you_PredicateStandard 10-1_7-0_9-0_they_PredicateStandard
10-1_7-0_9-3_I_PredicateStandard
10-1_7-0_9-1_we_PredicateStandard
--- Evolutionary Processor 10 ---

--- Evolutionary Processor 11 ---
10-0_3-1_11-2_a_NounSingular_Predicate3Singular
10-0_3-1_11-0_this_NounSingular_Predicate3Singular
10-0_3-1_11-1_the_NounSingular_Predicate3Singular
--- Evolutionary Processor 12 ---
10-0_3-2_DeterminantAn_VowelNounSingular_Predicate3Singular
--- Evolutionary Processor 13 ---
10-0_3-1_DeterminantSingular_NounSingular_Predicate3Singular
10-0_3-0_Pronoun3Singular_Predicate3Singular
10-0_3-2_DeterminantAn_VowelNounSingular_Predicate3Singular
10-0_NounPhrase3Singular_Predicate3Singular
--- Evolutionary Processor 14 ---
10-1_7-1_DeterminantPlural_NounPlural_PredicateStandard
--- Evolutionary Processor 15 ---
10-1_NounPhraseStandard_PredicateStandard
10-1_7-1_DeterminantPlural_NounPlural_PredicateStandard
10-1_7-0_PronounNo3Singular_PredicateStandard
--- Evolutionary Processor 16 ---

--- Evolutionary Processor 17 ---

***** NEP CONFIGURATION - COMMUNICATION STEP - TOTAL STEPS: 10 *****

```



```

--- Evolutionary Processor 0 ---
[...]
[AT THIS POINT, PARSING TREES WITH INCORRECT TERMINALS HAVE BEEN PRUNED]
--- Evolutionary Processor 16 ---
10-0_3-2_5-0_an_VowelNounSingular_Predicate3Singular
10-1_7-1_DeterminantPlural_NounPlural_PredicateStandard
10-0_3-1_DeterminantSingular_NounSingular_Predicate3Singular
10-0_3-0_Pronoun3Singular_Predicate3Singular
10-0_3-2_DeterminantAn_VowelNounSingular_Predicate3Singular
10-1_7-1_4-2_the_NounPlural_PredicateStandard
10-0_3-1_11-1_the_NounSingular_Predicate3Singular
10-0_NounPhrase3Singular_Predicate3Singular
--- Evolutionary Processor 17 ---

***** NEP CONFIGURATION - EVOLUTIONARY STEP - TOTAL STEPS: 11 *****
--- Evolutionary Processor 0 ---
[...]
--- Evolutionary Processor 16 ---
10-0_3-2_5-0_an_VowelNounSingular_Predicate3Singular
10-1_7-1_DeterminantPlural_NounPlural_PredicateStandard
10-0_3-1_DeterminantSingular_NounSingular_Predicate3Singular
10-0_3-0_Pronoun3Singular_Predicate3Singular
10-0_3-2_DeterminantAn_VowelNounSingular_Predicate3Singular
10-1_7-1_4-2_the_NounPlural_PredicateStandard
10-0_3-1_11-1_the_NounSingular_Predicate3Singular
10-0_NounPhrase3Singular_Predicate3Singular
--- Evolutionary Processor 17 ---

***** NEP CONFIGURATION - COMMUNICATION STEP - TOTAL STEPS: 12 *****
--- Evolutionary Processor 0 ---

--- Evolutionary Processor 1 ---

--- Evolutionary Processor 2 ---
10-0_3-0_Pronoun3Singular_Predicate3Singular
--- Evolutionary Processor 3 ---
10-0_NounPhrase3Singular_Predicate3Singular
--- Evolutionary Processor 4 ---
10-1_7-1_DeterminantPlural_NounPlural_PredicateStandard
--- Evolutionary Processor 5 ---
10-0_3-2_DeterminantAn_VowelNounSingular_Predicate3Singular
--- Evolutionary Processor 6 ---

--- Evolutionary Processor 7 ---

--- Evolutionary Processor 8 ---
10-0_3-1_DeterminantSingular_NounSingular_Predicate3Singular
10-0_3-1_11-1_the_NounSingular_Predicate3Singular
--- Evolutionary Processor 9 ---

--- Evolutionary Processor 10 ---

--- Evolutionary Processor 11 ---
10-0_3-1_DeterminantSingular_NounSingular_Predicate3Singular
--- Evolutionary Processor 12 ---

```



```

10-0_3-2_5-0_an_VowelNounSingular_Predicate3Singular
10-0_3-2_DeterminantAn_VowelNounSingular_Predicate3Singular
--- Evolutionary Processor 13 ---
10-0_3-2_5-0_an_VowelNounSingular_Predicate3Singular
10-0_3-1_DeterminantSingular_NounSingular_Predicate3Singular
10-0_3-0_Pronoun3Singular_Predicate3Singular
10-0_3-2_DeterminantAn_VowelNounSingular_Predicate3Singular
10-0_3-1_11-1_the_NounSingular_Predicate3Singular
10-0_NounPhrase3Singular_Predicate3Singular
--- Evolutionary Processor 14 ---
10-1_7-1_DeterminantPlural_NounPlural_PredicateStandard
10-1_7-1_4-2_the_NounPlural_PredicateStandard
--- Evolutionary Processor 15 ---
10-1_NounPhraseStandard_PredicateStandard
10-1_7-1_DeterminantPlural_NounPlural_PredicateStandard
10-1_7-0_PronounNo3Singular_PredicateStandard
10-1_7-1_4-2_the_NounPlural_PredicateStandard
--- Evolutionary Processor 16 ---

--- Evolutionary Processor 17 ---

***** NEP CONFIGURATION - EVOLUTIONARY STEP - TOTAL STEPS: 13 *****
[...]
***** NEP CONFIGURATION - COMMUNICATION STEP - TOTAL STEPS: 14 *****
[...]
***** NEP CONFIGURATION - EVOLUTIONARY STEP - TOTAL STEPS: 15 *****
[...]
[...]
[...]
***** NEP CONFIGURATION - COMMUNICATION STEP - TOTAL STEPS: 38 *****
--- Evolutionary Processor 0 ---
[...]
--- Evolutionary Processor 16 ---
10-0_3-2_5-0_an_12-0_apple_13-0_6-0_eats_0-0_7-0_PronounNo3Singular
10-0_3-2_5-0_an_12-0_apple_13-0_6-0_eats_0-1_3-2_5-0_an_VowelNounSingular
10-0_3-2_5-0_an_12-0_apple_13-0_6-0_eats_0-1_3-1_11-1_the_8-0_boy
[...]
10-0_3-1_11-1_the_8-0_boy_13-0_6-0_eats_0-0_NounPhraseStandard
10-0_3-1_11-1_the_8-0_boy_13-0_6-0_eats_0-1_3-2_DeterminantAn_VowelNounSingular
10-0_3-1_11-1_the_8-0_boy_13-0_6-0_eats_0-0_7-1_DeterminantPlural_NounPlural
[...]
--- Evolutionary Processor 17 ---
10-0_3-1_11-1_the_8-0_boy_13-0_6-0_eats_0-1_3-2_5-0_an_12-0_apple

----- NEP has stopped!!! -----

Stopping condition found: net.e.delrosal.jnep.stopping.NonEmptyNodeStoppingCondition
-----

```

If we analyze an incorrect sentence, such as *the boy eat the apple*, the PNEP will continue the computation after the steps summarized above, because in this case it is impossible to find a parsed string. To modify our PNEP to stop





when this happens, it is enough to take into account that the length of the input string is a bound for the number of steps needed (it is always possible to get equivalent context free grammars without chaining and lambda rules; in addition, the length of a given string is usually less than the depth of its derivation trees).

## 2.2 PNEPs for shallow parsing

### *Motivation*

The goal of the following paragraphs is to modify and use PNEPs for shallow parsing. Shallow parsing will be described later. It is a parsing technique frequently used in natural language processing to overcome the inefficiency of other approaches to syntactic analysis.

Some of the authors of this contribution were involved in developing IBERIA, a corpus of scientific Spanish which is able to process the sentences at the morphological level.

We are very interested in adding syntactic analysis tools to IBERIA. The current contribution has this goal.

Below we will introduce shallow parsing and FreeLing, a well-known free platform that offers parsing tools such as a Spanish grammar and shallow parsers for this grammar.

Then we will show how PNEPs can be used for shallow parsing and describe a jNEP implementation. Finally some examples will be given.

### *Introduction to FreeLing and shallow parsing*

Let us summarize some of the main difficulties encountered by parsing techniques when building complete parsing trees for natural languages:

- Spatial and temporal performance of the analysis. The Early algorithm and its derivatives [11, 24, 28] are some of the most efficient approaches. They, for example, provide parsing in polynomial time, with respect to the length of the input. Its time complexity for parsing context-free languages is linear in the average case, while in the worst case it is  $n^2$  and  $n^3$ , respectively, for unambiguous and ambiguous grammars.
- The size and complexity of the corresponding grammar, which is also difficult to design. Natural languages, for instance, are usually ambiguous.



The goal of shallow parsing is to analyze the main components of the sentences (for example, noun groups, verb groups, etc.) rather than complete sentences. It ignores the actual syntactic structure of the sentences, which are considered to be merely sets of these basic blocks. Shallow parsing tries to overcome, in this way, the performance difficulties that arise when building complete derivation trees.

Shallow parsing produces sequences of subtrees. These subtrees are frequently shown as children of a fictitious root node. This way of presenting the results of the analysis can confuse the inexperienced reader, because the final tree is not a real derivation tree: neither is its root the axiom of the grammar nor its branches correspond to actual derivation rules.

Shallow parsing includes different particular algorithms and tools (for instance FreeLing [25] or cascades of finite-state automata [16])

FreeLing is *An Open Source Suite of Language Analyzers* that provides the scientist with several tools and techniques. FreeLing includes a Spanish context-free grammar, adapted for shallow parsing, that does not contain a real axiom. This grammar has almost two hundred non-terminals and approximately one thousand rules. The actual number of rules is even greater, because they use regular expressions rather than terminal symbols. Each rule, then, represents a set of rules, depending on the terminal symbols that match the regular expressions.

The terminals of the grammar are *part-of-speech* tags produced by the morphological analysis. So they include labels like “plural adjective”, “third person noun” etc.

Figure 8 shows the output of FreeLing for a very simple sentence like “Él es ingeniero”<sup>5</sup>.

FreeLing built three subtrees: two noun phrases and a verb. After that, FreeLing just joins them under the fictitious axiom. Figure 5 shows a more complex example.

### *PNEP extension for shallow parsing*

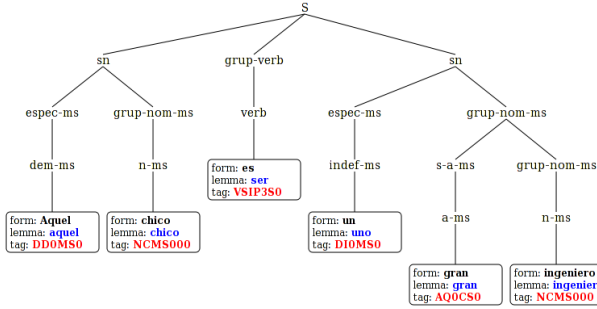
The main difficulty involved in adapting PNEPs to shallow parsing is the fictitious axiom. PNEPs are designed to handle context free grammars that must have an axiom.

We have also found additional difficulties in the way in which FreeLing reduces the number of derivation rules required by its grammar. As we have

---

<sup>5</sup> *He is an engineer*





**Fig. 5.** FreeLing output for “Aquel chico es un gran ingeniero” (*That guy is a great engineer*)

mentioned above, FreeLing uses regular expressions rather than terminal symbols. This kind of rules actually represents a set of rules: those whose terminals match the regular expressions. We have also added this mechanism to PNEPs in the corresponding filters that implement the matching.

In the paragraphs below we will explain both problems in greater detail.

The virtual root node and the partial derivation trees (for the different components of the sentence) force some changes in the behavior of PNEPs. Firstly, we have to derive many trees at once, one for each constituent, instead of only one tree for the complete sentence. Therefore, all the nodes that will apply derivation rules for the nonterminals associated with the components in which the shallow parser is focused will contain their symbol in the initial step. In [23] the nodes of the axiom were the only non empty nodes. More formally:

- Initially, in the original PNEP [23], the only non empty node is associated with the axiom and contains a copy of the axiom. Formally ( $N_A$  and  $\Sigma_N$  stand, respectively, for the node associated with the axiom and the set of nonterminal symbols of the grammar under consideration)

$$I_{N_A} = A$$

$$\forall N_i \in \Sigma_N, i \neq A \rightarrow I_{N_i} = \emptyset$$

- The initial conditions of the PNEP for shallow parsing are:

$$\forall N_i, I_{N_i} = i$$

In this way, the PNEP produces every possible derivation sub-tree beginning from each non-terminal, as if they were axioms of a virtually independent grammar. However, those sub-trees have to be concatenated and then joined



to the same parent node (virtual root node of the fictitious axiom). We get this behavior with splicing rules [8], [18] in the following way: (1) the PNEP marks the end and the beginning of the sub-trees with the symbol %, (2) splicing rules are applied to concatenate couples of sub-trees, taking the beginning of the first one and the end of the second as the splicing point.

To be more precise, a special node is responsible for the first step. Its specification in jNEP is the following:

```
<NODE initCond="">
  <EVOLUTIONARY_RULES>
    <RULE ruleType="insertion" actionType="RIGHT" symbol="%" />
    <RULE ruleType="insertion" actionType="LEFT" symbol="%" />
  </EVOLUTIONARY_RULES>
  <FILTERS>
    <INPUT type="2" permittingContext="SET_OF_VALID_TERMINALS"
      forbiddingContext="" />
    <OUTPUT type="RegularLangMembershipFilter"
      regularExpression="%%.*|%.*%|.%%%" />
  </FILTERS>
</NODE>
```

During the second step the splicing rules concatenate the sub-trees. We could choose a specialized node (just one node) or a set of nodes depending on the degree of parallelism we prefer. The splicing rule required could be defined as follows:

```
<RULE ruleType="splicingChoudhary" wordX="terminal1" wordY="%"
      wordU="%" wordV="terminal2" />
```

Where terminal2 follows terminal1 in the sentence at any place. It should be remembered that % marks the end and beginning of the derivation trees. If the sentence has n words, there are n-1 rules/points for concatenation. It is important to note that only splicing rules that create a valid sub-sentence are actually concatenated.<sup>6</sup>

For example, if the sentence to be parsed is a\_b\_c\_d, we would need the following rules:

```
<RULE ruleType="splicingChoudhary" wordX="a" wordY="%"
      wordU="%" wordV="b" />
<RULE ruleType="splicingChoudhary" wordX="b" wordY="%"
      wordU="%" wordV="c" />
<RULE ruleType="splicingChoudhary" wordX="c" wordY="%"
      wordU="%" wordV="d" />
```

They could concatenate two sub-sentences like b\_c and d, resulting in b\_c\_d.

<sup>6</sup> In fact, we are using Choudhary splicing rules [8] with a little modification to ignore the symbols that belong to the trace of the derivation.



*Our PNEP for the FreeLing's Spanish grammar*

The jNEP configuration file for our PNEP adapted to FreeLing's grammar is large. It has almost 200 hundred nodes and some nodes have dozens of rules. We will show, however, some of its details. Let the sentence to be parsed be "Él es ingeniero". The output node has the following definition:

```
<NODE initCond="">
  <EVOLUTIONARY_RULES>
    <RULE ruleType="deletion" actionType="RIGHT" symbol=""/>
  </EVOLUTIONARY_RULES>
  <FILTERS>
    <INPUT type="RegularLangMembershipFilter"
      regularExpression=
        "%[0-9\\-]* (PP3MS000|PP\\*) [0-9\\-]* (VSIP3S0|VSI\\*)
        [0-9\\-]* (NCMS000|NCMS\\*|NCMS00\\*) %" />
    <OUTPUT type="1" permittingContext=""
      forbiddingContext="PP*_PP3MS000_VSI*_VSIP3S0
        _NCMS*_NCMS00*_NCMS000" />
  </FILTERS>
</NODE>
```

We have explained above that the input sentence includes part-of-speech tags instead of actual Spanish words. This sequence of tags, together with the indexes of the rules that will be used to build the derivation tree, are in the input filter for the output node. We can also see some tags written as regular expressions. We have added this kind of tags because FreeLing also uses regular expressions to reduce the size of the grammar.

As an example, we show the specification of one of the deriving nodes. We can see below that the non-terminal group-verb has many rules. The rule with trace ID 70-7 is the one that is actually needed to parse our example.

```
<NODE initCond="grup-verb" id="70">
  <EVOLUTIONARY_RULES>
    <RULE ruleType="leftMostParsing" symbol="grup-verb"
      string="70-0_grup-ve [...]"
    <RULE ruleType="leftMostParsing" symbol="grup-verb"
      string="70-1_grup-ve [...]"
    <RULE ruleType="leftMostParsing" symbol="grup-verb"
      string="70-7_verb" [...]"
    [...]
  </EVOLUTIONARY_RULES>
  <FILTERS>
    <INPUT type="1" permittingContext="grup-verb"
      forbiddingContext=""/>
  </FILTERS>
</NODE>
```



The output of jNEP is also considerable. However, we can show at least the main dynamics of the process (see Figures 6 and 7). The comments between brackets provide explanations to facilitate understanding.

```
*****NEP INITIAL CONFIGURATION*****
--- Evolutionary Processor 0 ---
[THE INITIAL WORD OF EVERY DERIVATION NODE IS ITS CORRESPONDING
NON-TERMINAL IN THE GRAMMAR]
[...]
--- Evolutionary Processor 70 ---
grup-verb
[...]
--- Evolutionary Processor 112 ---
sn
[...]
--- Evolutionary Processor 190 ---
[THE OUTPUT NODE IS EMPTY]
***** NEP CONFIGURATION - EVOLUTIONARY STEP -
TOTAL STEPS: 1 *****
[FIRST EXPANSION OF THE TREES]
[...]
--- Evolutionary Processor 70 ---
70-6_verb-pass 70-7_verb 70-0_grup-verb_patons_patons_patons[...]
[...]
--- Evolutionary Processor 112 ---
112-104_grup-nom 112-103_grup-nom-ms 112-97_pron-mp 112-95_pron-ns[...]
[...]
***** NEP CONFIGURATION - COMMUNICATION STEP -
TOTAL STEPS: 2 *****
--- Evolutionary Processor 0 ---
[THE FIRST TREES WITH ONLY TERMINALS APPEAR AT THE BEGINNING OF
SPLICING SUB-NET]
--- Evolutionary Processor 178 ---
57-3_NCMS00* 151-35_VSI* 1-2_PP3MS000 99-0_NCMS* 121-2_VSI*
[...]
[THE REST GO TO THE PRUNING NODE]
--- Evolutionary Processor 189 ---
112-87_psubj-mp_indef-mp 8-3_s-a-ms 44-6_prep_s-a-fp [...]
```

**Fig. 6.** jNEP output for “Él es ingeniero”. 1 of 2

As jNEP shows, the output node contains more than one derivation tree. We design the PNEP in this way because ambiguous grammars have more than one possible derivation tree for the same sentence. In this case, our PNEP will produce all the possible derivation trees, while FreeLing is only able to show the most likely.

Figure 8 also clearly corresponds to the output of jNEP when our PNEP is run for shallow parsing.



```

***** NEP CONFIGURATION - COMMUNICATION STEP - TOTAL STEPS: 4 *****
[THE PROCESS OF MARKING THE END AND THE BEGINNING STARTS]
[...]
--- Evolutionary Processor 178 ---
1-2_PP3MS000_% _151-35_VSI* 57-3_NCMS00*_% _1-2_PP3MS000 _%99-0_NCMS* 99-0_NCMS*_%
151-35_VSI*_% 121-2_VSI*_% _%121-2_VSI* _%57-3_NCMS00*
[...]
***** NEP CONFIGURATION - EVOLUTIONARY STEP - TOTAL STEPS: 7 *****
[THE SPLICING SUB-NET STARTS TO CONCATENATE THE SUB-TREES]
[...]
--- Evolutionary Processor 178 ---
156-3_1-2_PP3MS000_% 77-13_57-3_NCMS00*_% _%70-7_151-35_VSI* 34-11_99-0_NCMS*_%
_%111-4_1-2_PP3MS000 111-4_1-2_PP3MS000_% 70-7_151-35_VSI*_% _%77-13_57-3_NCMS00*
_%34-11_99-0_NCMS* _%156-3_1-2_PP3MS000
[...]
--- Evolutionary Processor 187 ---
_%121-2_VSI*_99-0_NCMS*_% _% _%151-35_VSI*_% _%99-0_NCMS*_% _%121-2_VSI*_%
_%151-35_VSI*_99-0_NCMS*_%
--- Evolutionary Processor 188 ---
_%121-2_VSI*_57-3_NCMS00*_% _%151-35_VSI*_57-3_NCMS00*_% _% _%151-35_VSI*_%
_%121-2_VSI*_% _%57-3_NCMS00*_%
[...]
***** NEP CONFIGURATION - COMMUNICATION STEP - TOTAL STEPS: 18 *****
[THE OUTPUT NODE RECEIVES THE RIGHT DERIVATION TREE. IT IS THE SAME AS THE ONE OUTPUT
BY FREELING]
--- Evolutionary Processor 190 ---
[THE FIRST ONE IS THE OUTPUT DESIRED]
_%112-99_111-4_1-2_PP3MS000_70-7_151-35_VSI*_112-103_77-13_57-3_NCMS00*_%
_%1-2_PP3MS000_151-35_VSI*_57-3_NCMS00*_%
[...]

```

Fig. 7. jNEP output for “Él es ingeniero”. 2 of 2

## References

1. R. Adleman. Molecular computation of solutions to combinatorial problem. *Science*, 266:1021–1024, 1994.
2. E. Alfonseca. *An Approach for Automatic Generation of on-line Information Systems based on the Integration of Natural Language Processing and Adaptive Hypermedia techniques*. PhD thesis, Computer Science Department, UAM, 2003.
3. G. Bel Enguix, M. D. Jiménez-López, R. Mercaş and A. Perekrestenko. Networks of evolutionary processors as natural language parsers. In *Proceedings ICAART 2009*, 2009.
4. T. Brants. Tnt—a statistical part-of-speech tagger. In *Proceedings of the 6th Conference on Applied Natural Language Processing*, 2000.
5. J. Castellanos, P. Leupold, and V. Mitran. On the size complexity of hybrid networks of evolutionary processors. *Theoretical Computer Science*, 330(2):205–220, 2005.
6. J. Castellanos, C. Martín-Vide, V. Mitran, and J. M. Sempere. Networks of evolutionary processors. *Acta Informatica*, 39(6-7):517–529, 2003.



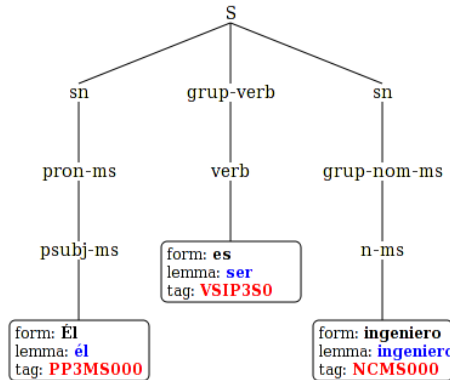


Fig. 8. Shallow parsing tree for “Él es ingeniero”


7. J. Castellanos, C. Martín-Vide, V. Mitrana, and J. M. Sempere. Solving NP-complete problems with networks of evolutionary processors. In *Connectionist Models of Neurons, Learning Processes and Artificial Intelligence: 6th International Work-Conference on Artificial and Natural Neural Networks, IWANN 2001 Granada, Spain, June 13-15, 2001, Proceedings, Part I*, 2001.
8. A. Choudhary and K. Krithivasan. Network of evolutionary processors with splicing rules. *Mechanisms, Symbols and Models Underlying Cognition, PT 1, Proceedings*, 3561:290–299, 2005.
9. E. Csuhaj-Varjú and V. Mitrana. Evolutionary systems: a language generating device inspired by evolving communities of cells. *Acta Informatica*, 36(11):913–926, May 2000.
10. E. Csuhaj-Varjú and A. Salomaa. *Lecture Notes on Computer Science 1218*, chapter Networks of parallel language processors. 1997.
11. J. Earley. An efficient context-free parsing algorithm. *Communications of the ACM*, 13(2):94–102, 1970.
12. C. Martín-Vide, F. Manea and V. Mitrana. Accepting networks of splicing processors: Complexity results. *Theoretical Computer Science*, 371(1-2):72–82, 2007.
13. C. Gómez, F. Javier, D. Valle Agudo, J. Rivero Espinosa, and D. Cuadra Fernández. *Procesamiento del lenguaje Natural*, chapter Methodological approach for pragmatic annotation, pages 209–216. 2008.
14. D. Grune, H.E. Bal, C. Jacobs and Langendoen, K.G. *Modern Compiler Design*. John Wiley and sons, 2002.
15. D. Grune and C. Jacobs. *Parsing Techniques: a Practical Guide*. Springer New York, 2008.
16. Z. S. Harris. *String Analysis of Sentence Structure*. Mouton, The Hague, 1962.





17. F. Manea. Using AHNEPS in the recognition of context-free languages. In *In Proceedings of the Workshop on Symbolic Networks ECAI*, 2004.
18. F. Manea and V. Mitrana. All NP-problems can be solved in polynomial time by accepting hybrid networks of evolutionary processors of constant size. *Information Processing Letters*, 103(3):112–118, July 2007.
19. M. Margenstern, V. Mitrana, and M. J. Pérez-Jiménez. Accepting hybrid networks of evolutionary processors. *DNA Computing*, 3384:235–246, 2005.
20. C. Martín-Vide, V. Mitrana, M. J. Pérez-Jiménez and F. Sancho-Caparrini. Hybrid networks of evolutionary processors. *Genetic and Evolutionary Computation. GECCO 2003, PT I, Proceedings*, 2723:401–412, 2003.
21. A. Mikheev. Periods, capitalized words, etc. *Computational Linguistics*, 28(3):289–318, 2002.
22. R. Mitkov. *The Oxford Handbook of Computational Linguistics*. Oxford University Press, 2003.
23. A. Ortega, E. del Rosal, D. Pérez, R. Mercaş, A. Perekrestenko and M. Alfonso. *PNEPs, NEPs for Context Free Parsing: Application to Natural Language Processing*, chapter Bio-Inspired Systems: Computational and Ambient Intelligence, pages 472–479. LNCS. 2009.
24. S. Seifert and I. Fischer. *Parsing String Generating Hypergraph Grammars*. Springer, 2004.
25. TALP. <http://www.lsi.upc.edu/nlp/freeling/>, 2009.
26. M. Volk. *Introduction to Natural Language Processing*. Course CMSC 723 / LING 645 in the Stockholm University, Sweden., 2004.
27. W. Weaver. *Translation, Machine Translation of Languages: Fourteen Essays*. 1955.
28. A. Zollmann and A. Venugopal. Syntax augmented machine translation via chart parsing. In *Proceedings of the Workshop on Statistic Machine Translation. HLT/NAACL*, New York, June. 2006.





**TRIANGLE** is a periodical published by the Department of Romance Studies. It aims to present the results of interdisciplinary research which adopts new approaches to understanding language sciences.

This volume aims to provide a state-of-the-art of the work recently done, by some relevant Spanish Research Groups, **in the** area of nets of processors.

It could be interesting for a wide spectrum of audience: from mathematicians and linguists to computer scientists that are looking for efficient new models of computation to apply to their problems.

