NETWORKS OF EVOLUTIONARY PROCESSORS: JAVA IMPLEMENTATION OF A THREADED PROCESSOR

Miguel Angel Díaz, Luis Fernando de Mingo López, Nuria Gómez Blas

***Abstract**: This paper is focused on a parallel JAVA implementation of a processor defined in a Network of Evolutionary Processors. Processor description is based on JDom, which provide a complete, Java-based solution for accessing, manipulating, and outputting XML data from Java code. Communication among different processor to obtain a fully functional simulation of a Network of Evolutionary Processors will be treated in future. A safe-thread model of processors performs all parallel operations such as rules and filters. A non-deterministic behavior of processors is achieved with a thread for each rule and for each filter (input and output). Different results of a processor evolution are shown.*

***Keywords**: Networks of Evolutionary Processors, Membrane Systems, Natural Computation.*

***ACM Classification Keywords**: F.1.2 Modes of Computation, I.6.1 Simulation Theory, H.1.1 Systems and Information Theory*

Introduction

A network of evolutionary processors of size n is a construct NEP = $(V, N_1, N_2, \ldots, N_n, G)$, where V is an alphabet and for each $1 \leq i \leq n$, $N_i = (M_i, A_i, PI_i, PO_i)$ is the i-th evolutionary node processor of the network. The parameters of every processor are:

- M_i is a finite set of evolution rules of one of the following forms only:

  $a \rightarrow b$, $a, b \in V$ (substitution rules)

  $a \rightarrow \varepsilon$, $a \in V$ (deletion rules)

  $\varepsilon \rightarrow a$, $a \in V$ (insertion rules)

  More clearly, the set of evolution rules of any processor contains either substitution or deletion or insertion rules.

- A_i is a finite set of strings over V. The set A_i is the set of initial strings in the i-th node. Actually, in what follows, we consider that each string appearing in any node at any step has an arbitrarily large number of copies in that node, so that we shall identify multisets by their supports.
- PI_i and PO_i are subsets of V^*representing the input and the output filter, respectively. These filters are defined by the membership condition, namely a string $w \in V^*$ can pass the input filter (the output filter) if $w \in PI_i$ ($w \in PO_i$).

G = $(N_1, N_2, \ldots, N_n, E)$ is an undirected graph called the underlying graph of the network [Paun, 2002] [Paun, 2000]. The edges of G, that is the elements of E, are given in the form of sets of two nodes. Kn denotes the complete graph with n vertices. By a configuration (state) of an NEP as above we mean an n-tuple C = $(L_1, L_2, \ldots, L_n)$$, with $L_i \subseteq V^*$ for all $1 \leq i \leq n$. A configuration represents the sets of strings (remember that each string

appears in an arbitrarily large number of copies) which are present in any node at a given moment; clearly the initial configuration of the network is $C_0 = (A_1, A_2, \ldots , A_n)$.

A configuration can change either by an evolutionary step or by a communicating step. When changing by an evolutionary step, each component L_i of the configuration is changed in accordance with the evolutionary rules associated with the node i. When changing by a communication step, each node processor N_i sends all copies of the strings it has which are able to pass its output filter to all the node processors connected to N_i and receives all copies of the strings sent by any node processor connected with N_i providing that they can pass its input filter [Manea, 2006] [Martin, 2005] [Garey, 1979].

Theorem 1. A complete NEP of size 5 can generate each recursively enumerable language.

Theorem 2. A star NEP of size 5 can generate each recursively enumerable language.

Theorem 3. The bounded PCP can be solved by an NEP in size and time linearly bounded by the product of K and the length of the longest string of the two Post lists.

Next sections deal with the JAVA implementation of a processor as the first step to achieve a fully functional simulation of NEPs. The non-deterministic behavior of NEPs must be taken into account, that is, a massive parallel implementation is reached having each rule in a thread and each filter in a thread. Objects in processor are locked to avoid mutual exclusion problems due to concurrent programming. [Fahlman, 1983] [Errico, 1994]

JAVA Implementation

NEP processors must behave in a non-deterministic way. Configuration changes are outcome by a communication step or by an evolutionary step, but these two steps are accomplished with no order at all, that is, evolution or communication is chosen depending on the thread model of processor [Diaz, 2007]. Rules and filters (input and output) are implemented as threads extending `Runnable` interface. Therefore a processor is the parent of a set of threads, which use objects in processor in a mutual exclusion region.

Figure 1 shows an UML class diagram corresponding to all classes involved in the definition of a NEP processor. Rules, filters and objects are part of a processor. Filters can be either input filters or output filters, depending on their behavior, controlling how objects are sent or are received by different processors. Substitution rules have an antecedent and a consequent implemented as an object set. When a processor is run through the `start` method, it starts in a cascade way the rule threads and filter threads.

The whole system is prepared to a NEP implementation; only the communication classes must be coded in order to add the communication step to NEP since there exists methods to send a to receive objects in the processor class.

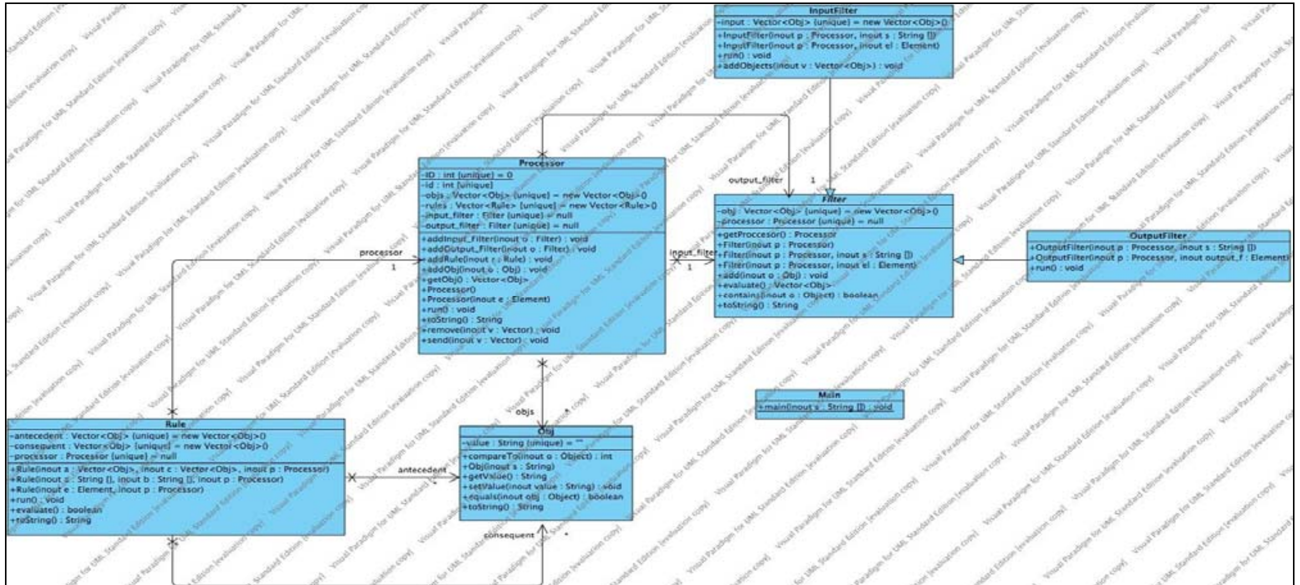

Figure 1.- UML Class diagram of a NEP processor.

This is the basic composition of an evolutionary processor; nevertheless, there exist NEP architectures that have forbidden filters in the input and in the output. Differences in the implementation for the resolution of problems will be defined as types of the generic model for such given kind of problems.

Processors

According to figure 1, each processor has a number of rules, objects and one input filter and one output filter. When the processor thread starts all rule threads are started and input/output threads to, see figure 2 and listing 1. Objects in processor are store using the `Vector` class that is thread-safe, so synchronization is guaranteed.

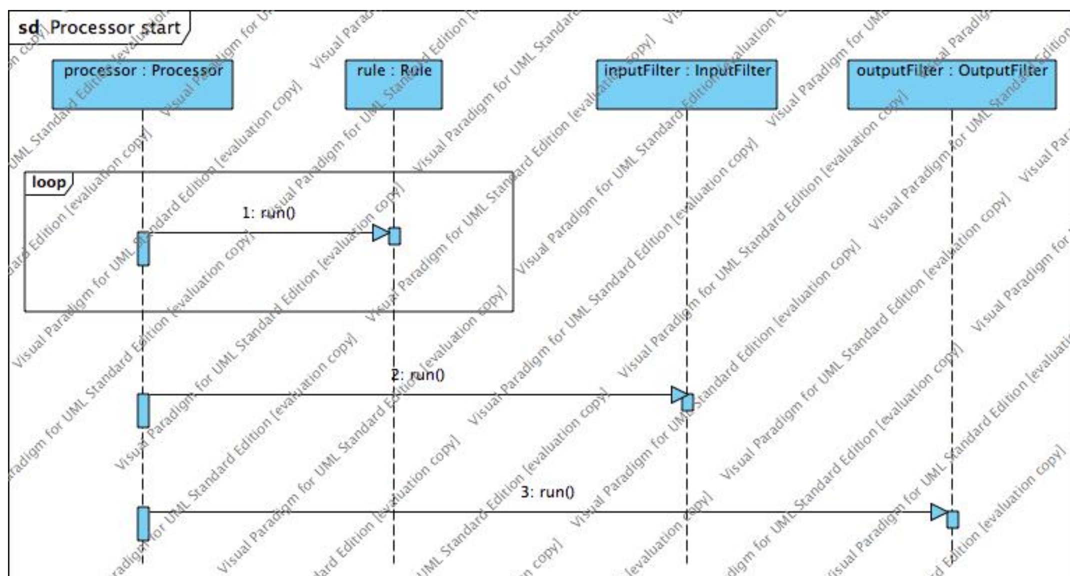

Figure 2.- UML Sequence diagram of processor.

```
public void run() {
    for (int i=0; i<this.rules.size(); i++)
```

```
      new Thread(this.rules.get(i)).start();
     new Thread((OutputFilter) this.output_filter).start();
     new Thread((InputFilter) this.input_filter).start();
     return;
}
```

Listing 1.- Processor behavior schema.

Processors can send and receive objects provided filter constraints are satisfied. This communication is perform in the following way:

- Sending objects. Output filter will check constraints and all objects that satisfy them will be removed from the object pool. In future they will be sent to processors connected to this one.

```
public void run() {
 Vector v = null;
        while (true) {
                v = super.evaluate();
                super.getProccesor().remove(v);
                super.send(v);
        }
        }
```

- Receiving objects. When the method `send` is invoked from another processor, some object will be located in the input filter pool to be checked, if it pass the constraints then it will be added to the processor if it is not present.

```
synchronized public void send(Vector v) {
 ((InputFilter)this.input_filter).addObjects(v);
        }
```

Please note that all rules and filters are threaded, so the non-deterministic behavior is guaranteed.

Rules

Rule threads are quite simple, they only check if the antecedent objects are in the processor object pool, if so objects in consequent are incorporated in processor pool, see figure 3. There is no order when applying rules, they all in separated threads, therefore they all check object pool at the "same time" and they will be applied at the "same time". Some random delay has been incorporated in order to make a more realistic simulation. There are no insertion and deletion rules in our system, but they can be easily added just defining a `null` object in the configuration file `config.xml` and the system will work in such NEP schema.

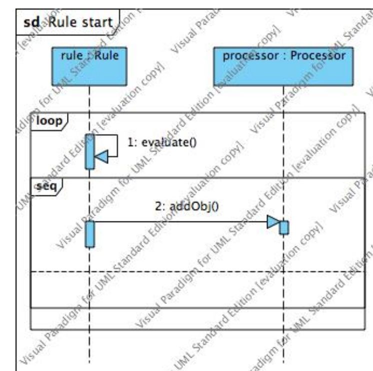

Figure 3.- UML Sequence diagram of rules.

Listing 2 shows an outline of the rule behavior according to figure 3. It can be noted that if the antecedent is satisfactory evaluated then all consequent objects will be added to the processor pool.

```
while(true) {
   if (this.evaluate()) {
     Enumeration<Obj> e = this.consequent.elements();
```

```
    while (e.hasMoreElements()) {
     Obj o = e.nextElement();
     if (!processor.getObj().contains(o)) this.processor.addObj(o);
    }
  }
}
```

Listing 2.- Rule behavior schema.

Filters

Filters are implemented as threads. Therefore they run in parallel with rules. When a filter is satisfied then it will remove or add some objects to the processor pool. Main difference between input and output filters is that, see figure 4:

- Input filter just add objects to processor if they pass the constraints.
- Output filter evaluate object pool to guess which objects must be sent out.

Both of them use the functionality of a `Filter` class, which provides the evaluation of objects, see listing 3.

Several filters can be implemented in the evolutionary processors. A filter is a system that allows a symbol to go from one processor to another. Normally, the detection system is to compare a symbol with another one. Among possible filters that an evolutionary processor can have, most common filters are the PI or input filters, and the PO or output filters. A processor can have several filters of each type. This is the basic composition of an evolutionary processor, nevertheless, there exist NEP architectures that have forbidden filters in the input and in the output. Differences in the implementation for the resolution of problems will be defined as types of the generic model for such given kind of problems.

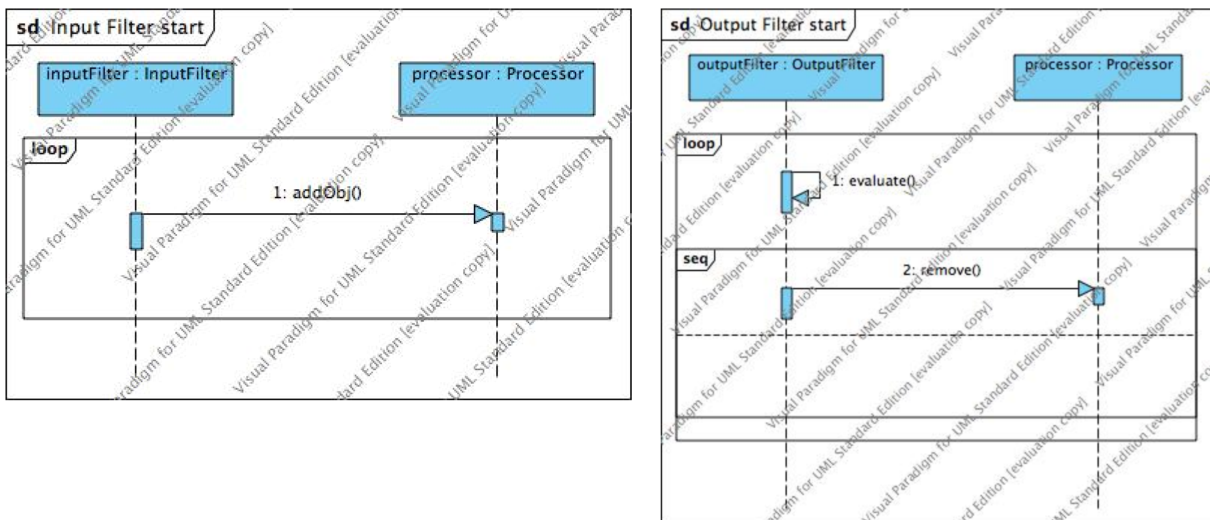

Figure 4.- UML Sequence diagram of Input and Output filters.

Listing 3 shows evaluation method corresponding to output filter to check if objects in pool must be sent out or not to connected processors, if so they will be removed from processor (see sending objects in processor subsection).

```
public Vector<Obj> evaluate() {
```

```
  Vector<Obj> v = this.processor.getObj();
  Enumeration<Obj> e = v.elements();
  v = new Vector<Obj>();
  while(e.hasMoreElements()) {
    Obj o = e.nextElement();
    if (this.obj.contains(o)) if (!v.contains(o)) v.add(o);
  }
  return v;
}
```

Listing 3.- Filter evaluation using object pool.

Configuration

Configuration of a given processor is done with an XML file. In order to parse such configuration JDOM technology is used. There is no compelling reason for a Java API to manipulate XML to be complex, tricky, unintuitive, or a pain in the neck. JDOM is both Java-centric and Java-optimized. It behaves like Java, it uses Java collections, it is completely natural API for current Java developers, and it provides a low-cost entry point for using XML. While JDOM interoperates well with existing standards such as the Simple API for XML (SAX) and the Document Object Model (DOM), it is not an abstraction layer or enhancement to those APIs. Rather, it provides a robust, lightweight means of reading and writing XML data without the complex and memory-consumptive options that current API offerings provide.

Table 1 shows a processor with its corresponding XML configuration. Elements in such XML are: `processor`, `object`, `rule`, `antecedent`, `consequent`, `inputfilter` and `outputfilter`.

Table 1.- XML Configuration corresponding to sample processor in figure.

$\{a, b, c\}$

$\{a,b\} \rightarrow \{d, e\}$

$\{c,d\} \rightarrow \{e\}$

Input Filter $\{c,d\}$ Output Filter $\{c,d,k\}$

```
<NEP>
  <processor>
    <object>a</object>
    <object>b</object>
    <object>c</object>
    <rule>
      <antecedent>
        <object>a</object>
        <object>b</object>
      </antecedent>
      <consequent>
        <object>d</object>
        <object>e</object>
      </consequent>
    </rule>
    <rule>
     <antecedent>
          <object>c</object>
        <object>d</object>
      </antecedent>
      <consequent>
        <object>e</object>
      </consequent>
    </rule>
    <inputfilter>
      <object>c</object>
      <object>d</object>
    </inputfilter>
    <outputfilter>
      <object>c</object>
      <object>d</object>
      <object>k</object>
```

<object>d</object>	</outputfilter>
<object>e</object>	</processor>
	</NEP>

This configuration file is extensible to a NEP system just adding as many processors as needed and adding a XML communication element to define the underlying graph in NEP architecture. As mentioned before, processors have methods to send and to receive objects and only the finalization condition must be taken into account to obtain a fully functional NEP system.

Results: Non-deterministic behavior

This section shows results of evolution corresponding to processor in table 1. When the system starts some threads are create:

- One thread of rule [a, b] --> [d, e]
- One thread of rule [c, d] --> [e]
- One thread for output filter [c, d, k]
- One thread for input filter [c, d]

All four threads have access to objects [a, b, c] in processor. Depending on which thread access the object set it will be modified with new objects (rules) or some objects will be deleted (output filter) or some objects will be added (input filter). The input filter thread controls when new objects are sent from other processor to this one, this case is not yet implemented but it will be in future, when the NEP system will work as a whole not only isolated processors.

Table 2.- Initial and Final configuration of processor in table 1 (one possible evolution).

Initial Configuration	Final Configuration
---------- Processor 1 Rules: [[a, b] --> [d, e], [c, d] --> [e]] Objects: [a, b, c] Output Filter: [c, d, k] Input Filter: [c, d] ----------	---------- Processor 1 Rules: [[a, b] --> [d, e], [c, d] --> [e]] Objects: **[a, b, e]** Output Filter: [c, d, k] Input Filter: [c, d] ----------

Another execution of same processor outputs results in table 3. Please note objects in final configuration, they are different to those in table 2. This is due to the fact that first rule produces objects d and e but output filter was not activated (the processor stops). In table 2, object d is sent out by output filter.

Table 3.- Initial and Final configuration of processor in table 1 (another possible evolution).

Initial Configuration	Final Configuration
---------- Processor 1 Rules: [[a, b] --> [d, e], [c, d] --> [e]]	---------- Processor 1 Rules: [[a, b] --> [d, e], [c, d] --> [e]]

Objects: [a, b, c] Output Filter: [c, d, k] Input Filter: [c, d] ----------	Objects: [a, b, e, d] Output Filter: [c, d, k] Input Filter: [c, d] ----------

If this processor receives an object it will take part of object set. With this implementation the communication and evolution steps have a non-deterministic way, that is, both steps have no priority. In some cases the communication will take place before the evolution and in other cases evolution will take place before communication.

Conclusion and Future Work

This paper has introduced the novel computational paradigm Networks of Evolutionary Processors that is able to solve NP-problems in linear time. The implementation of such model in a traditional computer is being performed and this paper shows an UML architecture. This architecture is a generic representation of a NEP processor behavior. The non-deterministic behavior is performed using JAVA threads accessing the object pool in processor; depending on the Java Virtual Machine a thread will run faster than another. Tables 2 and 3 show such non-deterministic behavior on a given processor.

Future work includes the simulation of a NEP system; only communication must be added to presented model. Such communication will be expressed in the XML configuration file. A separate thread for each processor will be created in final model together with a communication matrix to open communication channels with other processors.

Bibliography

[Diaz, 2007] Miguel Angel Diaz, Miguel Angel Peña, and Luis F. de Mingo: Simulation of Networks of Evolutionary Processors with Filtered Connections. WESAS Transactions on Information, Science and Applications. Issue 3, Vol. 4. ISSN: 1709-0832. Pp.: 608-616. (2007).

[Errico, 1994] L. Errico and C. Jesshope. Towards a new architecture for symbolic processing. Artificial Intelligence and Information-Control Systems of Robots 94, 31–40, World Scientific, Singapore. (1994).

[Fahlman, 1983] S. Fahlman, G. Hinton, and T. Seijnowski. Massively parallel architectures for AI: NETL, THISTLE and Boltzmann machines. Proc. AAAI National Conf. on AI, 1983:109–113, William Kaufman, Los Altos. (1983).

[Garey, 1979] M. Garey and D. Johnson. Computers and Intractability. A Guide to the Theory of NP-completeness. Freeman, San Francisco, CA, (1979).

[Hillis, 1985] W. Hillis. The Connection Machine. MIT Press, Cambridge, (1985).

[Manea, 2006] F. Manea, C. Martın-Vide, and V. Mitrana. All NP-problems can be solved in polynomial time by accepting networks of splicing processors of constant size. Proc. of DNA 12, in press. (2006).

[Martin, 2005] C. Martin-Vide and V. Mitrana. Networks of evolutionary processors: Results and perspectives. Molecular Computational Models: Unconventional Approaches. 78–114, Idea Group Publishing, Hershey. (2005).

[Paun, 2000] Paun G. Computing with Membranes. In: Journal of Computer and Systems Sciences, 61, 1. 108--143. (2000).

[Paun, 2002] Gh. Paun. Membrane Computing. An Introduction, Springer-Verlag, Berlin, (2002).

Authors' Information

Miguel Angel Díaz – Dept. Organización y Estructura de la Información, Escuela Universitaria de Informática, Universidad Politécnica de Madrid, Crta. De Valencia km. 7, 28031 Madrid, Spain; e-mail: mdiaz@eui.upm.es

Luis Fernando de Mingo López – Dept. Organización y Estructura de la Información, Escuela Universitaria de Informática, Universidad Politécnica de Madrid, Crta. De Valencia km. 7, 28031 Madrid, Spain; e-mail: lfmingo@eui.upm.es

Nuria Gómez Blas – Dept. Organización y Estructura de la Información, Escuela Universitaria de Informática, Universidad Politécnica de Madrid, Crta. De Valencia km. 7, 28031 Madrid, Spain; e-mail: ngomez@dalum.eui.upm.es