Implementing Evolutionary Processors in *JAVA*: A case study[*]

M. Campos, J. González, T.A. Pérez and J.M. Sempere

Dept. de Sistemas Informáticos y Computación, Universidad Politécnica de Valencia
Camino de Vera s/n, 46022 Valencia (Spain)
(Tel : + 34 96 387 73 53; Fax :+34 96 387 73 59)
(Email address: {mcampos,jegonzalez,taperez,jsempere}@dsic.upv.es)

Abstract: Networks of Evolutionary Processors are universal models of computation inspired by point mutations in DNA strands performed in nature. These models have been used to solve NP-complete problems efficiently under a parallel computation assumption. Here, we propose a *Java* architecture for the Evolutionary Processors of the network as a first step to obtain a full simulator of the model. We propose Java methods and interfaces in order to obtain a versatile and portable version of the model. In order to show our proposal we will study the case of solving a classical problem in computational complexity theory, the well known *Satisfiability Problem* (SAT).

Keywords: Evolutionary Processors, simulators, OO methods, Java implementation, SAT.

I. INTRODUCTION

Networks of Evolutionary Processors (NEP) were first introduced in [1] as a computation model to solve NP-complete problems. The model is inspired by point mutations in DNA strands (insertion, substitution and deletion of pairs of nucleotides). So, an evolutionary processor can be viewed as a cell with genetic information encoded inside and with some operations that can be performed over them in order to obtain new genetic codes. The network, as presented in [2], is composed by a finite number of evolutionary processors running in an independent way with some communication abilities to send strings from one processor to the rest of processors connected to it. The communication operations include the presence of input and output filters. It has been proved that this model is universal in the sense that it can generate any recursively enumerable language [2]. A modified version of this model was proposed in [3], the so called Hybrid Networks of Evolutionary Processors (HNEP). The main difference between a NEP and a HNEP is the way in which every operation is applied at every processor. In the first model, we have uniform actions (all the deletion and insertion operations are performed in the same positions), while in HNEPs every operation could be different from one processor to the others. HNEPs have been proposed as generating devices (GHNEPs), and accepting ones (AHNEPs). Both proposals have been proved to be universal [3]. A complete survey about NEPs can be viewed in [4].

In this work, we propose a full architecture of classes and interfaces in Java for evolutionary processors as a first step to implement a complete simulator of (Hybrid) Networks of Evolutionary Processors. The advantages of obtaining such a simulator are clear. Observe that NEPs were first proposed to solve NP-complete problems. So, the simulator that we will implement will be an adequate tool to solve such problems by introducing the correct user interface. Furthermore, if we supply a complete benchmark to the simulator for solving any problem, then we can study the problems under a different point of view by observing the work-load at any processor.

Other simulators have been proposed for different computation models based on the cell. We could mention the proposal in [5] where a simulator for P systems is proposed. (P systems [6], also known as membrane systems, are universal models of computation based on the structure of membranes of the living cell.)

In the next section, we will provide the basic notions and definitions of NEPs that we will implement in the simulator.

II. BASIC CONCEPTS AND NOTATION

An Evolutionary Processor is defined as the tuple $\Pi = (M, A, PI, FI, PO, FO)$ where M is a finite set of evolution rules of the following forms only:

[*] Work partially supported by the Spanish Ministry of Education and Science under research project TIN2007-60769.

1. $a \rightarrow b$ (substitution rules)
2. $a \rightarrow \lambda$ (deletion rules)
3. $\lambda \rightarrow a$ (insertion rules)

We will permit the application of insertion and deletion rules at every position of any string.

A is a finite set of strings over the alphabet Σ with multiplicities (i.e. every string has an arbitrarily large number of copies) and *PI* and *PO* are subsets of Σ that denote the input and output permitted contexts of the processor, while *FI* and *FO* (which are again subsets of Σ) are the input and output forbidden contexts of the processor. For a given couple of permitted and forbidden contexts *P* and *F* we will define four ways of running the filters to act over a string *w*. They will be defined as follows (alph(*w*) will denote the set of symbols that *w* contains):

1. $(P \subseteq alph(w)) \wedge (F \cap alph(w) = \emptyset)$
2. $alph(w) \subseteq P$
3. $(P \subseteq alph(w)) \wedge (F \not\subset alph(w))$
4. $(alph(w) \cap P \neq \emptyset) \wedge (F \cap alph(w) = \emptyset)$

In addition, we can assume *P* and *F* to be languages over Σ instead of symbols. Then, the filters can be adapted to keep the membership property instead of the $\subseteq$ and $\cap$ relations.

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

A hybrid network of evolutionary processors is a tuple $\Gamma = (V, G, N, C_0, \beta, i_0)$ where

1. *V* is an alphabet
2. $G=(X_G, E_G)$ is an undirected graph which represents the topology of the network
3. $N: X_G \rightarrow EP_V$ is a mapping that associates to each node an evolutionary processor
4. $C_0 : X_G \rightarrow 2^{V^*}$ is a mapping that associates to every node an initial set of words. Namely, it is the initial configuration of the network.
5. $\beta: X_G \rightarrow \{(1), (2), (3), (4)\}$ defines the type of input/output filters of every processor.
6. $i_0 \in X_G$ is the output node of the network

Observe that in the definition given in [2], the network includes a component α that defines the application of the insertion and deletion rules at the right, left or at any position of any string. Here, we eliminate such component given that the rules will be applied at an arbitrary position.

The running of a NEP starts by an *evolutionary* step where the rules are applied in every processor independently. All the combinations of rule applications are considered due to the presence of multiple copies of every string. Then, a *communication* step is performed, where the strings inside every processor are filtered as outputs and then they are sent out to the rest of processors which are connected to the source one. When the strings arrive at any processor they are filtered as inputs. These two phases are repeated as many times as possible. Finally, when the network stops (i.e. there is no application of rules and communication of strings) the result is collected in the processor i_0.

This model has been proved to be universal. That is, the model is able to recognize or generate any recursively enumerable language [2].

In the following section we will show a proposal to implement every component of the evolutionary processors.

III. IMPLEMENTING EVOLUTIONARY PROCESSORS

Here, we propose an implementation of evolutionary processors in *Java* [7]. This implementation is based on a set of interfaces that define the operativity of the application and the relations between the different classes that we will propose. For every interface we can define different classes, so we have a high degree of flexibility and a robust formalism for every function that the operating units must perform.

For example, the classes **ExecEngineSequential** and **ExecEngineConcurrent** represent a sequential running engine and a concurrent running engine respectively. They share the same interface to communicate with the rest of classes.

The dependencies between classes and interfaces are showed in Figure 1 which represents the diagram of classes in UML.

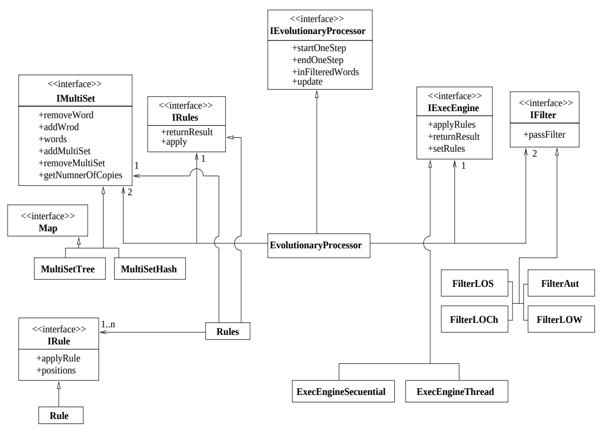

Fig. 1. Diagram of classes in UML

We will explain the diagram of Fig. 1. First, we can see that the application main class is

EvolutionaryProcessor which is implemented by the interface **IEvolutionaryProcessor**. In addition, this class has a set of components which are defined through their interfaces: One or more rules, two filters (one to act over strings and the other that acts as a buffer) and a running engine.

The rules can be of insertion, substitution or deletion. The filters are defined as *lists*, and they can be lists of words (LOW), lists of segments (LOS), lists of characters (LOCh) which can be permitted or forbidden or finite automata (Aut) which define regular languages in order to make membership filters. The multisets have been implemented through the interface **Map<K,V>** in Java. We can use **hashMap<K',V>** or **treeMap<K,V>**.

Finally, for the running engine we have two implementations: one which is sequential and the other which is concurrent but it is based on the sequential engine.

The operability of the evolutionary processor is composed of two phases: First we have a calculation phase which starts with a call to the method **startOneStep** of the evolutionary processor. This method calls to the method **applyRules** which applies the rules which have been previously passed through **setRules**. These rules are applied over the words of the multiset which has been passed as a parameter and, finally, the result is stored. The method of the evolutionary processor **endOneStep**, gets the last stored result and calculates the words which remains in the processor (they will be stored in **buffer**). In addition it calculates the words that will be sent out the processor. We can see the full operability of the evolutionary processor in Fig. 2.

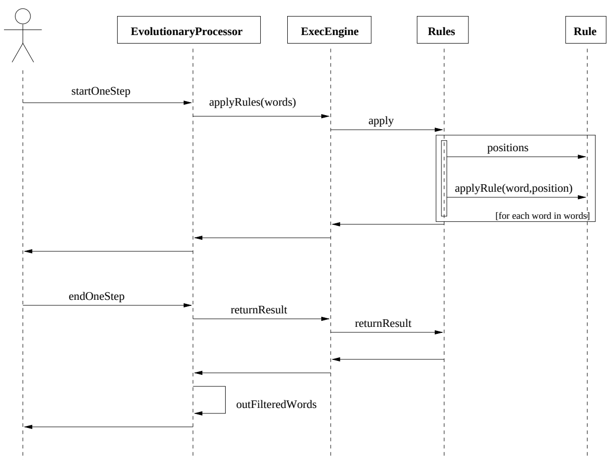

Fig. 2. Operability in the evolutionary processor.

The second phase that we have referred before is the communication phase. After the running of **endOneStep** we have the set of words that will be sent out the processor to the rest of connected processors in the network. In addition, the rest of connected processors in the network will sent their words to the current processor. We will use the method **inFilteredWords** to collect these multisets and we store in **buffer** the words that pass the input filter. Finally the method **update** is called to update the multiset of words with the words stored in **buffer**.

Interfaces and source code

Now we will describe every component of the evolutionary processor defined by its interface. We will explain every interface and its methods.

Rule

We have the following interface to manage a single rule of the processor

```
public interface IRule {
   Vector<String> applyRule(Vector<String> s,
                                   int pos);
    int[] positions(Vector<String> s);
    String getType();
   }
```

The main method is called **applyRule** and it takes a word and one of its positions as parameters. Then, the rule is applied over the word at the specified position and the method returns a new word. In order to calculate the available positions, we use the method **positions** which returns an **array** with all the positions in which we can apply the rule over the word. The method **getType** will be used in different classes to get information about the objects (usually the information returned is the type of the object).

The class **RuleGram** implements this interface. We use it in order to define deletion, substitution or insertion rules. Every rule has two **Strings** (the head and the body) with length greater or equals to zero (which represent the empty string).

Rules

We have the following interface to manage a set of rules of the processor

```
public interface IRules {
      void apply(IMultiset m);
      IMultiset returnResult();
      String getXML();
      String getType();
   }
```

The main method is called **apply** and it takes a multiset as input and then it inspects every word that it contains and it applies an arbitrary rule to every copy of the word. The rest of the words remain stored in an auxiliary multiset that belongs to the running engine. In this phase we use the methods of the rule. We use the methods of the rule **positions** and **applyRule**. This process can avoid the random component to select rules in the case that we have an infinite number of rules given that we can apply all the available rules over the words and we can obtain all the combinations with an infinite number of results. The method **returnResult** allows accessing the auxiliary multiset to obtain the result of the calculation step. The method **getXML** obtains the corresponding XML scheme and, finally, the method **getType** will be used in different classes to get information about the objects. In this case, it returns the type of rules that we are applying..

Filters

The input and output filters of the processor are managed through the following interface

```
public interface IFilter {
      boolean passFilter(Vector<String> s);
      String getXML();
       String getType();
}
```

The main method, **passFilter**, tells whether a word pass the filter or not, while **getType** returns the type of the filter.

The class **FilterLOW** implements this interface and represents the filter which is defined by a list of strings. In order to pass the filter a membership query is performed. If a given word belongs to the vector of strings then it passes the filter, otherwise it does not. In order to create the filter, we impose a lexicographic order in the elements of the vector. So, when we are interested to test whether a word belongs to the vector or not, we employ a logarithmic time process instead a linear one.

There are three more available implementations, (**filterLOS**, **filterLOCh** and **filterAut**). They represent filters that contain segments or characters which can be permitted or forbidden or a finite automaton in order to define membership filters. We can manage four different ways of running the filters. See the definitions in section 2. We will use *segm*(*w*) instead of *alph*(*w*) when we apply **filterLOS**. Here, *segm*(*w*) represents the list of segments of *w*.

We can observe that, when we use **filterLOW** there is only one way of running.

Multisets

The multisets that appear in the evolutionary processors can be managed through the following interface

```
import java.util.Iterator;
import java.util.Vector;
public interface IMultiSet {
  void addMultiSet(IMultiSet m);
  int getNumberOfCopies(Vector<String>
                                   word);
  void addWord(Vector<String> word,
                          Integer copies);
  void removeWord(Vector<String> word);
  Iterator<Vector<String>> words();
  void removeMultiSet(IMultiSet ms);
  String getType();
  int size();
  String getXML();
}
```

This interface represents a multiset of words and it includes the set of basic functions to manage it. In order to add a word to a multiset we use **addWord**. Here we pass a word and its number of copies (the number of copies must be greater than 0 and we use -1 to represent an infinite number of copies). The method **removeWord** erases every copy of a given word. **addMultiSet** adds a multiset to the current multiset (if they have common words then it sums the number of copies) and **removeMultiSet** deletes all the words contained in the parameter multiset from the current multiset. In addition, we have two more methods which allows the inspection of a multiset. They are **words**, that obtains an iterator which contains every word, and **getNumberOfCopies** which tells the number of copies of every word in the multiset. Finally, to get the information of a multiset we have the methods **getType** which returns the type of the multiset and **size** which tells the number of words that it contains.

This interface has two different implementations: we can use **MultiSetHash** or **MultiSetTree**. Both classes depend from the interface **Map** and the unique difference between them is the concrete implementation from the interface **Map**. So, **MultiSetHash** uses a **HashMap**, while **MultiSetTree** uses a **TreeMap**. We will use these data structures to store every word (which acts as a key) and the number of its copies.

The running engine

The following interface implements the running engine

```
public interface IExecEngine {
        void applyRules(IMultiSet m);
        IMultiSet returnResult();
        void setRules(IRules r);
}
```

The running engine is one of the main components of the implementation given that it will support most of the work. So, an efficient implementation is fundamental to obtain an application with a high performance efficiency. First, we must specify the rules to be applied before calling any calculation step. In order to make so, we use the method **setRules**. Then, we can start the calculation process which is divided into two different phases. In the first phase, **applyRules** calls the method **apply** of the class **Rules**. The method **returnResult** calls the method returnResult of the class **Rules**.

Evolutionary Processor

The following interface implements the evolutionary processor.

```
public interface IEvolutionaryProcessor {
        void startOneStep();
        IMultiSet endOneStep();
        void inFilteredWords(IMultiSet m);
        void update();
        Object info(String t);
        IMultiSet getState();
        IRules getRules();
        IFilter getOutputFilter();
        IFilter getInputFilter();
        String getId();
        String getEngine();
}
```

This is the main interface of the processor given that it fully defines the operability of the application. The calculation step is implemented through two methods: **startOneStep** calls to the running engine method **applyRules**, and sends the multiset of words of the processor as a parameter. The second method, **endOneStep**, calls to the method **returnResult**, and then it returns a multiset. The method takes the last multiset and evaluates the words that pass the output filters, it returns these words. The rest of the words are stored in an auxiliary multiset. The communication is performed between **returnResult** (the words that are sent out) and **inFilteredWords** which takes a multiset (coming from a different processor) and inspects the words that can pass the input filter. These words are stored in the auxiliary multiset. After the running of the calculation and communication phases we have stored the words that must be kept in the processor inside the auxiliary multiset. In order to update the state of the processor we will use the method **update**. The rest of methods of this interface have been designed to get the information about the processor. We can obtain the general information by using **info**, or we can use concrete methods such as **getState, getRules, getOutputFilter, getInputFilter, getId** and **getEngine**.

In order to store the descriptions of the processor and the results that it produces, we have decided to use XML files. This format allows a high compatibility and power to make descriptions. An additional advantage to use XML is that it is a well known standard with fully implemented parsers and it allows an easy definition of the structure of labels to be used. Whenever we wish to run the application, we must pass the location of the files which contains the definitions of the different processors that we want to use. We have defined a XML scheme for every component of the processor.

IV. SOLVING THE *SAT* PROBLEM

The *Satisfiability Problem* (SAT) is addressed as the first problem to be NP-complete as it was proved by SA Cook [8]. The problem can be enunciated as follows: "*Given a finite set of boolean variables and a finite set of clauses over the variable. Is there any assignment for the variables that holds every clause true* ?"

Here, we propose a polynomial time solution for the SAT problem by using a NEP of $2n + 2$ processors, where n is the number of boolean variables of the SAT instance. In the following, we give a formal definition of the NEP that we propose to solve SAT.

The SAT instance can be encoded as a finite string $x=<Q_1><Q_2>\ldots<Q_3>$ where $Q_j=b_1b_2\ldots b_k$,. is the j^{th} clause and b_p is a literal (that is the boolean variable or its negated version). Let $T=\{a_1, a_2, \ldots, a_n\}$ be the set of the boolean variables of the SAT instance and $T_{not}=\{\neg a_1, \neg a_2, \ldots, \neg a_n\}$ be the corresponding negated variables. We will consider the set $C=\{<, >, t, f\}$ and $V=T\cup T_{not}\cup C$. The network that we propose is composed of 2n + 2 processors, namely N_{in}, N_{out}, N_{1t}, $N_{2t}, \ldots, N_{nt}$, N_{1f}, N_{2f}, …, N_{nf}. Here, N_{in} is the input processor and N_{out} the output processor. The definition of every processor is as follows, observe that we will define every processor by the tuple (A,B,C,D) where A is the finite set of evolution rules, B is the finite set of initial strings in the processors, C is a language for the input filter and D is a language for the output filter (here we will use the class **FilterLOW**)

$N_{in}=(\varnothing, x, \varnothing, V^*)$

$N_{out}=(\varnothing, \varnothing, C^*\text{-}C^*<f^+>C^*, \varnothing)$

$(\forall\ i\ 1 \le i \le n)$

$N_{it}=(\{a_i\to t; \neg a_i\to f\}, \varnothing,$
$\quad V^*(a_i,\neg a_i)V^*, V^*\text{-}V^*(ai,\neg a_i)V^*)$

$(\forall\ i\ 1 \le i \le n)$

$N_{if}=(\{a_i\to f; \neg a_i\to t\}, \varnothing,$
$\quad V^*(a_i,\neg a_i)V^*, V^*\text{-}V^*(ai,\neg a_i)V^*)$

The topology of the network will be a complete graph.

The network works as follows: Initially we introduce the encoded instance x in the input processor N_{in}. Then, given that the output filter is V^*, the string x is communicated to the rest of the evolutionary processors and it is stored inside them with the exception of the output processor N_{out} due to its input filter. Observe that, for any processor N_{it} or N_{if}, the string *x* will pass the input filter if the encoded instance has an appearance of the variable a_i or its negated version $\neg a_i$. In the following evolutionary step the encoded instance is assigned with true or false values for every variable. Observe that it is performed in any processor N_{it} or N_{if} with their corresponding substitution rules. Then, the assigned string is communicated to the rest of processors and the assignment process is performed in various steps as described before. Once the string has all its variables with an assignment value then it is communicated to the output node and it will pass its input filter if and only if all the clauses are satisfied.

Some remarks:

(1) The network assigns to the variables the same value in different clauses. That is, if we assign the true value to the variable a_i then it is reflected in any clause (observe that a string only leaves the processor N_{it} or N_{if} if the variable a_i has an assigned value at any clause due to its output filter).

(2) The time complexity is polynomial with the size of variables and clauses. Observe that we make all the possible assignments in linear time (depending on the number of clauses) and we communicate the strings in linear time (depending on the number of variables).

(3) All the combinations of value assignments to the variables are tested due to the topology of the graph, which is complete.

V. CONCLUSION

We have proposed a full architecture of methods and classes in Java in order to implement Evolutionary Processors. This is the first step towards a full simulator of the NEP model. As an example we have proposed a solution for the SAT problem that holds the implementation that we have proposed.

Actually we are performing a set of experiments in order to test our implementation for the solution of the SAT problem. The results will be reported in future works. In addition, we are developing a user interface in order to complete a full simulator for the NEP model. Again, it will be reported in future works.

REFERENCES

[1] Castellanos J, Martín-Vide C, Mitrana V, Sempere. JM (2001) Solving NP-complete problems with networks of evolutionary processors. Proceedings of IWANN 2001, LNCS 2084, pp 621-628. Springer-Verlag.

[2] Castellanos J, Martín-Vide C, Mitrana V, Sempere JM (2003) Networks of evolutionary processors. Acta Informatica 39: 517-529

[3] Martín-Vide C, Mitrana V, Pérez-Jiménez M, Sancho-Caparrini F (2003) Hybrid Networks of Evolutionary Processors. Proceedings of GECCO 2003, LNCS 2723, pp 401-412. Springer.

[4] Martín-Vide C, Mitrana V (2005) Networks of Evolutionary Processors: Resuls and Perspectives. Molecular Computational Models. Unconventional Aproaches. (M. Gheorghe, editor). Idea Group Publishing. 2005.

[5] Arroyo F (2004) Structures and Biolanguage to Simulate Membrane Computing (in Spanish). PhD Thesis. Universidad Politécnica de Madrid (Madrid, Spain)

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

[7] Horstmann CS, Cornell G (2005) Core Java (Vols 1 and 2) Prentice Hall

[8] Cook, SA (1971) The Complexity of Theorem Proving Procedures. Proceedings Third Annual ACM Symposium on Theory of Computing pp 151-158