Triangle: Language, Literature, Computation, n. 7, 2012
Publicacions Universitat Rovira i Virgili · ISSN: 2013-939X
https://revistes.urv.cat/index.php/triangle

Developing Tools for Networks of Processors [*]

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

[1] 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

[2] 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.

[*] 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 attemps 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 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:

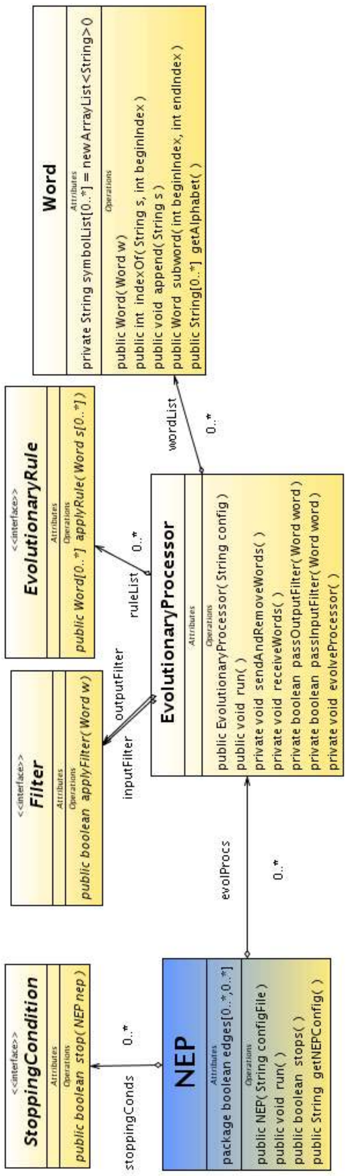

Word
Attributes
private String symbolList[0..*] = new ArrayList<String>()
Operations
public Word(Word w)
public int indexOf(String s, int beginIndex)
public void append(String s)
public Word subword(int beginIndex, int endIndex)
public String[0..*] getAlphabet( )
<<interface>>
EvolutionaryRule
Attributes
Operations
public Word[0..*] applyRule(Word s[0..*])
wordList
0..*
ruleList
0..*
<<interface>>
Filter
Attributes
Operations
public boolean applyFilter(Word w)
outputFilter
inputFilter
EvolutionaryProcessor
Attributes
Operations
public EvolutionaryProcessor(String config)
public void run( )
private void sendAndRemoveWords( )
private void receiveWords( )
private boolean passOutputFilter(Word word)
private boolean passInputFilter(Word word)
private void evolveProcessor( )
evolProcs
0..*
<<interface>>
StoppingCondition
Attributes
Operations
public boolean stop(NEP nep)
stoppingConds
0..*
NEP
Attributes
package boolean edges[0..*,0..*]
Operations
public NEP(String configFile)
public void run( )
public boolean stops( )
public String getNEPConfig( )

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 *Word*s 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: *Thread*s and *Process*es. 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 *Process*es 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 Thread*s 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, *Thread*s 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] ::= λ
- [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"/> [conditionTag]
- [conditionTag] ::= <CONDITION type="NonEmptyNodeStoppingCondition" nodeID="[integer]"/> [conditionTag]
- [conditionTag] ::= λ

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] [nodeFiltersTag] </NODE> [nodeTag]
- [nodeTag] ::= λ
- [auxWordList] ::= auxiliaryWords="[wordList]" | λ
- [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= "[symbolList]" wordU= "[symbolList]" wordV= "[symbolList]"/> [ruleTag]
- [ruleTag] ::= λ
- [ruleType] ::= insertion | deletion | substitution

- [actionType] ::= LEFT | RIGHT | ANY
- [nodeFiltersTag] ::= [inputFilterTag] [outputFilterTag]
- [nodeFiltersTag] ::= [inputFilterTag]
- [nodeFiltersTag] ::= [outputFilterTag]
- [nodeFiltersTag] ::= λ
- [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] ::= λ
- [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.

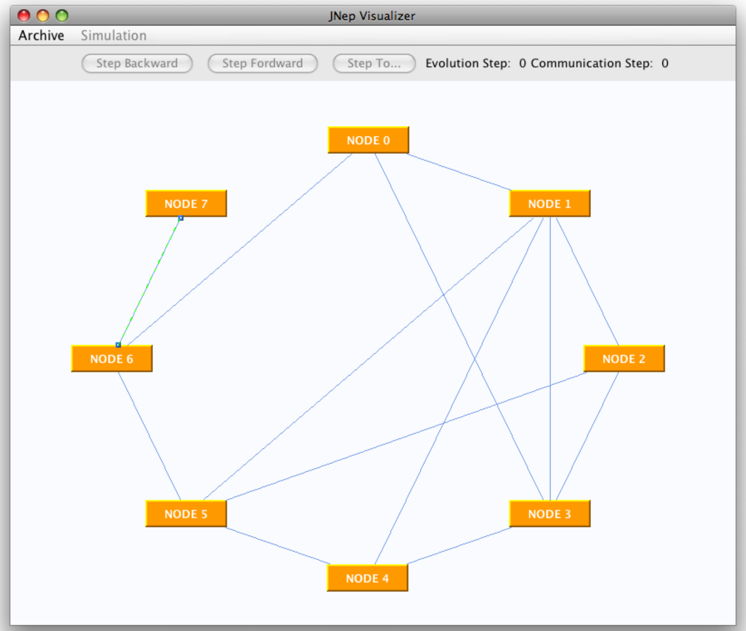


Fig. 2. Window that shows the layout of the simulated NEP

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

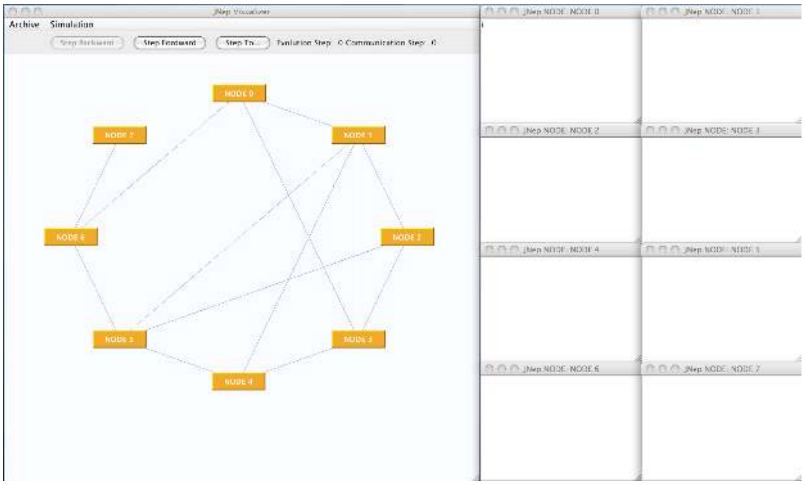


Fig. 3. Initial simulation step

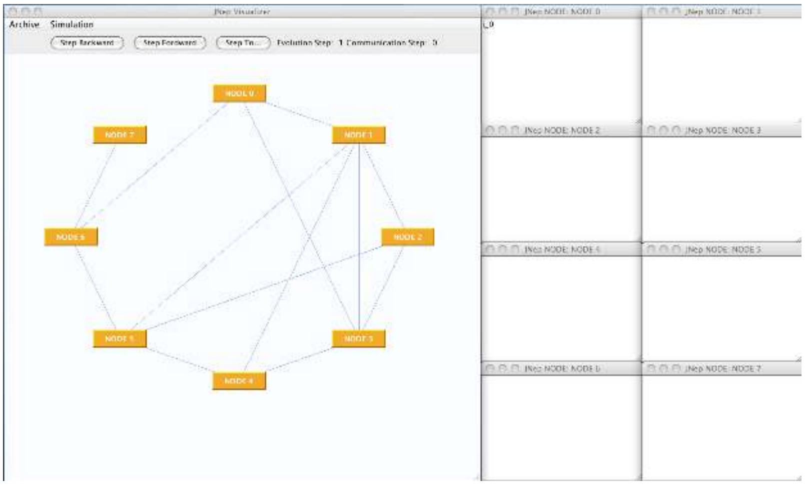
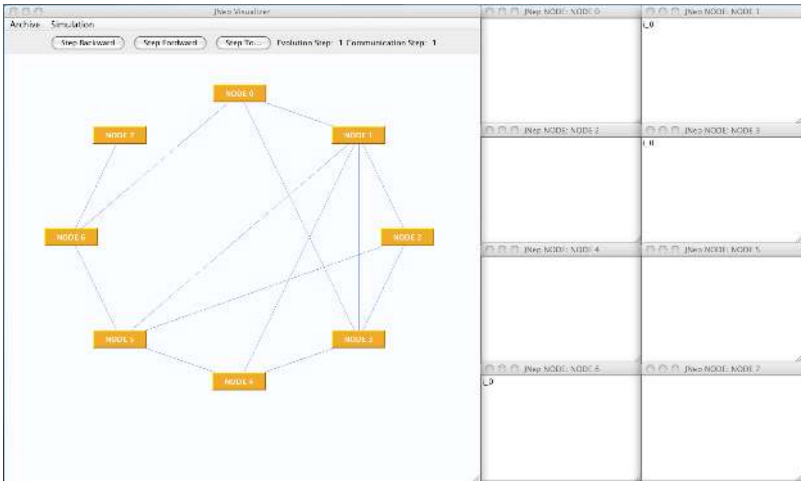


Fig. 4. Next simulation step

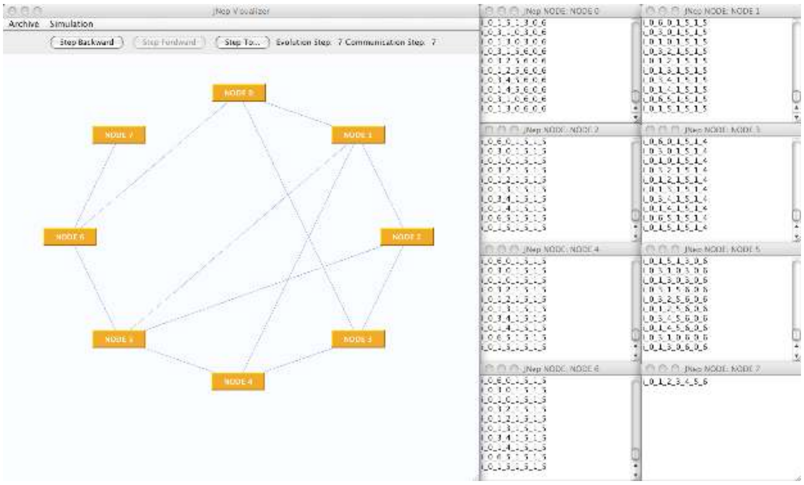
heterogeneous parallel environments is of general significance across a wide array of problems.

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



JNep NODE: NODE 1
```
i_0
```
JNep NODE: NODE 3
```
i_0
```
JNep NODE: NODE 6
```
i_0
```

Fig. 5. Second simulation step



JNep NODE: NODE 0
```
i_0_1_3_1_3_0_6
i_0_3_1_0_3_0_6
i_0_1_3_0_3_0_6
i_0_3_1_3_6_0_6
i_0_1_2_5_6_0_6
i_0_1_2_3_6_0_6
i_0_1_4_5_6_0_6
i_0_1_4_3_6_0_6
i_0_3_1_0_6_0_6
i_0_1_3_0_6_0_6
```
JNep NODE: NODE 1
```
i_0_6_0_1_5_1_5
i_0_3_0_1_5_1_5
i_0_1_0_1_5_1_5
i_0_3_2_1_5_1_5
i_0_1_2_1_5_1_5
i_0_1_3_1_5_1_5
i_0_3_4_1_5_1_5
i_0_1_4_1_5_1_5
i_0_6_5_1_5_1_5
i_0_1_5_1_5_1_5
```
JNep NODE: NODE 2
```
i_0_6_0_1_5_1_5
i_0_3_0_1_5_1_5
i_0_1_0_1_5_1_5
i_0_1_2_1_5_1_5
i_0_1_2_1_5_1_5
i_0_1_3_1_5_1_5
i_0_3_4_1_5_1_5
i_0_1_4_1_5_1_5
i_0_6_5_1_5_1_5
i_0_1_5_1_5_1_5
```
JNep NODE: NODE 3
```
i_0_6_0_1_5_1_4
i_0_3_0_1_5_1_4
i_0_1_0_1_5_1_4
i_0_3_2_1_5_1_4
i_0_1_2_1_5_1_4
i_0_1_3_1_5_1_4
i_0_3_4_1_5_1_4
i_0_1_4_1_5_1_4
i_0_6_5_1_5_1_4
i_0_1_5_1_5_1_4
```
JNep NODE: NODE 4
```
i_0_6_0_1_5_1_5
i_0_3_0_1_5_1_5
i_0_1_0_1_5_1_5
i_0_1_2_1_5_1_5
i_0_1_2_1_5_1_5
i_0_1_3_1_5_1_5
i_0_3_4_1_5_1_5
i_0_1_4_1_5_1_5
i_0_6_5_1_5_1_5
i_0_1_5_1_5_1_5
```
JNep NODE: NODE 5
```
i_0_1_5_1_3_0_6
i_0_3_1_0_3_0_6
i_0_1_3_0_3_0_6
i_0_3_1_5_6_0_6
i_0_3_2_5_6_0_6
i_0_1_2_5_6_0_6
i_0_3_4_5_6_0_6
i_0_1_4_5_6_0_6
i_0_3_1_0_6_0_6
i_0_1_3_0_6_0_6
```
JNep NODE: NODE 6
```
i_0_6_0_1_5_1_5
i_0_3_0_1_5_1_5
i_0_1_0_1_5_1_5
i_0_1_2_1_5_1_5
i_0_1_2_1_5_1_5
i_0_1_3_1_5_1_5
i_0_3_4_1_5_1_5
i_0_1_4_1_5_1_5
i_0_6_5_1_5_1_5
i_0_1_5_1_5_1_5
```
JNep NODE: NODE 7
```
i_0_1_2_3_4_5_6
```

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 $\{i, 0, 1, ..., n\}$ 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.

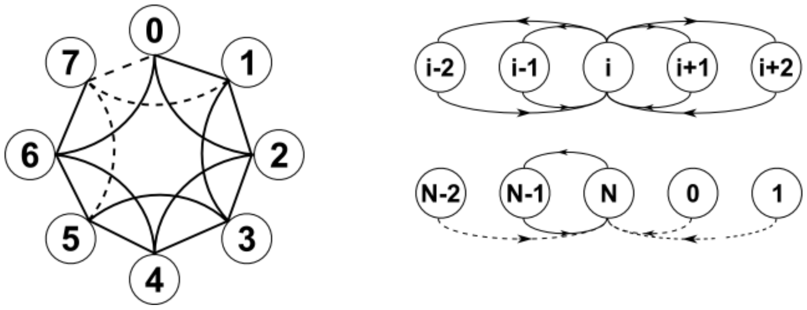


Fig. 7. Example of a NEP with n=6 and the extra one to collect the strings

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 ω 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 ω 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.

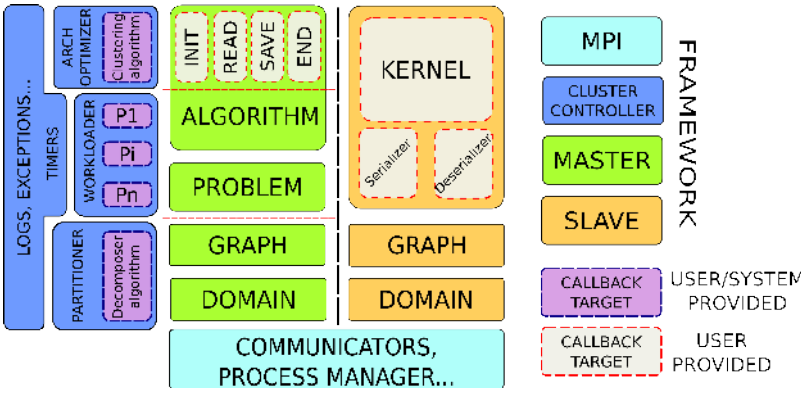


Fig. 8. Framework architecture implemented to run sequential code as parallel in a cluster of computers.

Both master and slave processes work with graph stuctures so, the master translates the information given by the user into a graph and descomposes 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 synchronization 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 $4x10^3$ valid strings) to $n = 24$ ($5x10^5$ 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.

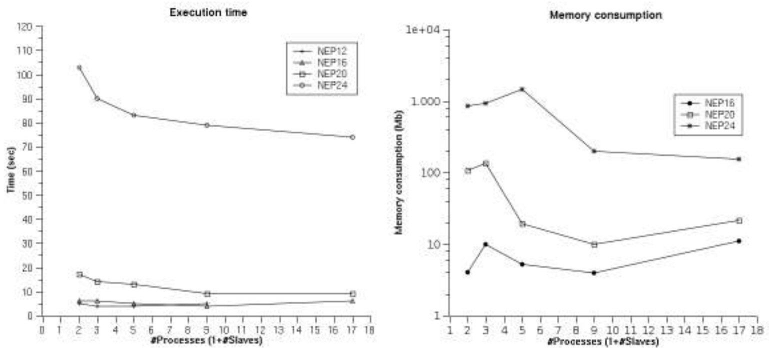


Fig. 9. a) Execution time running a sequential NEP algorithm in parallel using the framework described. The performance of the application was better with the NEP algorithm than with the single slave execution. b) Semilog plot for the memory consumption running the NEP algorithm under the framework.

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

NEPvl

Introduction to Domain Specific Visual Languages and AToM^3

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^3 [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 DSVL to design Networks of Evolving Processors (NEPs). For this purpose, we built a meta-model in the AToM^3 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*^{3} 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:

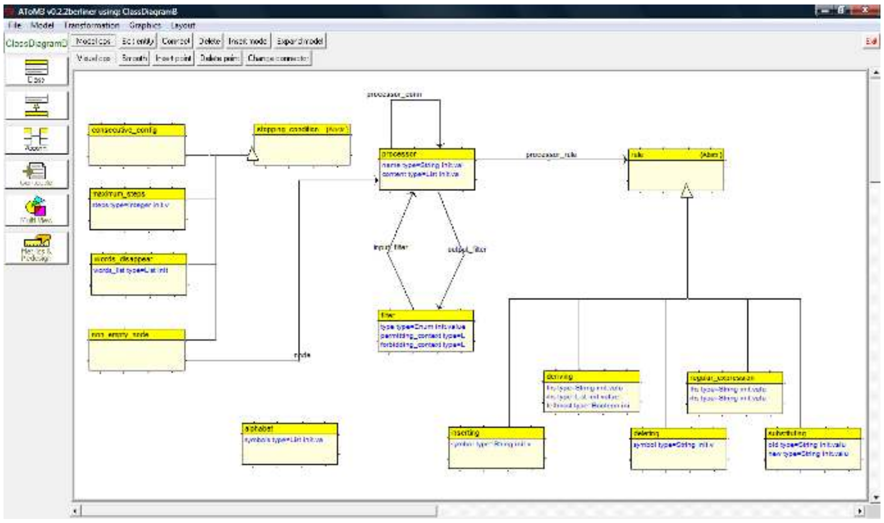


Fig. 10. The meta-model UML class diagram

- 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 ouput 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 $ATоM^3$, 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 NEPvl 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.

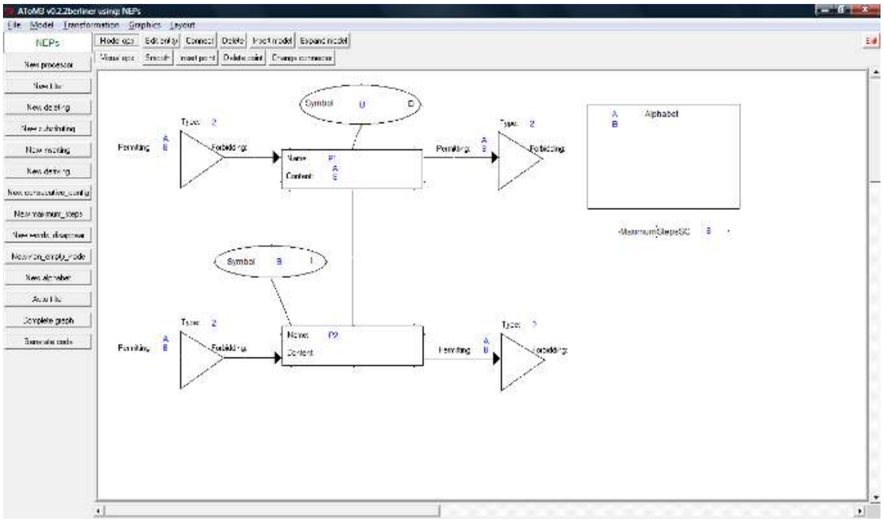


Fig. 11. Example of graphic representation of a NEP

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 NEPvl. 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 [3] 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 $\{\wedge(power), +, -, *, /\}$

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.

[3] 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 $\{m_{0,a}, m_{0,b}, m_{0,c}, \ldots m_{10,a}, m_{10,b}, m_{10,c}\}$.

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 ouput 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_emtpy_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 NEPVl 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.

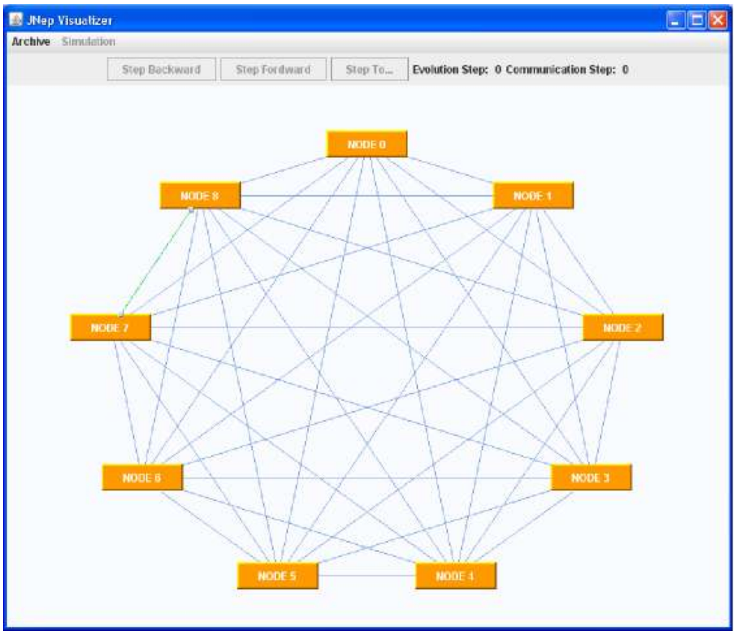


Fig. 12. jNEPview window showing the complete graph of a NEP with 9 processors.

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,O} // Alphabet
@N=  {final}+ {n{symbol}:symbol->{X,S,O}} /* Nodes associated
with non terminal symbols */
@c{n{X}}={X}  // Initial content of the axiom node
@r{n{X}}= {X-->SO} // Deriving rules for the axiom
@r{n{S}}= {S-->aSb, S-->ab}
@r{n{O}}= {O-->o, O-->oO, O-->Oo}
@C=@complete  // The graph is complete
@S={ @non_emtpy_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. Hochleistungsrechenszentrum bayern, http://www.lrz.de.
2. http://jgrapht.sourceforge.net/.
3. http://wwwipd.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. Perekrestenko. 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^3: 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 Martin-Vide Florin Manea and Victor Mitrana. 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 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.
29. M. Margenstern, V. Mitrana, and M. J. Perez-Jimenez. Accepting hybrid networks of evolutionary processors. *DNA Computing*, 3384:235–246, 2005.
30. C. Martín-Vide and V. Mitrana. Solving 3cnf-sat and hpp in linear time using www. *Machines, Computations, and Universality*, 3354:269–280, 2005.
31. C. Martín-Vide, V. Mitrana, 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 *Proccedings of the Workshop on Statistic Machine Translation.* HLT/NAACL, New York, June. 2006.