First International School on Biomolecular and Biocellular Computing (ISBBC'11)

Osuna 6/9/2011

Dr. Alfonso Ortega, Dept. Ingeniería Informática Universidad Auntónoma de Madrid
alfonso.ortega@uam.es

Networks of Evolutionary Processors (NEPs)

NEPs computers

An overview of a possible architecture for NEPs computers

- **Layout of the architecture**
- Solving an instance of the Hamiltonian path problem with NEPs
- Programming tools for NEPs
  - Graphic simulation environment on ‘classical’ architectures
  - Simulation on clusters of computers
  - Textual programming languages independent on the architecture
- Some applications
  - Solving NP-problems with lineally bounded resources
  - Some applications of NEPs to language processing

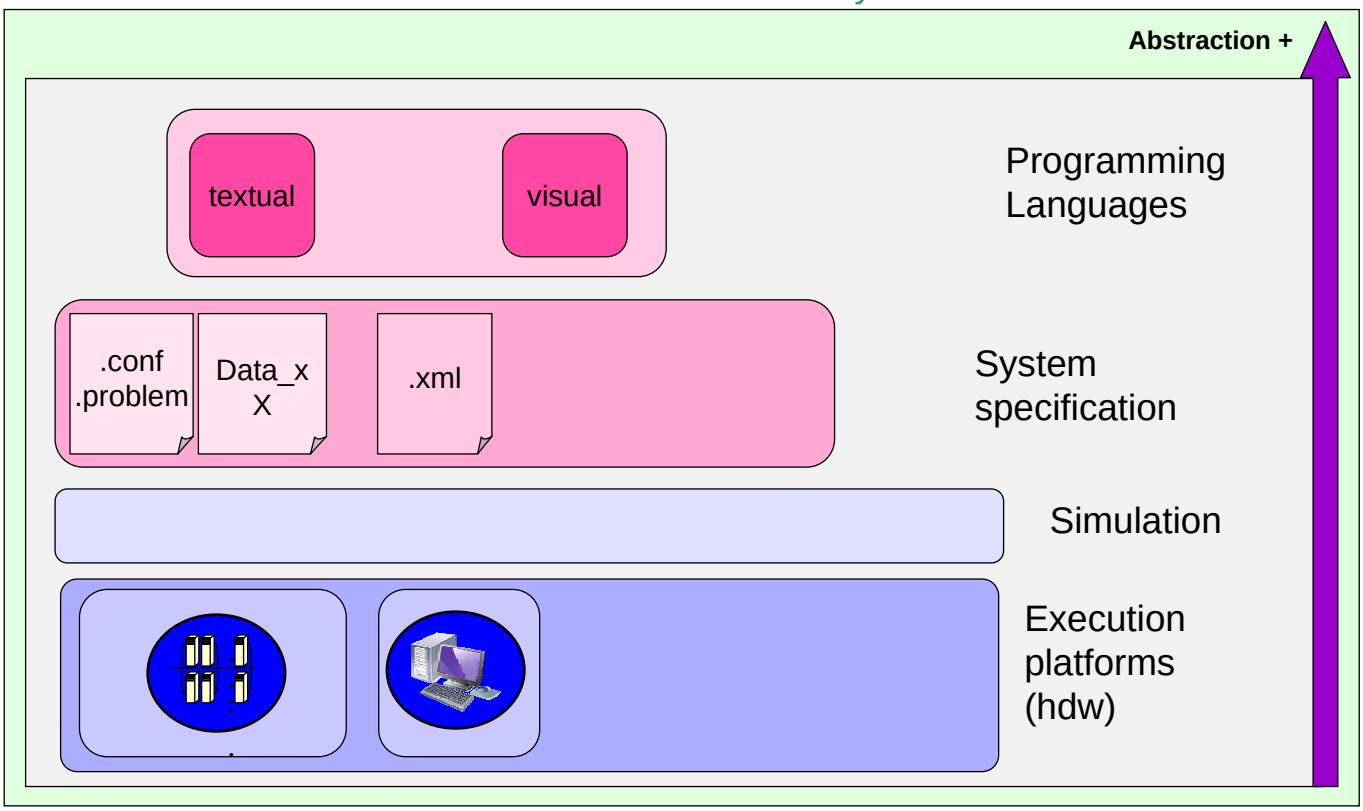
NEPs computers : Layout of the architecture

Motivation

- NEPs are one of the natural computers currently not supported by any real hardware platform
- NEPs are theoretically able to solve NP-problems with polynomial resources
- NEPs have to be currently simulated on ‘conventional’ computers loosing their potential advantage
- There are almost any tool for considering NEPs an alternative to von Neumann architectures
  - Suitable hardware/software platforms to run them
  - Programming languages and their processors (compilers)
  - Software developing tools for NEPs
- In this part of the course we will show our approach to overtake these drawbacks

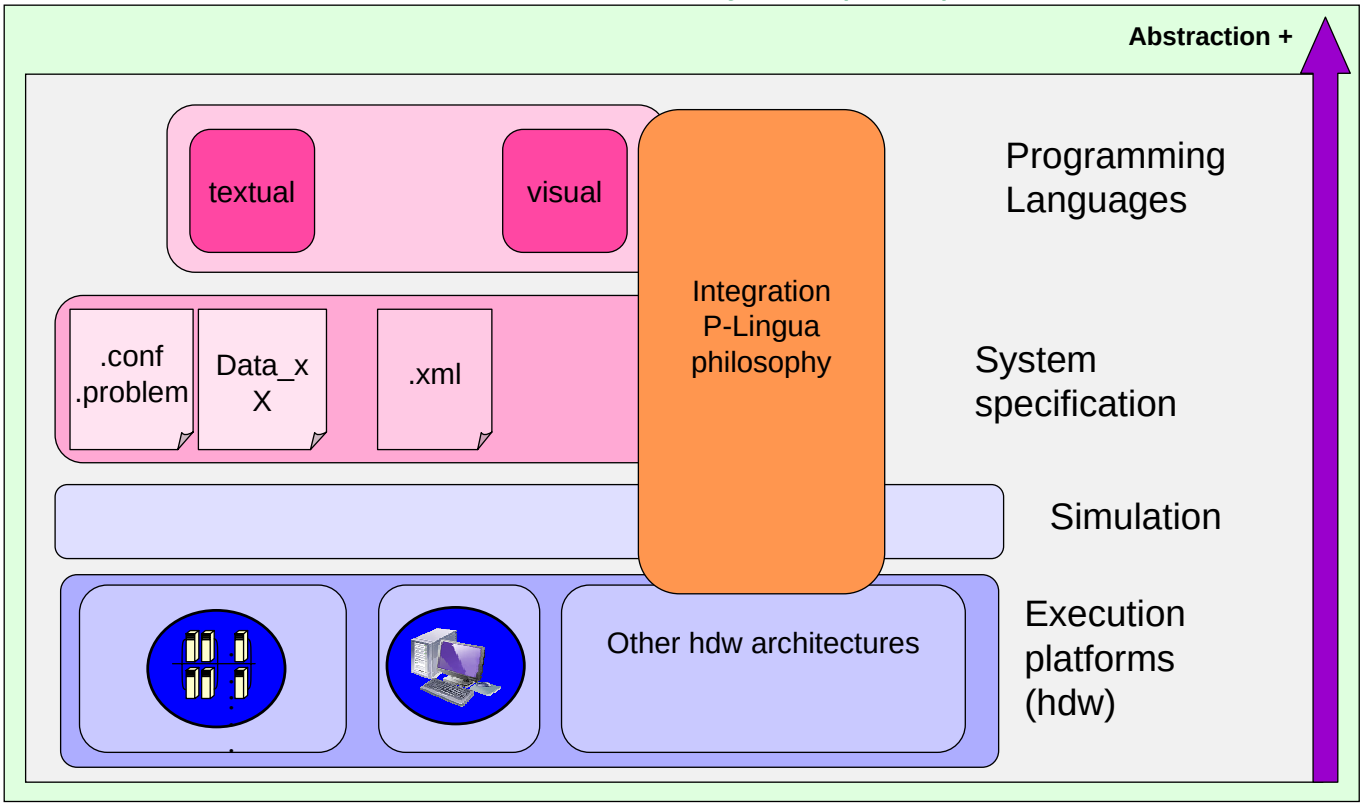
NEPs computers: Layout of the architecture

Structure of current the system



NEPs computers: Layout of the architecture

Structure of the system (future)



NEPs computers

An overview of a possible architecture for NEPs computers

- Layout of the architecture
- **Solving an instance of the Hamiltonian path problem with NEPs**
- Programming tools for NEPs
  - Graphic simulation environment on ‘classical’ architectures
  - Simulation on clusters of computers
  - Textual programming languages independent on the architecture
- Some applications
  - Solving NP-problems with lineally bounded resources
  - Some applications of NEPs to language processing

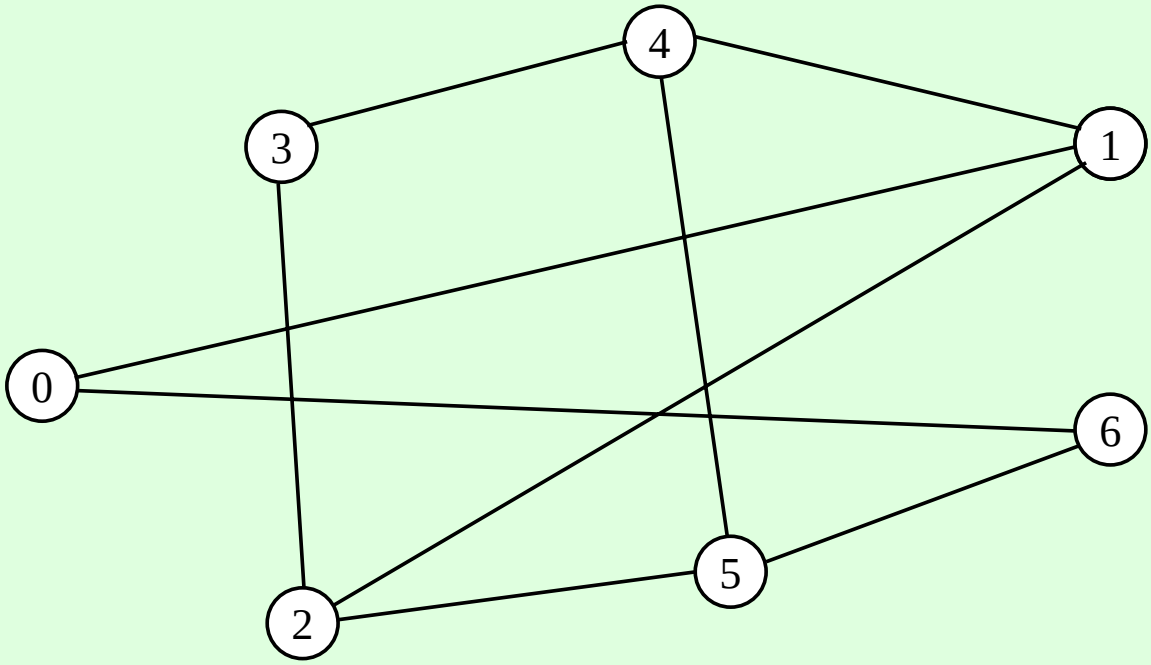
NEPs computers: an example

Motivation

- We will introduce a possible NEP to solve an a (toy) instance of the Hamiltonian path problem in a undirected graph with linear performance.
  - A Hamiltonian path is a path in an undirected graph that visits each vertex exactly once
- It will be used with different purposes allong this talk
  - We have considered it a particular case of a family of increasingly complex graphs in order to test the performance of the architectures that simulate it.

NEPs computers

Example: Hamiltonian path, undirected graphs



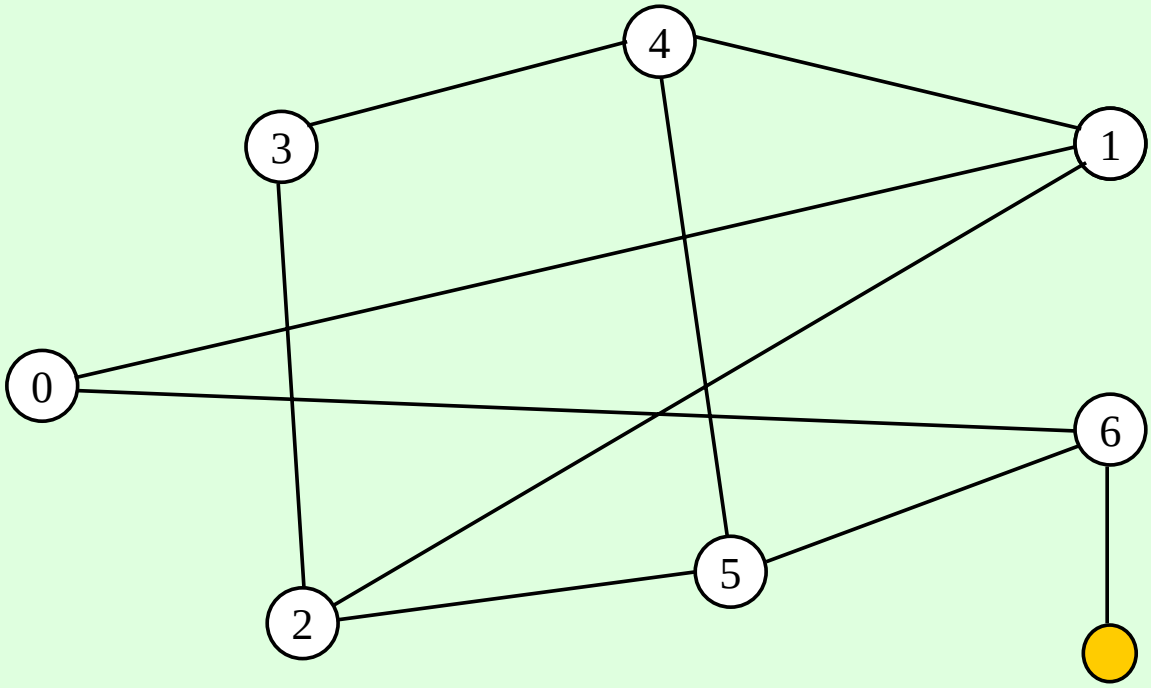
NEPs computers

Example: Hamiltonian path, undirected graphs

- The NEP for the previous graph
  - Has a processor for each node. Only the strings that have already visited the node cannot enter the node. There are no additional condition in the filters.
  - The NEP adds a new processor (the output one) connected to the output node.The output node only receives the solutions.
  - The only node with non empty initial contents is the initial (it will content the string i)
  - Each non ouput processor adds its number in the right end of its strings.
- It is a simpler version than the one we will show with jNEPview
  - It has less connections
  - The filters of its nodes are more restrictive: they reject the strings that have already visited the node

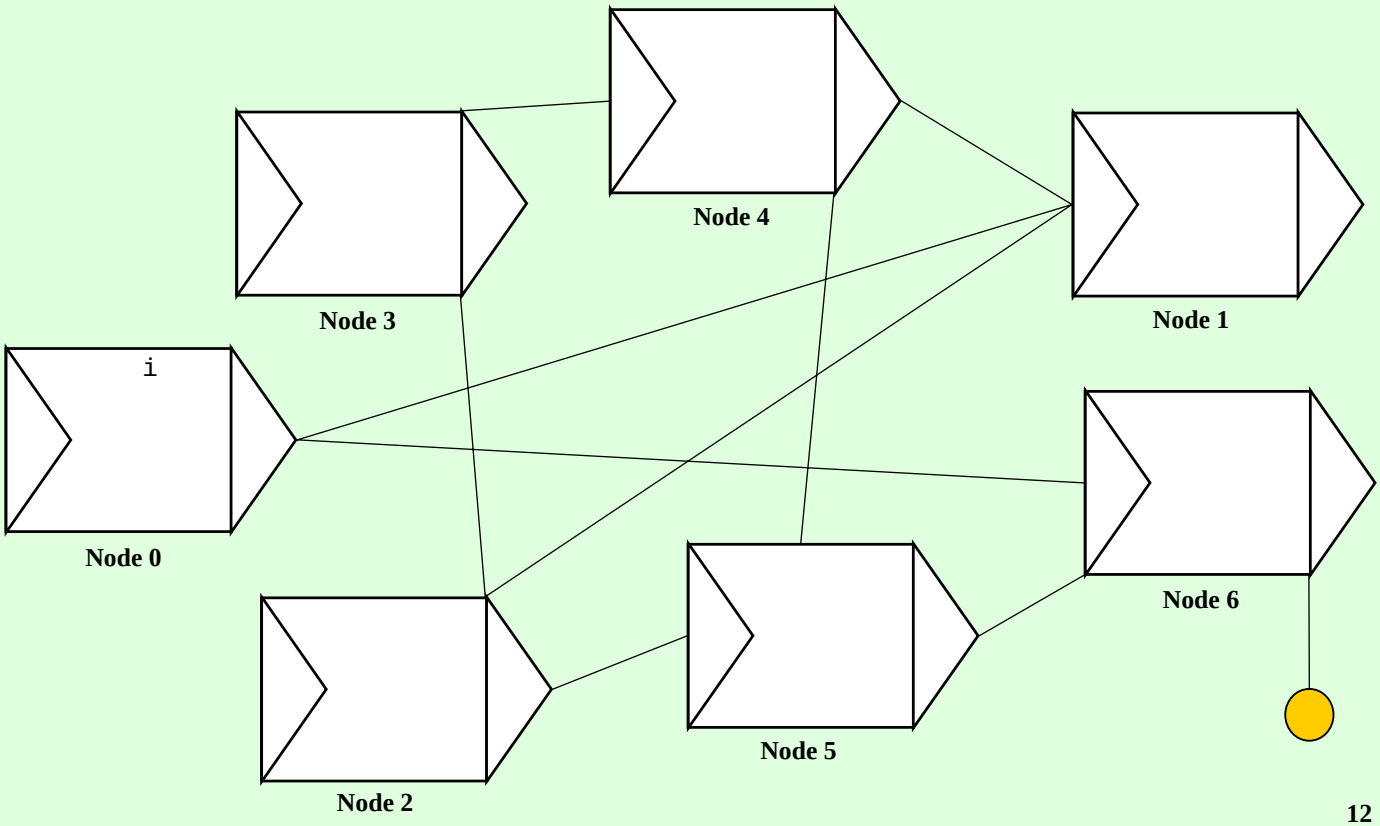
NEPs computers

Example: Hamiltonian path, undirected graphs.



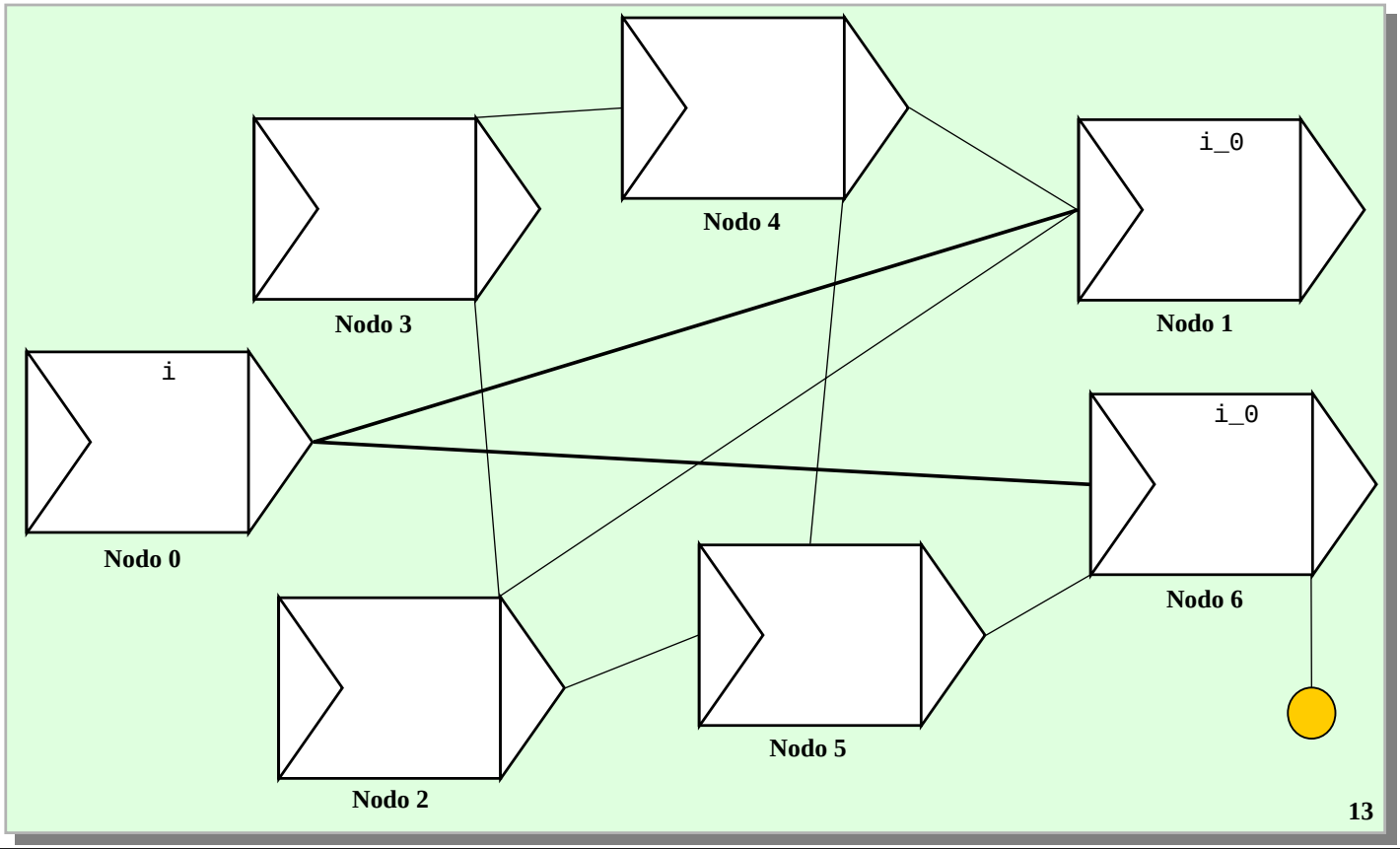
NEPs computers

Example: Hamiltonian path, undirected graphs. Step 0.



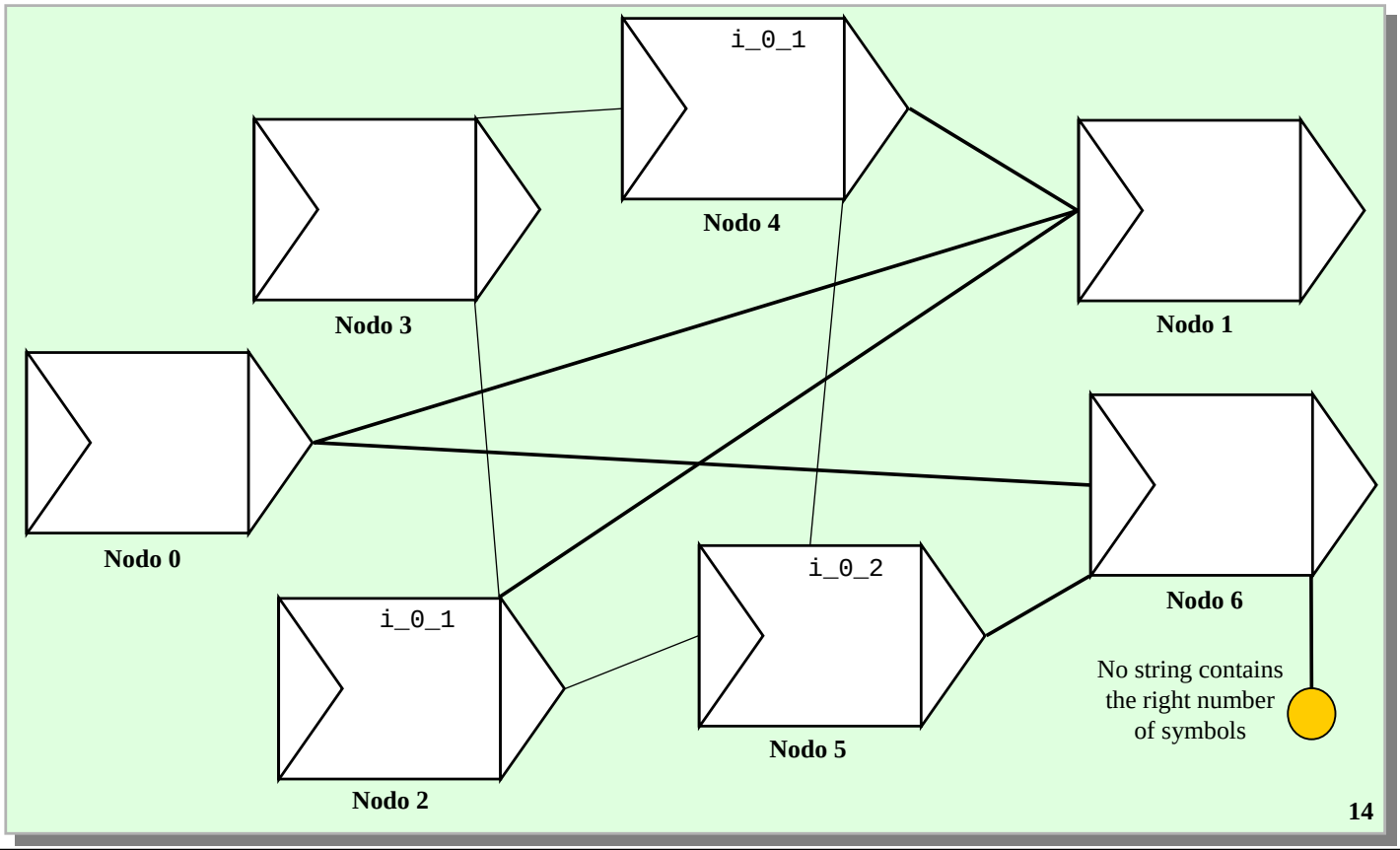
NEPs computers

Example: Hamiltonian path, undirected graphs . Step 1.



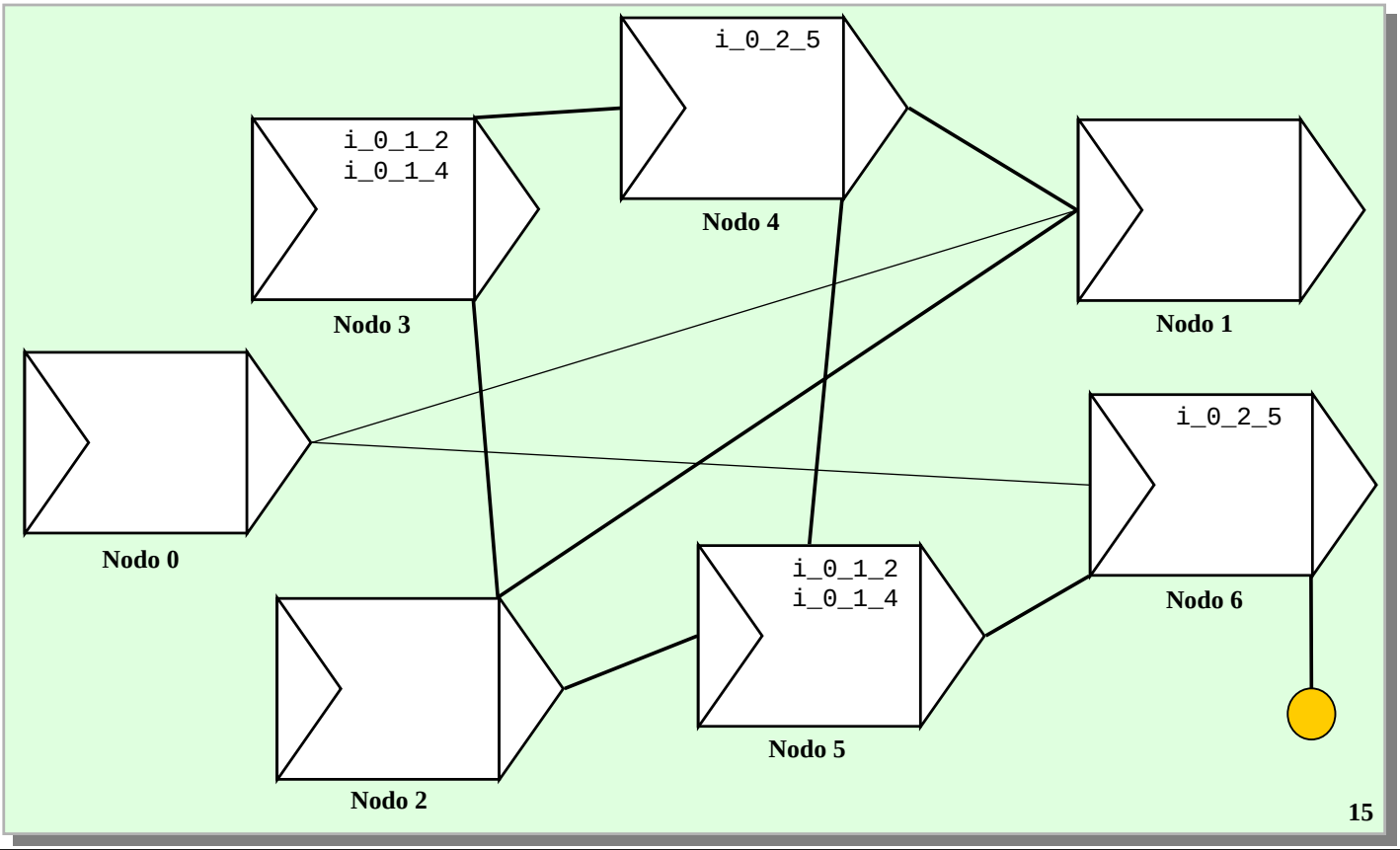
NEPs computers

Example: Hamiltonian path, undirected graphs . Step 2.



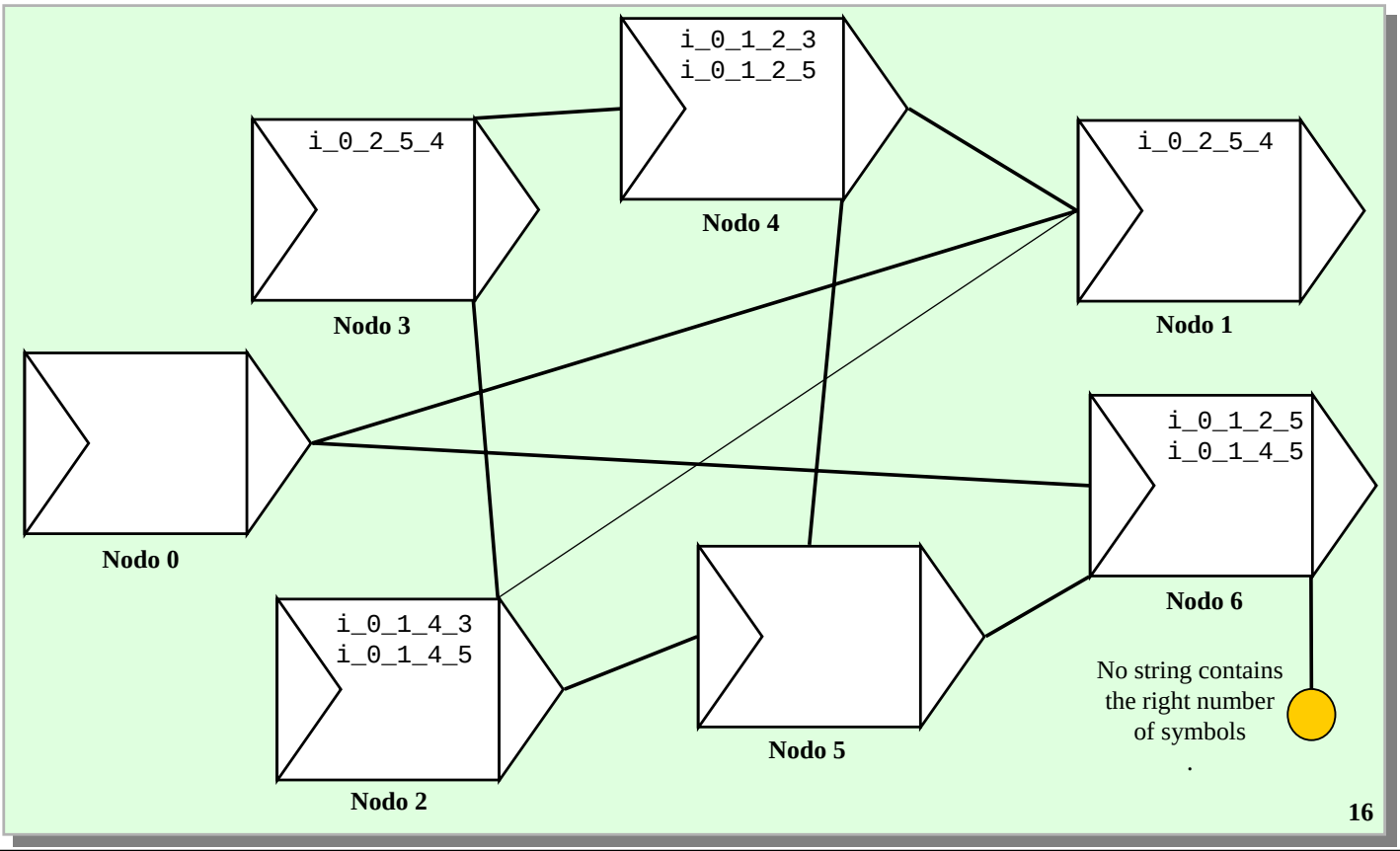
NEPs computers

Example: Hamiltonian path, undirected graphs . Step 3.



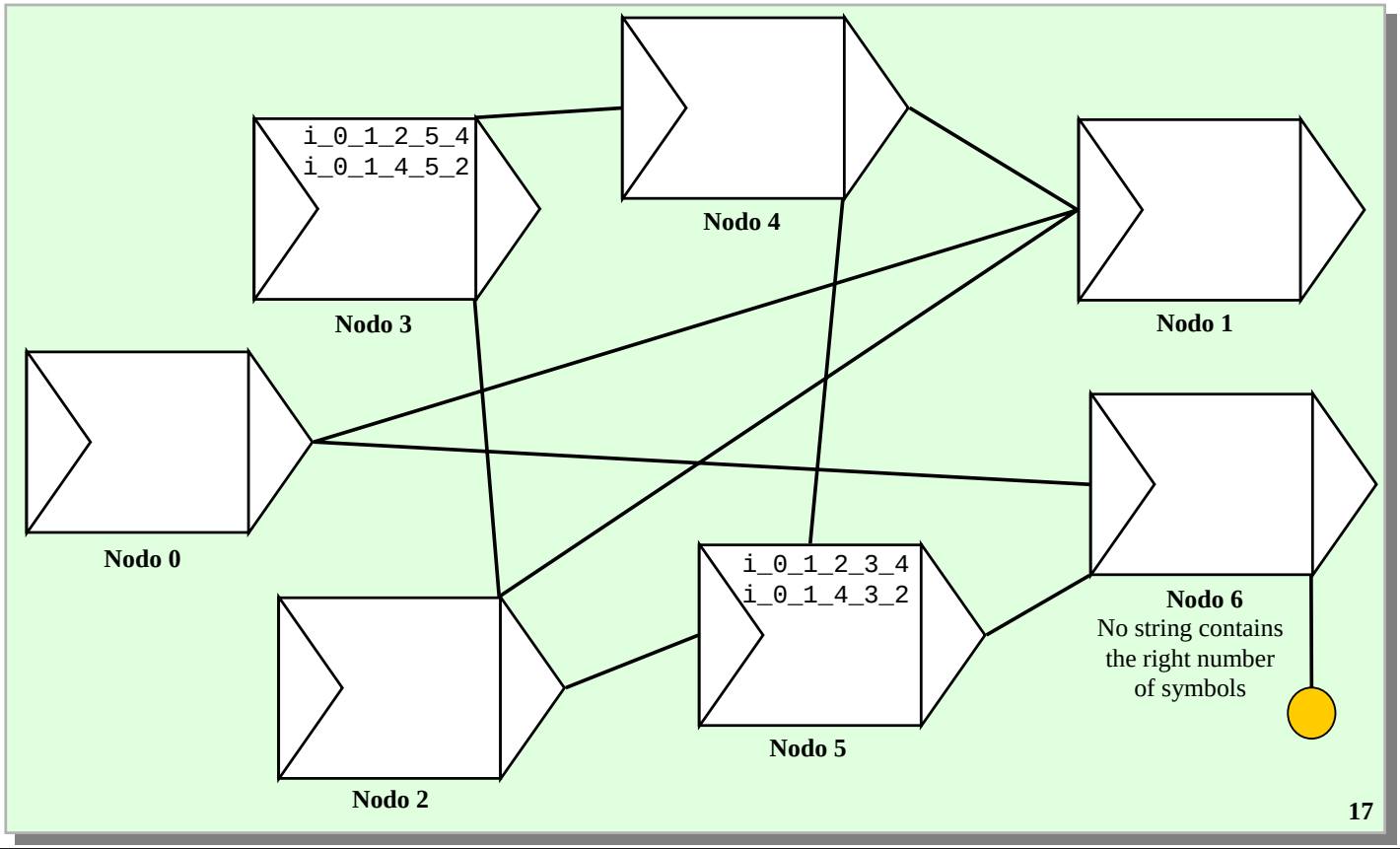
NEPs computers

Example: Hamiltonian path, undirected graphs . Step 4.



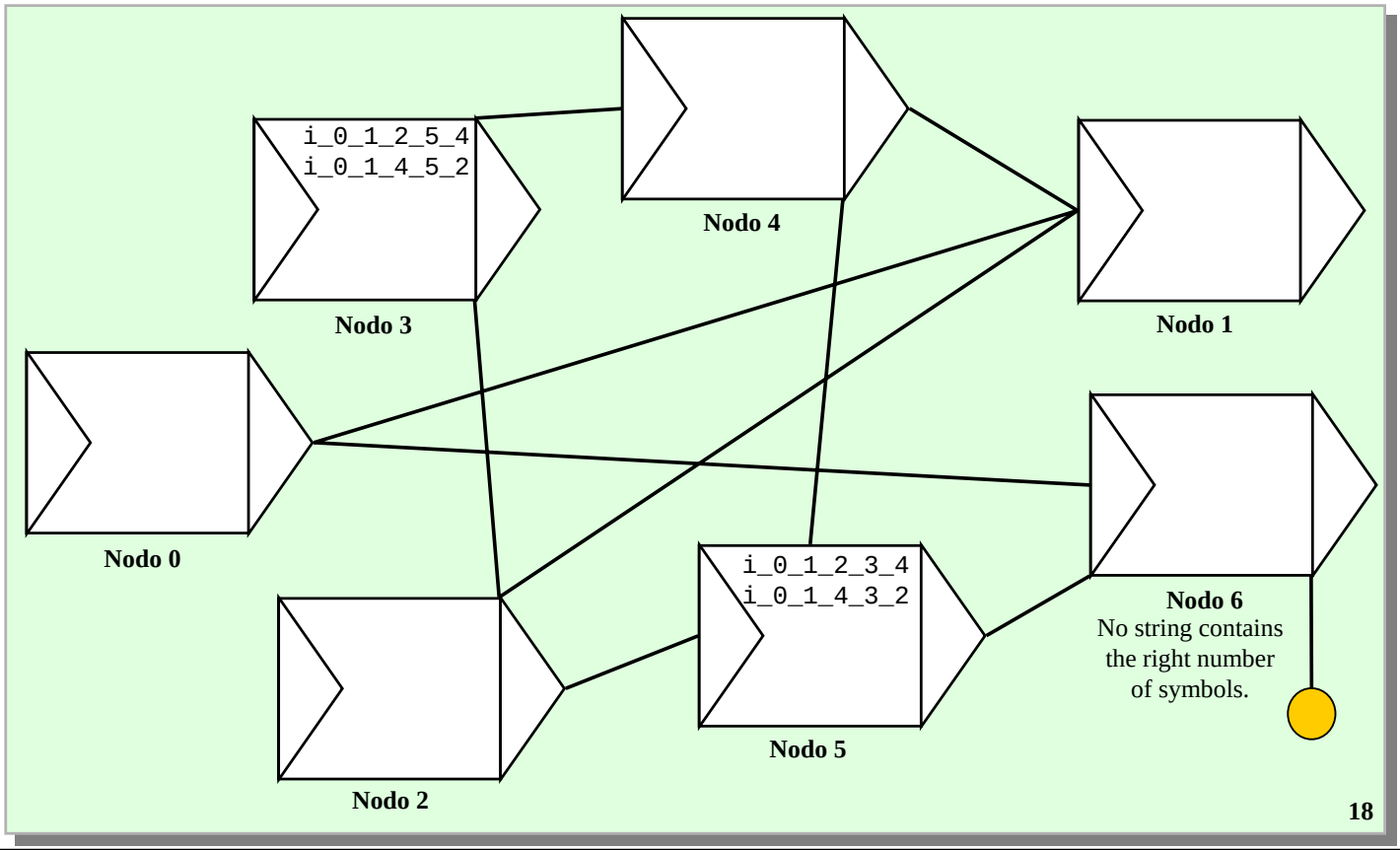
NEPs computers

Example: Hamiltonian path, undirected graphs . Step 5



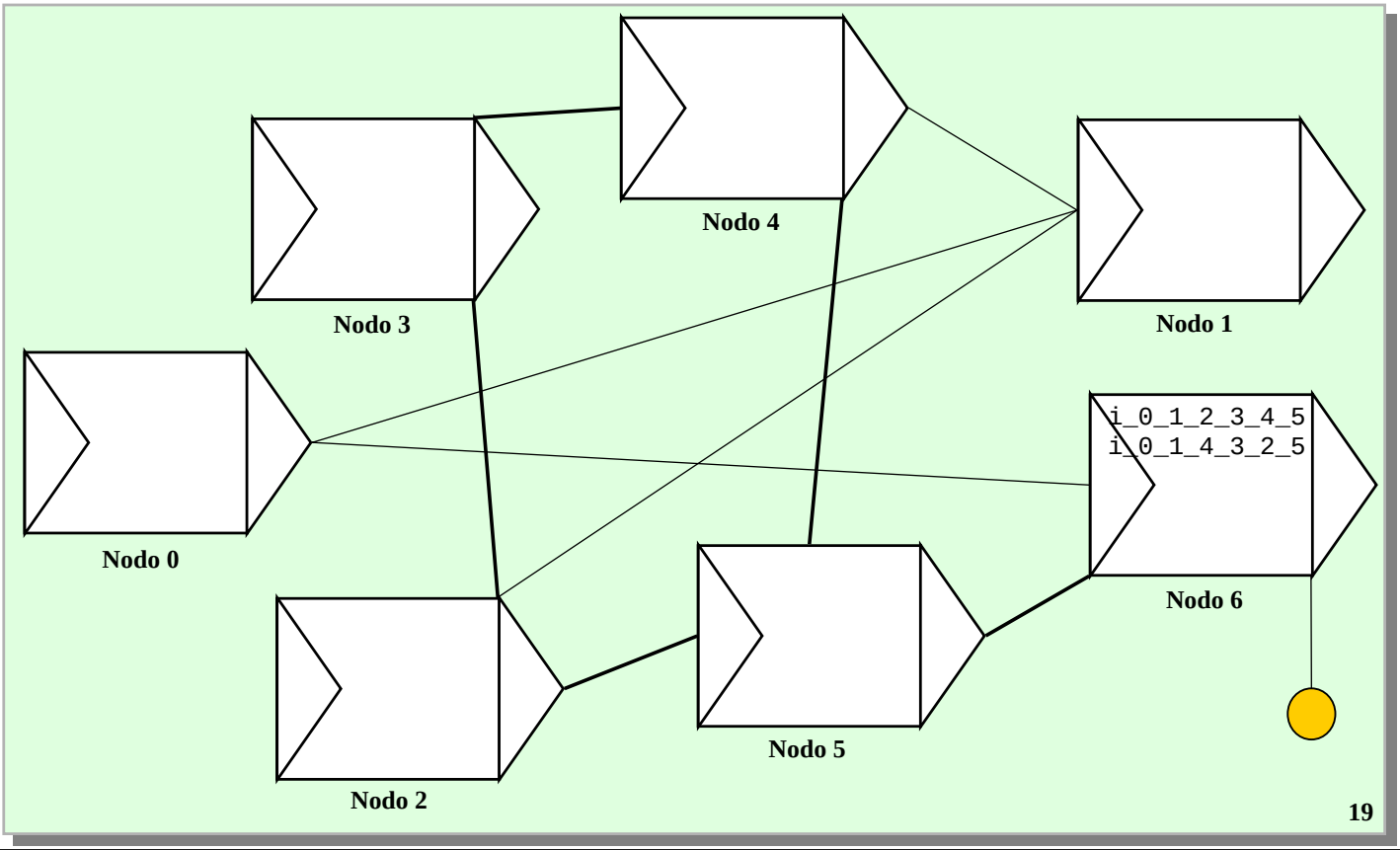
NEPs computers

Example: Hamiltonian path, undirected graphs . Step 6



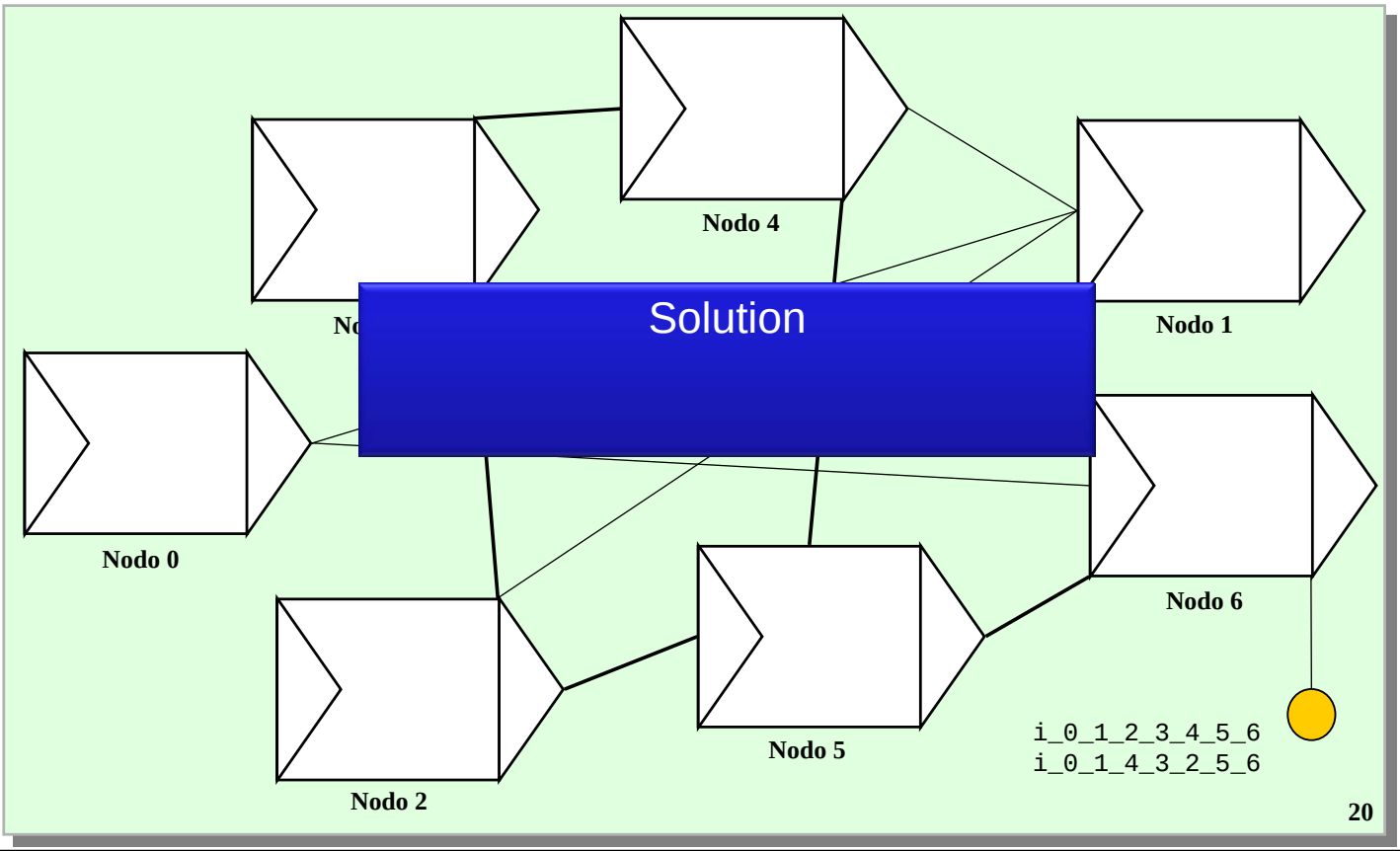
NEPs computers

Example: Hamiltonian path, undirected graphs . Step 7



NEPs computers

Example: Hamiltonian path, undirected graphs . Step 8



NEPs computers

An overview of a possible architecture for NEPs computers

- Layout of the architecture
- Solving an instance of the Hamiltonian path problem with NEPs
- **Programming tools for NEPs**
  - **Graphic simulation environment on ‘classical’ architectures**
    - **jNEP, a Java multithreaded NEP simulator**
    - jNEPView, agraphical viewer for the simulation of jNEP
    - NEPsVL, a visual programming language for NEPs
  - Simulation on clusters of computers
  - Textual programming languages independent on the architecture
- Some applications
  - Solving NP-problems with lineally bounded resources
  - Some applications of NEPs to language processing

NEPs computers

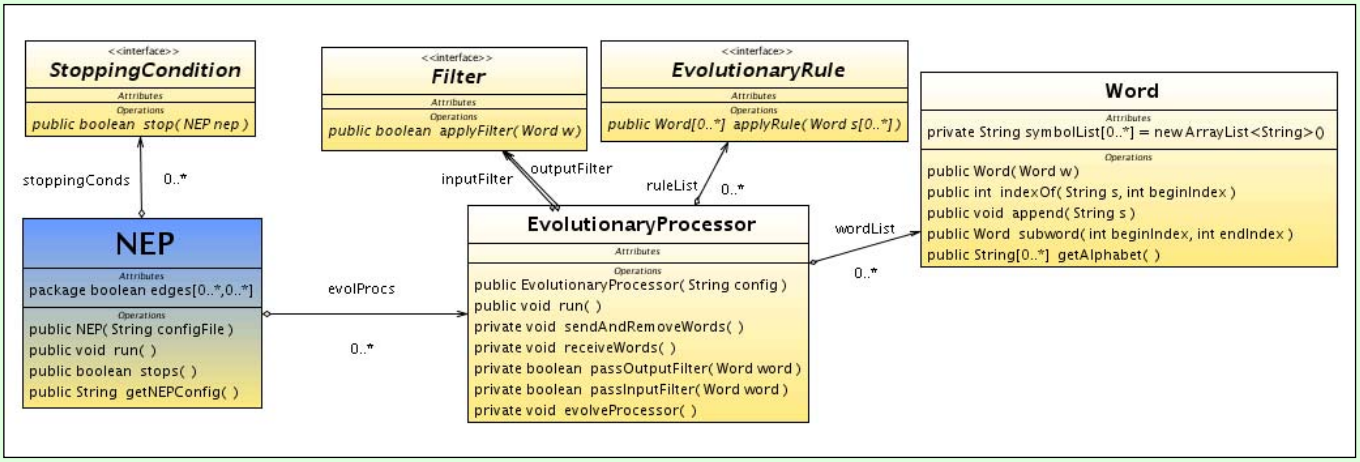
jNEP

- jNEP is simulator for NEPs with the following charateristics:
  - Multithreaded
  - Written in Java
  - As general, flexible and rigorous as possible
  - Input: xml files describing the NEP, example:

NEPs computers

jNEP

- The design NEP class mimics the NEP model definition



NEPs computers

jNEP

- jNEP demo for solving the instance of the Hamiltonian path problem above
  - Input specification xml file:

F:\intercambio_baja\escuela_verano_red_tematica_osuna\
simulador_jnep\jnep-last\adelman.xml

  - For running the program

```
> java -cp build/classes net.e_delrosal.jnep.ThreadedNEP Adleman.xml
```

NEPs computers

An overview of a possible architecture for NEPs computers

- Layout of the architecture
- Solving an instance of the Hamiltonian path problem with NEPs
- **Programming tools for NEPs**
  - **Graphic simulation environment on 'classical' architectures**
    - jNEP, a Java multithreaded NEP simulator
    - **jNEPView, a graphical viewer for the simulation of jNEP**
    - NEPsVL, a visual programming language for NEPs
  - Simulation on clusters of computers
  - Textual programming languages independent on the architecture
- Some applications
  - Solving NP-problems with lineally bounded resources
  - Some applications of NEPs to language processing

NEPs computers

jNEPview

- jNEPview works in the following way:
  1. jNEP has to be run with the –save option (to store its sequence of computation steps)

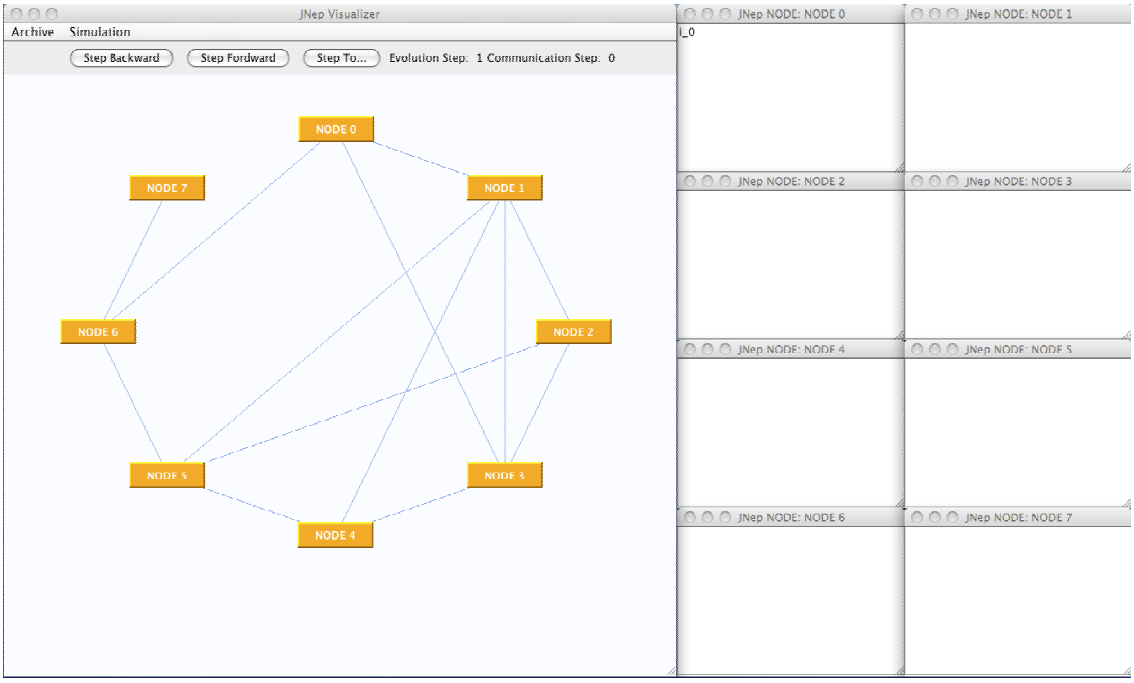
```
> java -cp build/classes net.e_delrosal.jnep.ThreadedNEP Adleman.xml –save
```

  2. jNEPview is started (double click on the .jar)
     1. The xml file of the NEP has to be loaded
     2. The sequence of states is shown
  3. Simulation can
     1. Advance (forward and bakward) step by step
     2. Advance an arbitrary number of steps

NEPs computers

jNEPview demo

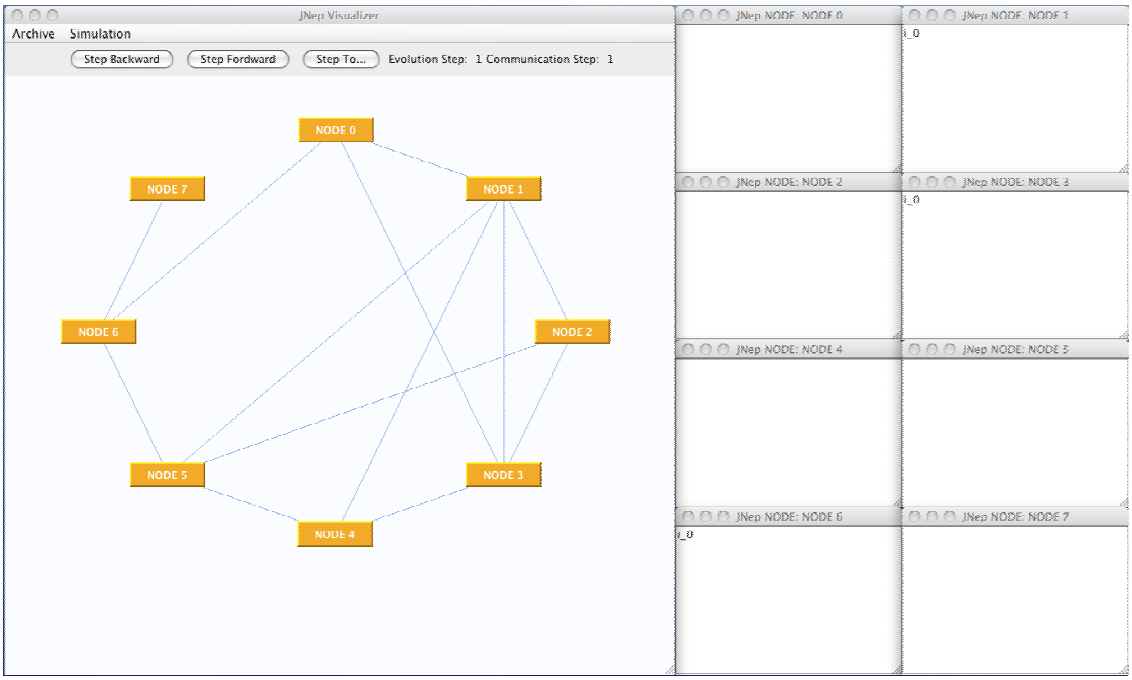
- Tracing jNEP for solving the instance of the Hamiltonian path problem above:
  - After opening the diagram and all its nodes



NEPs computers

jNEPview demo

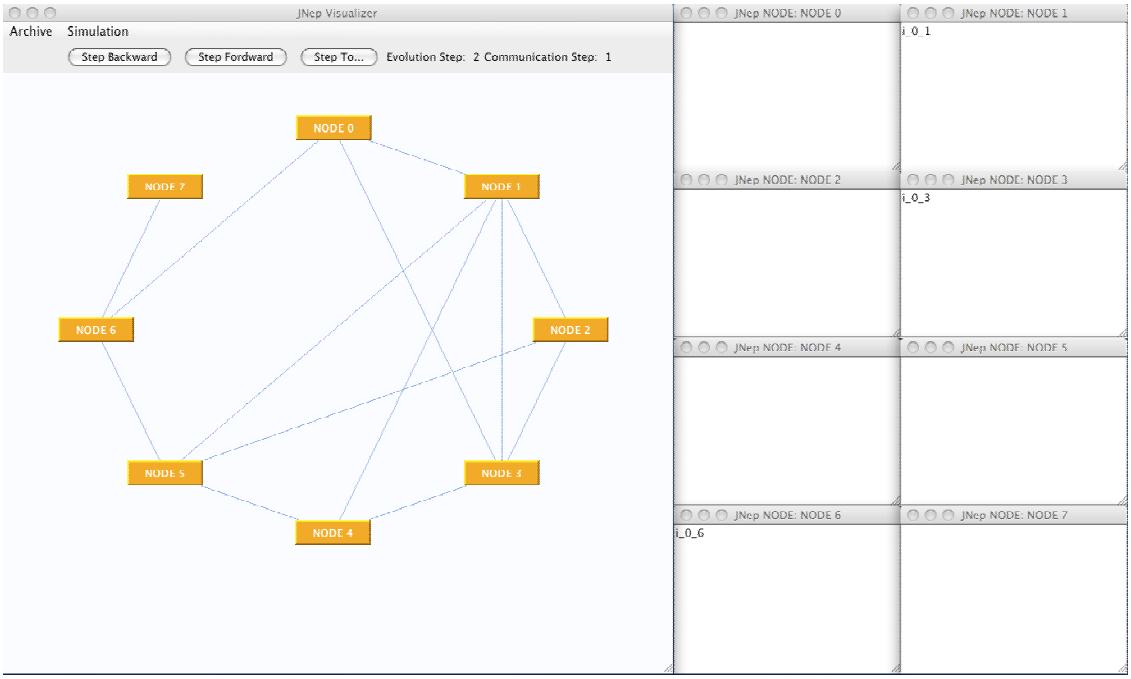
- Tracing jNEP for solving the instance of the Hamiltonian path problem above:
  - After pass 1



NEPs computers

jNEPview demo

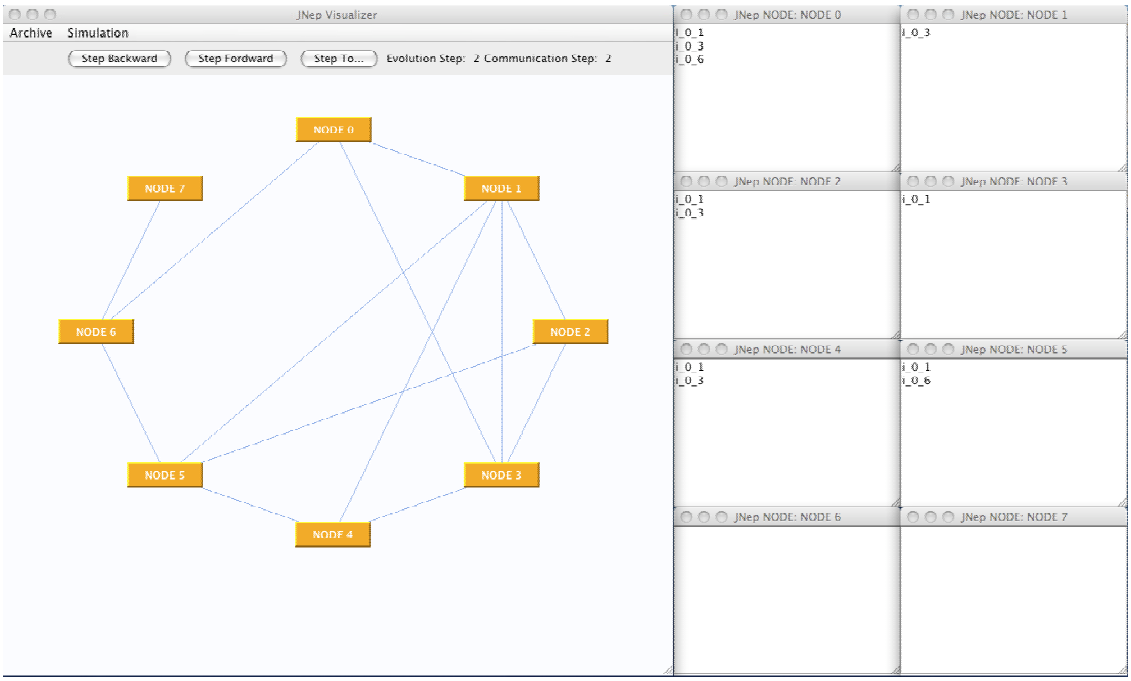
- Tracing jNEP for solving the instance of the Hamiltonian path problem above:
  - After pass 2



NEPs computers

jNEPview demo

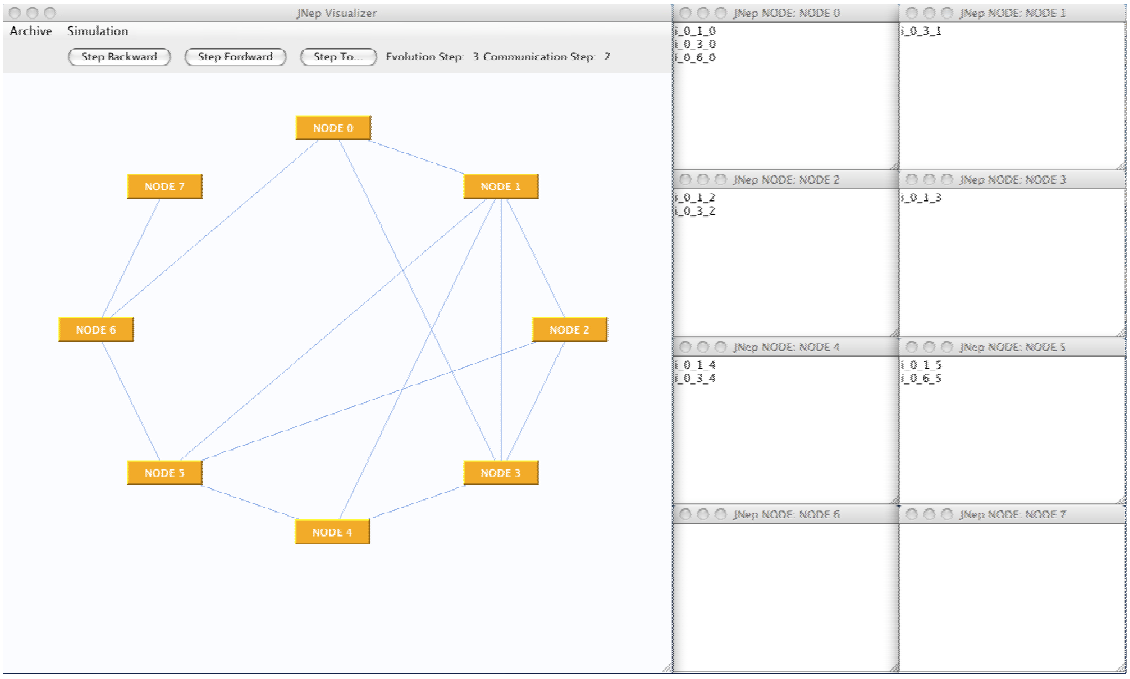
- Tracing jNEP for solving the instance of the Hamiltonian path problem above:
  - After pass 3



NEPs computers

jNEPview demo

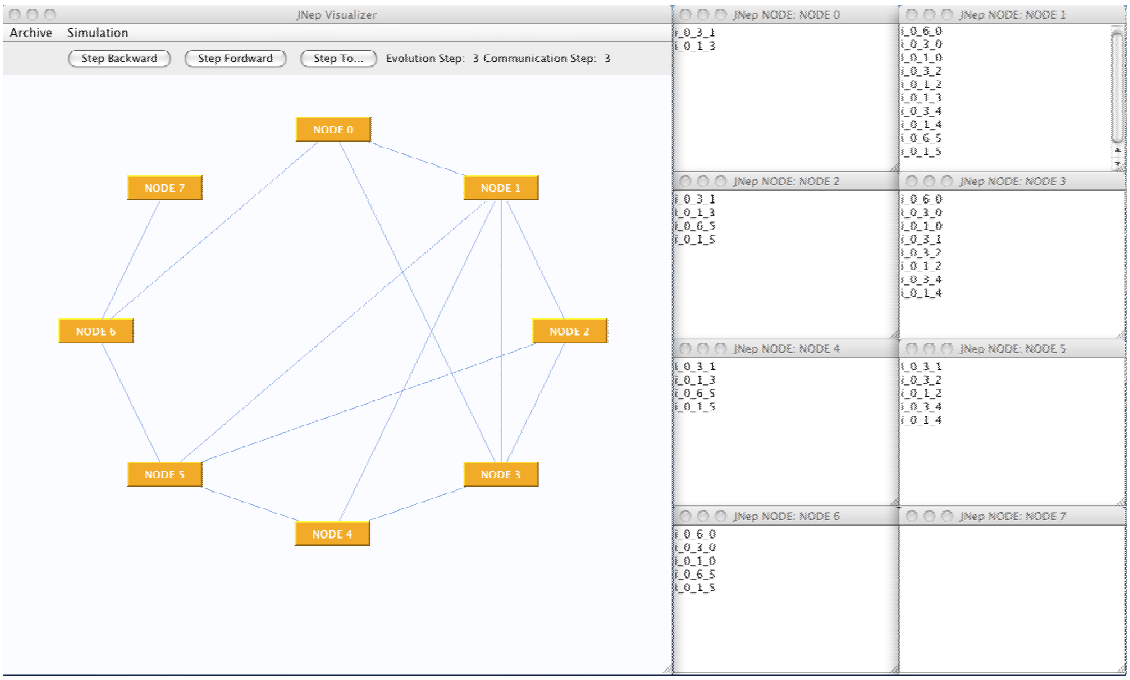
- Tracing jNEP for solving the instance of the Hamiltonian path problem above:
  - After pass 4



NEPs computers

jNEPview demo

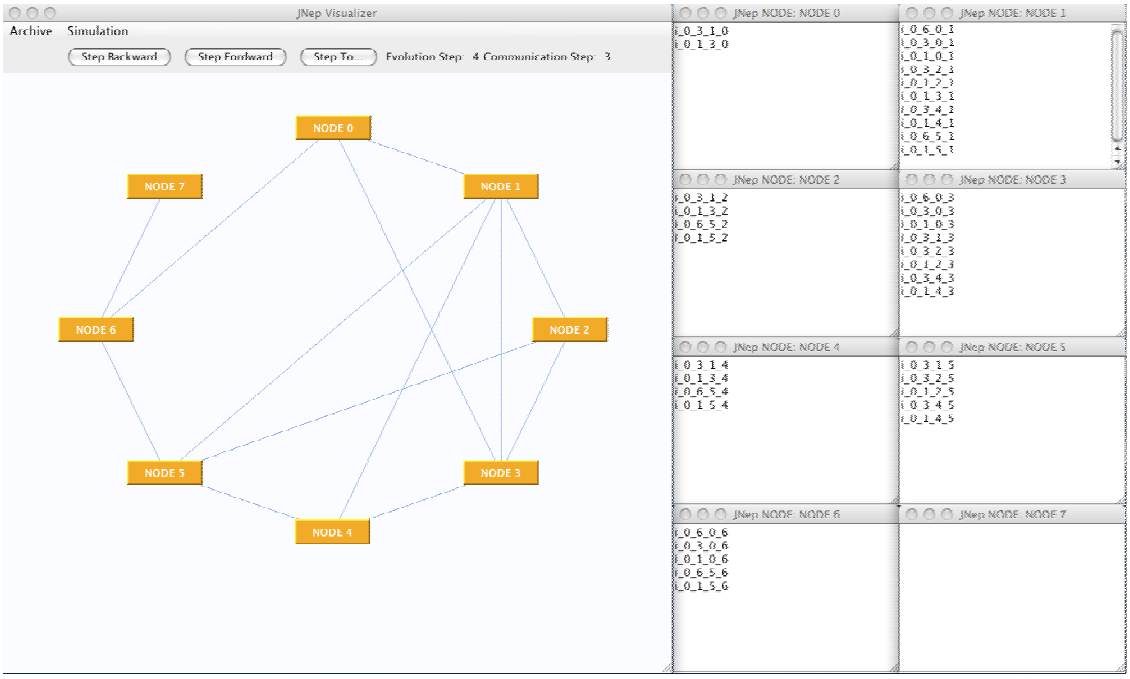
- Tracing jNEP for solving the instance of the Hamiltonian path problem above:
  - After pass 5



NEPs computers

jNEPview demo

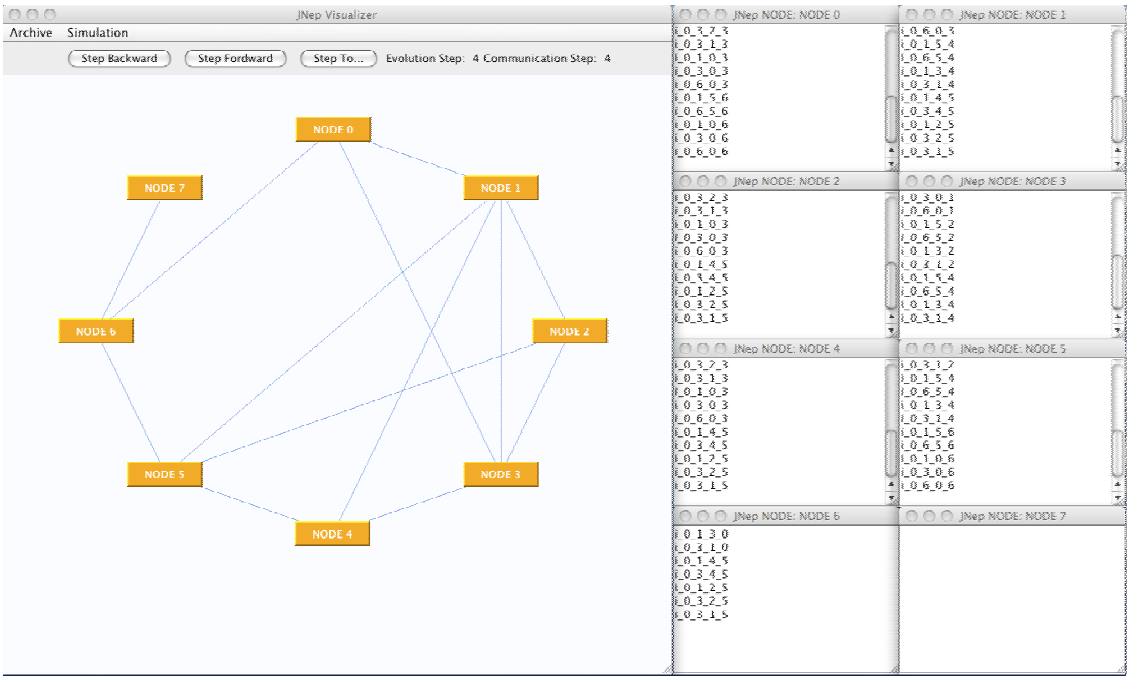
- Tracing jNEP for solving the instance of the Hamiltonian path problem above:
  - After pass 6



NEPs computers

jNEPview demo

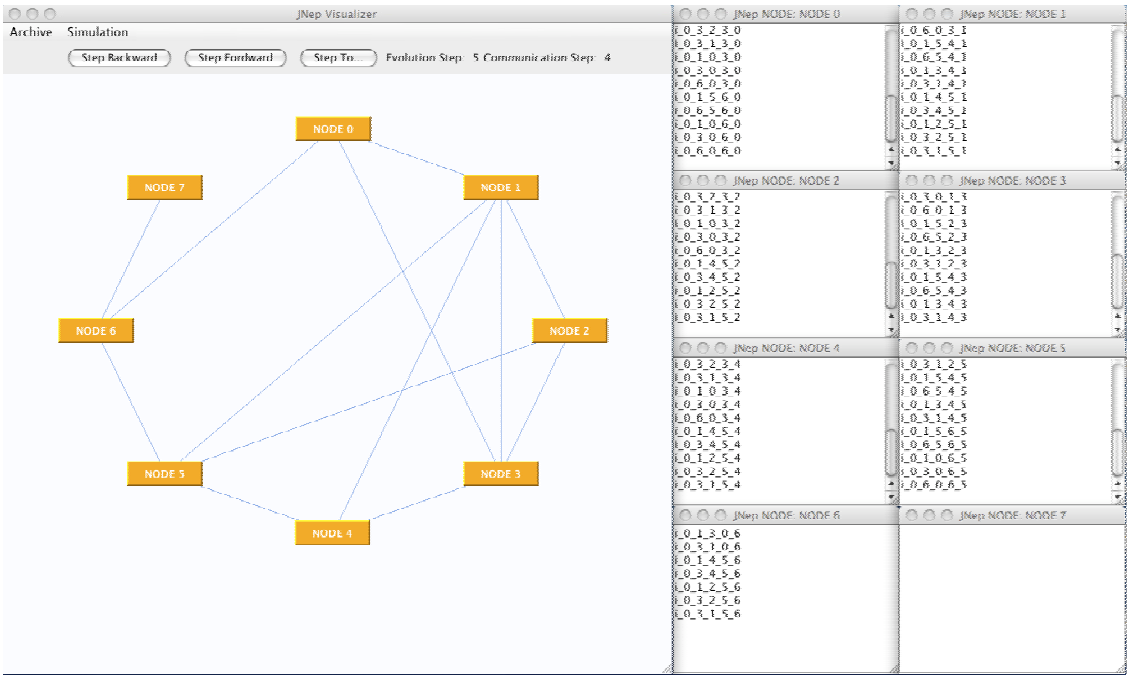
- Tracing jNEP for solving the instance of the Hamiltonian path problem above:
  - After pass 7



NEPs computers

jNEPview demo

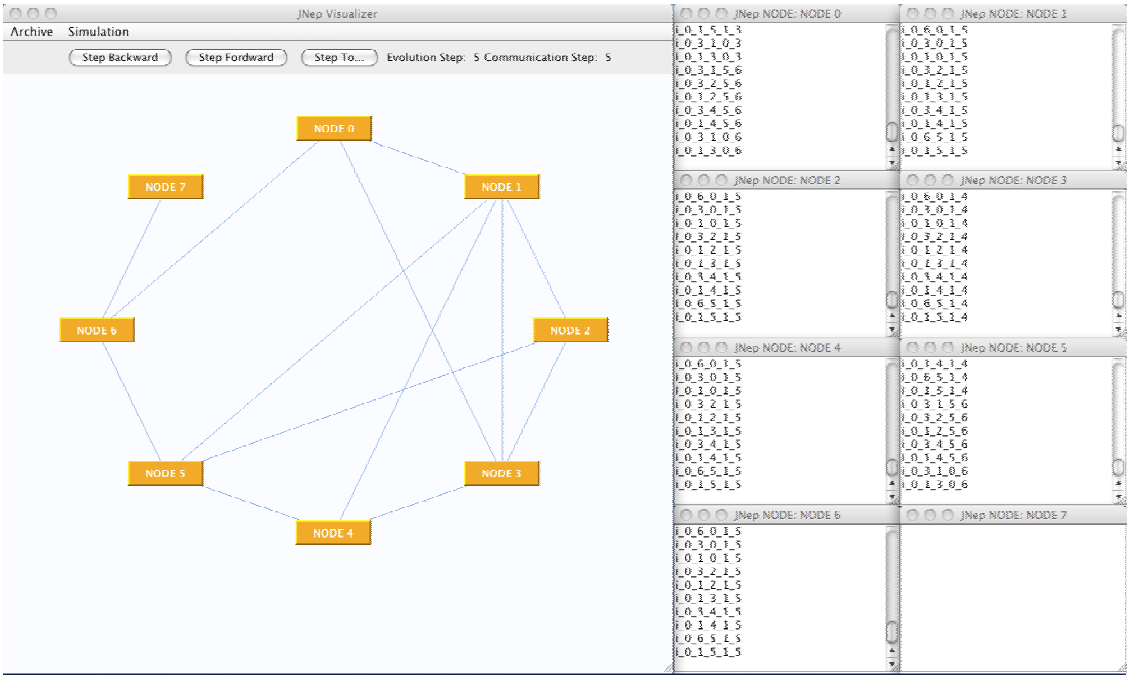
- Tracing jNEP for solving the instance of the Hamiltonian path problem above:
  - After pass 8



NEPs computers

jNEPview demo

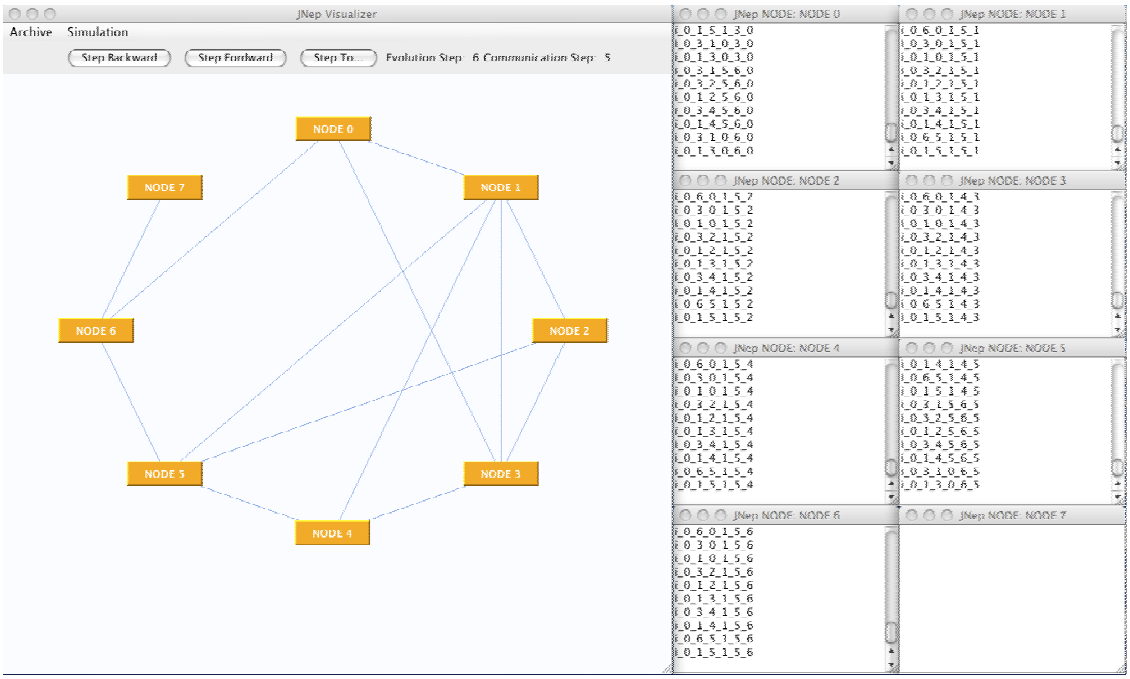
- Tracing jNEP for solving the instance of the Hamiltonian path problem above:
  - After pass 9



NEPs computers

jNEPview demo

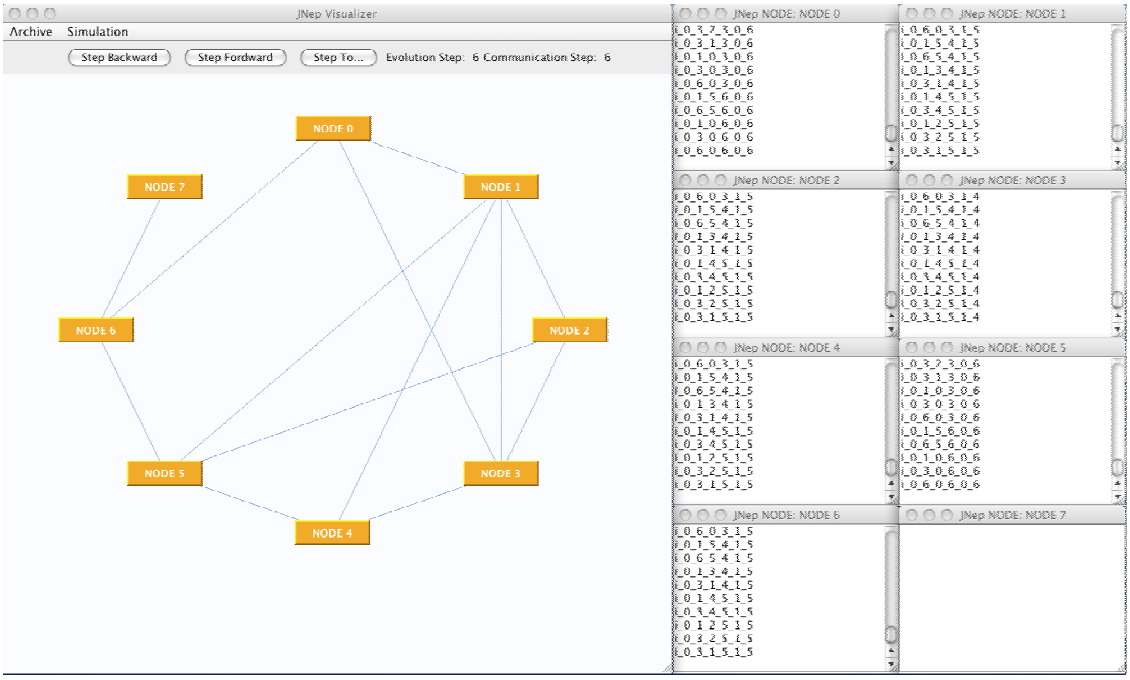
- Tracing jNEP for solving the instance of the Hamiltonian path problem above:
  - After pass 10



NEPs computers

jNEPview demo

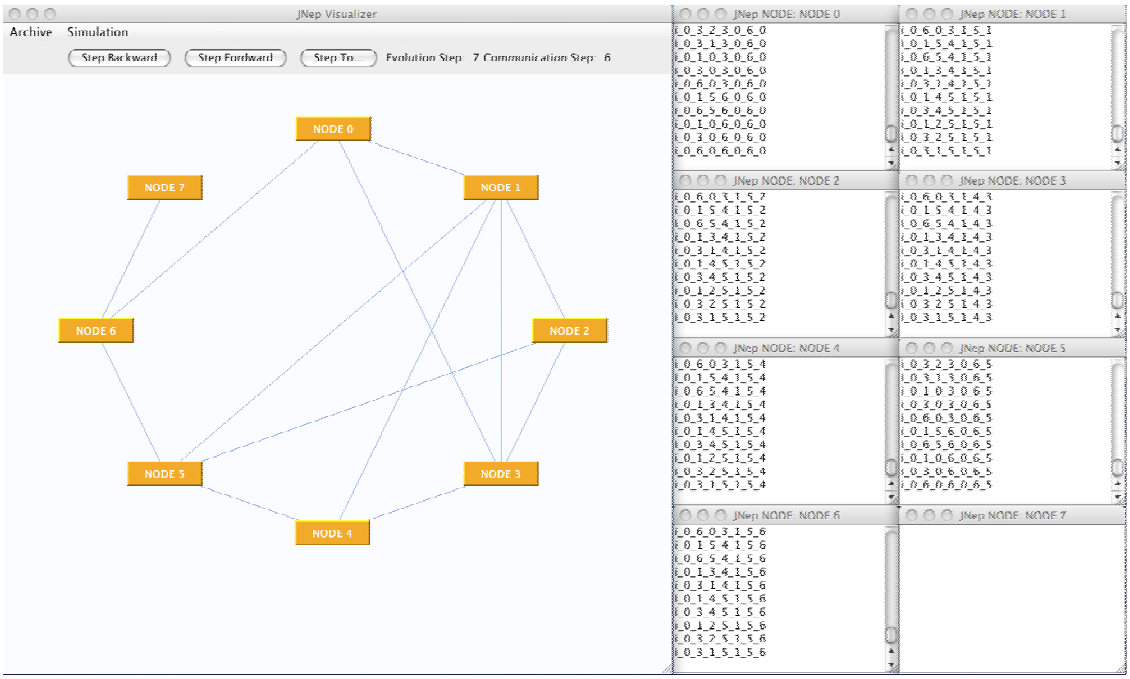
- Tracing jNEP for solving the instance of the Hamiltonian path problem above:
  - After pass 11



NEPs computers

jNEPview demo

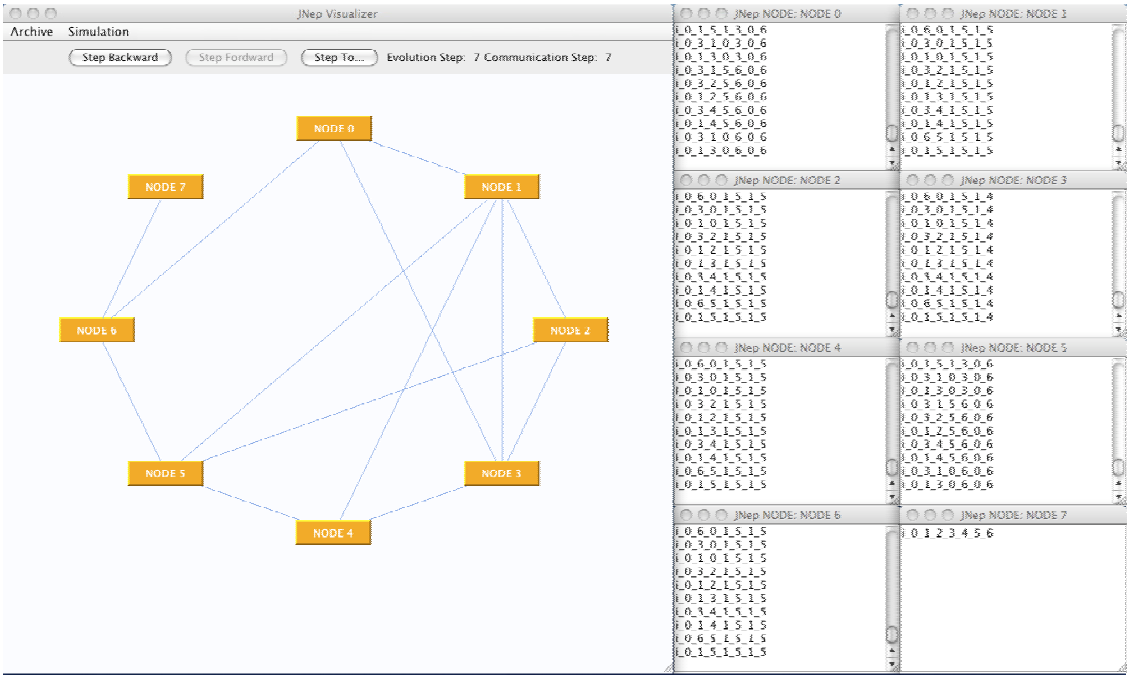
- Tracing jNEP for solving the instance of the Hamiltonian path problem above:
  - After pass 12



NEPs computers

jNEPview demo

- Tracing jNEP for solving the instance of the Hamiltonian path problem above:
  - After pass 13



NEPs computers

An overview of a possible architecture for NEPs computers

- Layout of the architecture
- Solving an instance of the Hamiltonian path problem with NEPs
- **Programming tools for NEPs**
  - **Graphic simulation environment on 'classical' architectures**
    - jNEP, a Java multithreaded NEP simulator
    - jNEPView, a graphical viewer for the simulation of jNEP
    - **NEPsVL, a visual programming language for NEPs**
      - **Brief introduction to AToM^3**
      - NEPsVL
  - Simulation on clusters of computers
  - Textual programming languages independent on the architecture
- Some applications
  - Solving NP-problems with lineally bounded resources
  - Some applications of NEPs to language processing

NEPs computers

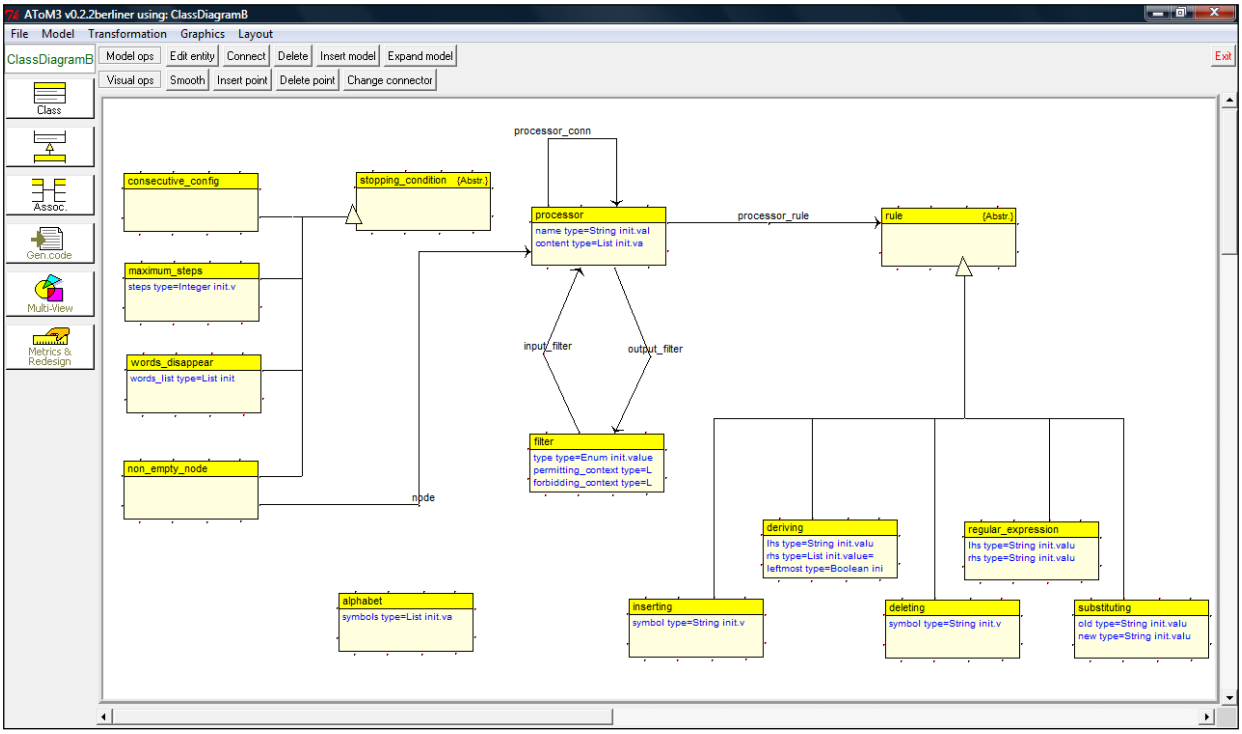
AToM^3-NEPsVL Demo

- AToM^3
  1. Python software platform to develop visual domain languages
  2. UML oriented model design
     1. The user defines the class diagram
     2. Extends it with graphical appearance
     3. Generates a canvas where to graphically design the final program by means of predefined buttons and menu options associated with the UML diagram
     4. Validates and generates code
- Demo
  - Double click on the atom3.py icon

NEPs computers

ATOM^3-NEPsVL Demo

- How to design a new DSVL (domain specific visual language) with ATOM^3?
  - A ‘metamodel’ of the domain elements has to be defined. This is the NEP metamodel



NEPs computers

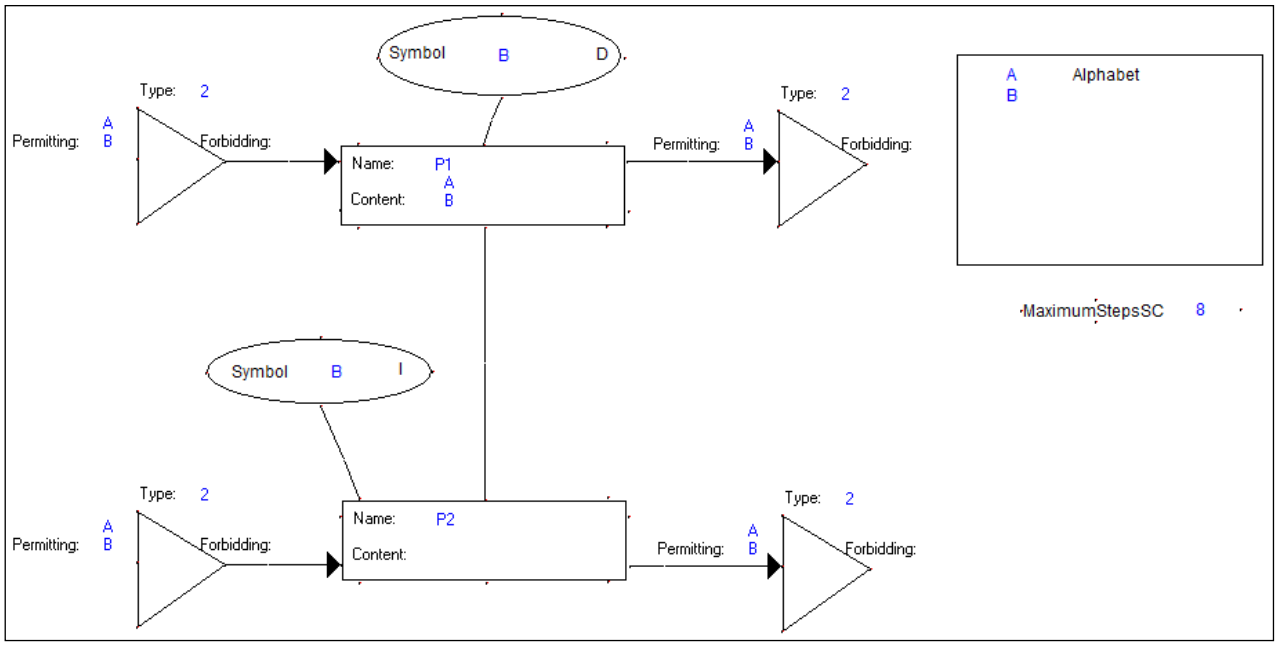
ATOM^3-NEPsVL Demo

- How to design a new DSVL (domain specific visual language) with ATOM^3?
  - Then, the graphical appearance with wich each element of the domain will be included in the programs has to be designed.
    - The following (toy) example shows the one used for NEPs:
      - Rectangles for processors and for the alphabet
      - Triangles for filters
      - Ovals for rules
      - Texts for the rest
    - When designing the graphical appearance the programmer also specifies the set of proper actions that have to be performed when the element under consideration is added to the programs

NEPs computers

ATOM^3-NEPsVL Demo

- How to design a new DSVL (domain specific visual language) with ATOM^3?
  - Graphical appearance of each element



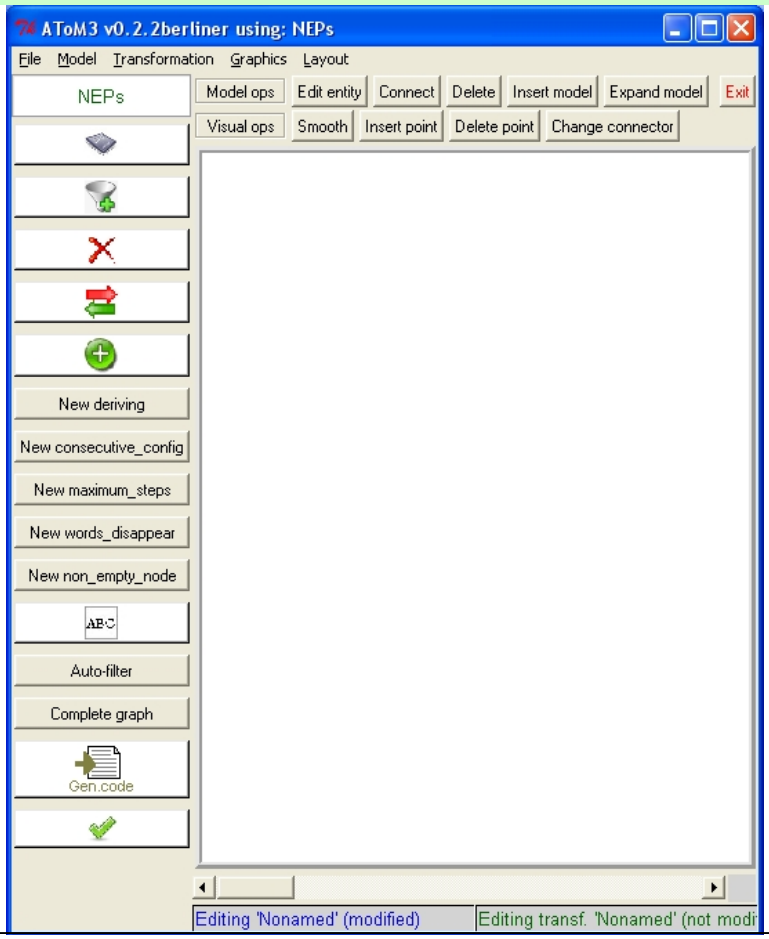
NEPs computers

ATOM^3-NEPsVL Demo

- How to design a new DSVL (domain specific visual language) with ATOM^3?
  - The buttons by means of wich the programmer will interact with ATOM^3 for graphically design NEPs have also to be designed.
    - The following figure shows the ATOM^3 window for programming NEPs
      - The first 11 buttons (below the title 'NEPs') insert into the program, respectively
        - A processor
        - A filter
        - A deletion rule
        - A substitution rule
        - An insertion rule
        - A derivation rule (they will be described later)
        - Four tipical stopping conditions
      - The last 4 buttons adds extra features that ease the programming. They will be explained in the following pages

NEPs computers

- How to design a new DSVL (domain specific visual language) with AToM³?
  - Buttons by means of wich the programmer will interact with AToM³



NEPs computers

AToM³-NEPsVL Demo

- Some useful extra features of AToM³?
  - AToM³ provides programmer with a set of useful features to simplify the programming
    - The programmer can specify several python programs to process the complete program. They are very useful for, for example
      - Validating the program (check semantic constraints that valid programs have to comply with)
      - Generating code (translate the graphic programs into other representations, in the case of NEPs, for example, into xml files for jNEP, or into NEPsLingua code)
      - Launching some task after finishing the process of the program (for example jNEP and jNEPView)
    - In the previous page, these three features are associated with the last two buttons

NEPs computers

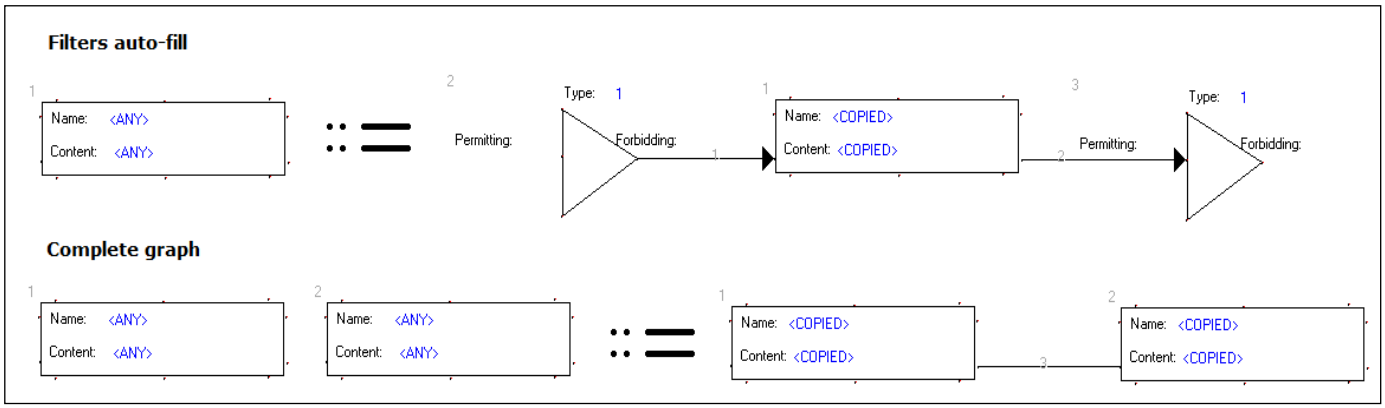
ATOM^3-NEPsVL Demo

- Some useful extra features of ATOM^3?
  - ATOM^3 provides programmer with a set of useful features to simplify the programming
    - Graph grammars
      - Some mechanical sections of the graphical programs can be automatically written (actually drawn), for example
        - The output and input filters of each processor can be automatically added
        - All the connections of complete graphs can also be automatically added
      - These two features are associated with the corresponding buttons (‘auto-filter’ and ‘complete graph’) of the previous page
      - Graph grammars are just grammars that change some ocurrences of a subgraph
      - These derivations rules are graphically defined in ATOM^3.
      - The following page shows the ‘auto-filter’ and ‘complete graph’ rules

NEPs computers

ATOM^3-NEPsVL Demo

- Some useful extra features of ATOM^3?
  - ATOM^3 provides programmer with a set of useful features to simplify the programming
    - Graph grammars



NEPs computers

An overview of a possible architecture for NEPs computers

- Layout of the architecture
- Solving an instance of the Hamiltonian path problem with NEPs
- **Programming tools for NEPs**
  - Graphic simulation environment on ‘classical’ architectures
  - **Simulation on clusters of computers**
  - Textual programming languages independent on the architecture
- Some applications
  - Solving NP-problems with lineally bounded resources
  - Some applications of NEPs to language processing

NEPs computers

Simulation on clusters of computers: motivation

- Which hardware/software platform use to run NEPs programs?
  - There is no real hardware platform for NEPs
  - The theoretical improvement (in the performance) is lost when simulating them on ‘conventional’ computers
  - Clusters of computers could be the best option
    - To overtake the performance lost
    - Although renouncing to the theoretical power

NEPs computers

Simulation on clusters of computers: structure of the system

- Some of our researchers have developed a software C++/mpi platform for automatic paralellisation of sequential codes using dynamic graphs partitioning and based on user-adaptable load balancing
- The programmer has only to write its algorithm in a predefined way.
- The platform handles all the low level details ensuring the best possible performance for the cluster under consideration.


```
#ifndef DATA_xxx_H_
#define DATA_xxx_H_

#include "ISerializable.h"
#include <iostream>
#include <sstream>

#include "Estructuras_de_datos_necesarias.h"
#include <string>

namespace Reconfiguration
{
    class Data : public ISerializable
    {
        public:

            // ATRIBUTOS DE LOS DATOS NECESARIOS DE CADA NODO
            // DE LA TOPOLOGIA "LOGICA"

            Data();
            Data( /* ESTRUCTURAS DE DATOS DEL CONTENIDO DE CADA
            NODO DE LA TOPOLOGIA LOGICA*/);
            Data(const Data&);
            virtual ~Data();
            void print();  // COMO SE IMPRIMIRA CADA NODO

            // SERIALIZADOR Y DESERIALIZADOF
            std::stringstream& operator<<(std::stringstream&);
            std::stringstream& operator>>(std::stringstream&);
    };
}

#endif /* DATA_xxx_H_ */
```

```
#ifndef XXX_H_
#define XXX_H_

#include "Data_xxx.h"
#include "Errors.h"
#include "Graph.h"

#define MAX_STEPS 5 // CONDICION DE TERMINACION STANDARD

namespace Reconfiguration
{
    class NEP : public Algorithm
    {
        int iSteps;
        public:
            NEP();
            virtual ~NEP();

            // PARA LA LECTURA DE LOS DATOS INICIALES
            // DE LA TOPOLOGIA LOGICA
            int process (const char*, Data**, int);

            // LO QUE DEBEN EJECUTAR LOS ESCLAVOS
            // (LOGICA DEL MODELO)
            void kernel(Graph *);

            // MANERA DE MOSTRAR LOS DATOS DEL ESCLAVO
            void save (Graph *, double **);
            int end();
    };
}

#endif /* XXX_H_ */
```

NEPs computers

An overview of a possible architecture for NEPs computers

- Layout of the architecture
- Solving an instance of the Hamiltonian path problem with NEPs
- **Programming tools for NEPs**
  - Graphic simulation environment on ‘classical’ architectures
  - Simulation on clusters of computers
  - **Textual programming languages independent on the architecture**
    - **NEPsLingua**
- Some applications
  - Solving NP-problems with lineally bounded resources
  - Some applications of NEPs to language processing

NEPs computers

NEPsLingua: goals

- Syntax as close as possible to the one used to describe NEPs in the literature
- To reduce the complexity and size of other kinds of specification
  - Xml configuration files (jNEP)
  - Domain specific visual languages (NEPsVL)
- As close to P-Lingua as possible with the aim of offering a set of similar textual programming languages to researchers who wants to use natural computing to solve their problems

NEPs computers

NEPsLingua syntax (by examples)

- Directive `@A` for the alphabet of the NEP (its set of symbols)

```
@A={X,S,a,b,o,O}
```

- `@N` for the nodes:
  - The most complex type of NEPs-Lingua data
  - Non indexed (defined by means of their names), `{initial, final}`
  - With numeric indices,

```
{m{i}: 0<=i<=10}
```

  - With symbolic indices,

```
{s{j}: j->{o,a,b}}
```

  - That could be mixed in the same sentence for declaring the nodes of the NEP (note the use of + for the union of sets)

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

NEPs computers

NEPsLingua syntax (by examples)

- Initial content
  - The set of strings that a given node initially contains. Notice that the node is written as a parameter of the content directive @c

```
@c{n{X}} = {X, S}
```

- Rules
  - `-->` separates left and right hand side
  - They have to be contained in sets
  - `#` represents the empty string
  - All types are available
    - Insertion `# -->a`
    - Deletion `a -->#`
    - Substitution `S-->a`
  - And are put together in sets associated to the node by means of the directive `@r`

```
@r{n{S}}= {a -->#, S-->a}
```


NEPs computers

NEPsLingua syntax (by examples)

- Filters
  - We have grouped the different filters of the literature in six types
    - Types 1, 2, 3, 4
    - Defined by means of regular expressions
    - Defined by sets of strings
  - Each node has an input and an oupt filter (suffixes `if` and `of`, respectively)
  - Each filter has its 'permitting and forbidding contexts' (prefixes `p` and `f`)
  - We use the followting syntax for regular patterns
    - Union `[]`
    - Intersection `][`
    - Empty string `#`
  - Examples of filters for several nodes

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

NEPs computers

NEPsLingua syntax (by examples)

- Connections (@C)
  - Complete graphs in a compatct form

```
@C=@complete
```

  - An explicit set of connections defined by means of pairs of nodes

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

- Stopping conditions (@S)
  - Two consecutive equal configurations@no_change
  - Take a maximum number of steps @max_steps
  - Some node is no more empty @non_empty_node

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


NEPs computers

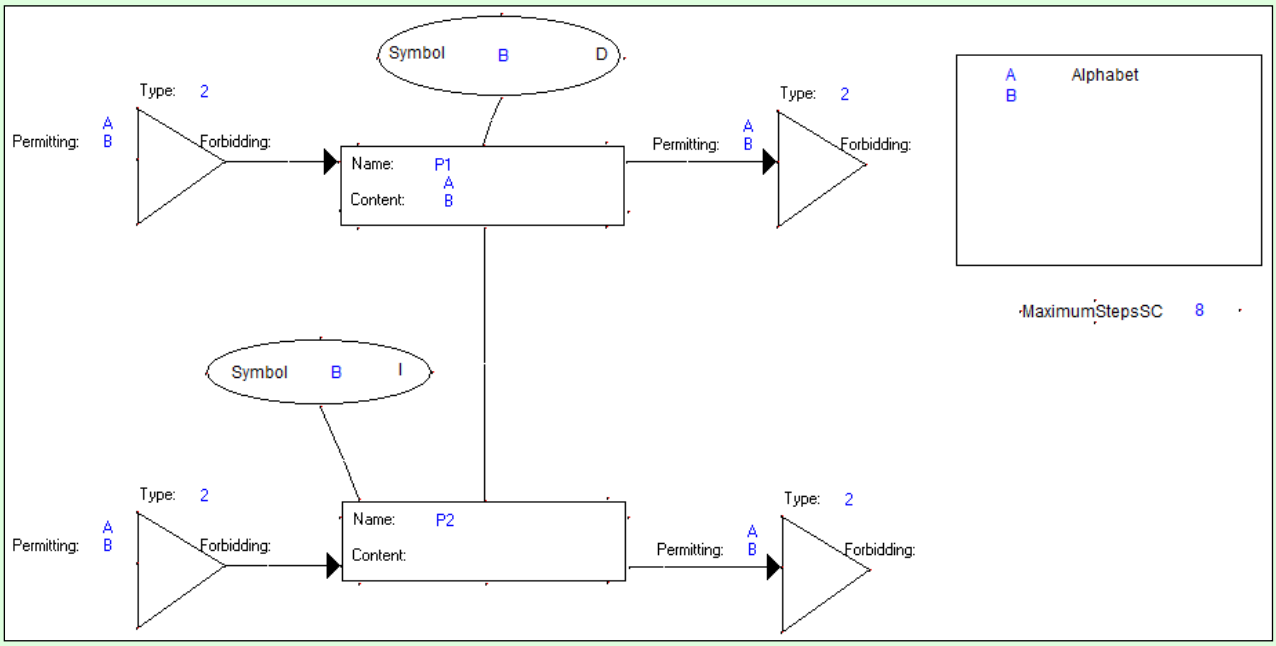
NEPsLingua syntax (complete examples)

- We will consider a toy NEP that
  - It has two nodes that, respectively, 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 the string A and adds again the symbol B.
  - The resulting string comes back to the initial node and the same process takes place again.

NEPs computers

NEPsLingua syntax (complete examples)

- It has the following NEPsVL representation



NEPs computers

NEPsLingua syntax (complete examples)

- And the following xml specification for jNEP

```
<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>
```

NEPs computers

NEPsLingua syntax (complete examples)

- And the following xml specification for jNEP

```
<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>
```


NEPs computers

NEPsLingua syntax (complete examples)

- And the following NEPsLingua specification

```
@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}
```

- It is obvious the liter size and greater simplicity of the NEPsLingua program
- These advantages are even greater as the NEPs becomes more complex

NEPs computers

NEPsLingua semantics (working on it)

- The semantic constraints for every NEPs-Lingua program are outlined below:
  - It contains 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 computers

NEPsLingua code generators (soon)

- ‘On demand’ at least for
  - jNEP (Xml configuration file)
  - Our cluster platform

NEPs computers

An overview of a possible architecture for NEPs computers

- Layout of the architecture
- Solving an instance of the Hamiltonian path problem with NEPs
- Programming tools for NEPs
  - Graphic simulation environment on ‘classical’ architectures
  - Simulation on clusters of computers
  - Textual programming languages independent on the architecture
- **Some applications**
  - **Solving NP-problems with lineally bounded resources**
  - Some applications of NEPs to language processing

NEPs computers

Solving NP problems with lineally bounded resources

- This possibility has been shown throughout the talk
  - Hamiltonian path problem
- It is easy to generalize the use of NEPs (jNEP and our cluster platform) for other NP problems

NEPs computers

An overview of a possible architecture for NEPs computers

- Layout of the architecture
- Solving an instance of the Hamiltonian path problem with NEPs
- Programming tools for NEPs
  - Graphic simulation environment on ‘classical’ architectures
  - Simulation on clusters of computers
  - Textual programming languages independent on the architecture
- **Some applications**
  - Solving NP problems with lineally bounded resources
  - **Some applications of NEPs to language processing**

NEPs computers

Parsing of formal and natural languages run on clusters

- Natural Language Processing (NLP) is a subfield of Computational Linguistics that focuses on building automatic systems able to interpret or generate information written in natural language

- The syntactical level is one of the linguistic levels a typical NLP system has to cover:
  - To use parsers to detect valid structures in the sentences, usually in terms of a certain grammar.

NEPs computers

Other approaches

- Earley (one of the most efficient algorithms) and its derivatives provide parsing in polynomial time, with respect to the length of the input:
  - Linear in the average case
  - n^2 in the worst case for unambiguous grammars and
  - n^3 in the worst case for ambiguous grammars

- Bel Enguix, G., Jimenez-Lopez, M. D., Merca¸s, R. and Perekrestenko, A. (2009): Networks of evolutionary processors as natural language parsers. In proc. ICAART 09.
  - Outlines a bottom up approach to natural language parsing with NEPs
  - Extends NEPs with context dependent rules
  - Linear performance in the average case

NEPs computers

PNEPs: motivation

- PNEP goal:
  - NEP based and top-down strategy
  - Keeping expressive power of NEP processors bounded
  - Get a similar performance
  - Get all the possible derivations of each string
  - Without modifying the grammar
    - No normal form needed
    - Ambiguity allowed
    - Recursive, erasing, renaming rules allowed

NEPs computers

PNEPs: main extension to NEP

- PNEPs use context free rules rather than classic substitution rules $A \rightarrow B$, $A \in V$ and $\underline{B \in V^+}$ instead of $A \rightarrow B$, $A,B \in V$
  - It is a classic way of handling strings in theoretical computer science
  - It does not imply expressive power greater than regular
- In Csuhaj-Varju, E. Martin Vide, C., Mitrana, V.: HNEPs are Computationally Complete. Acta Informatica; NEPs need different sets of additional nodes when simulating the application of context free rules:
  - To rotate the string
  - To actually apply the rule and
  - To delete additional symbols
- We can consider this mechanism as a subrutine so we always can build a classic NEP equivalent to any PNEP.
- This technique is sketched as follows:

NEPs computers

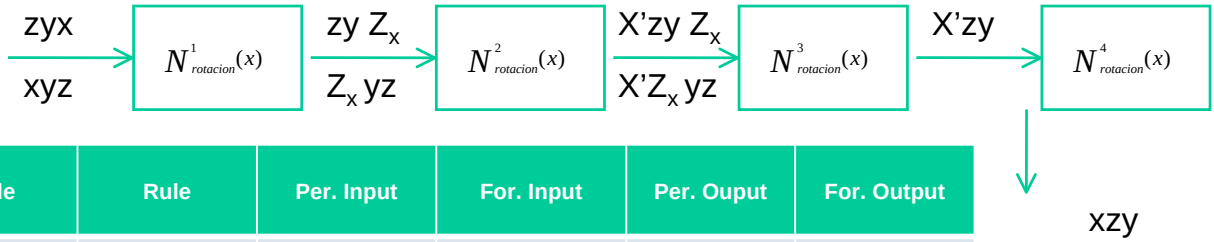
NEPs to apply context free rules

- Locating the non terminal by rotation
  - Each symbol in the alphabet needs these additional nodes

Node	Rule	Per. Input	For. Input	Per. Ouput	For. Output
$N^{1}_{rotacion}(X)$	X := Z_x	{$}	U \ (N ∪ T)	{Z_x}	
$N^{2}_{rotacion}(X)$	ε := X' (left)	{Z_x}	{X'}		
$N^{3}_{rotacion}(X)$	Z_x := ε (right)	{Z_x, X'}		{X'}	{Z_x}
$N^{4}_{rotacion}(X)$	X' := X	{X'}	{Z_x}		

NEPs computers

NEPs to apply context free rules



Node	Rule	Per. Input	For. Input	Per. Ouput	For. Output
$N^{1}_{rotacion}(X)$	X := Z_x	{$}	U \ (N ∪ T)	{Z_x}	
$N^{2}_{rotacion}(X)$	ε := X' (left)	{Z_x}	{X'}		
$N^{3}_{rotacion}(X)$	Z_x := ε (right)	{Z_x, X'}		{X'}	{Z_x}
$N^{4}_{rotacion}(X)$	X' := X	{X'}	{Z_x}		

NEPs computers

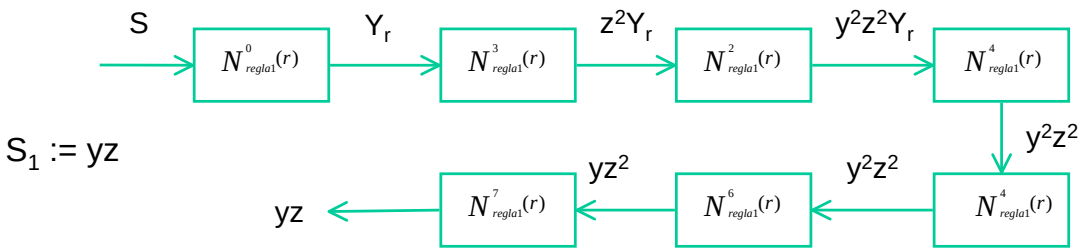
NEPs to apply context free rules

- Applying the rule
  - Each rule r S →x, where x = $x_1x_2…x_{pr}$
  - Needs the following additional nodes: (k ∈ {1,..,p})

Node	Rule	Per. Input	For. Input	Per. Ouput	For. Output
$N^{0}_{regla1}(r)$	S := Y_r		U \ (N ∪ T)	{Y_r}	
$N^{k}_{regla1}(r)$	ε := x^r_k (left)	Y_r ∪ {$x^r_{k+1}…x^r_{pr}$}	x^r_k		
$N^{p_r+1}_{regla1}(r)$	Y_r := ε (right)	Y_r ∪ {$x^r_1…x^r_{pr}$}		{$x^r_1…x^r_{pr}$}	Y_r
$N^{p_r+1+k}_{regla1}(r)$	x^r_k := x_k	x^r_k	Y_r		x^r_k

NEPs computers

NEPs to apply context free rules



Node	Rule	Per. Input	For. Input	Per. Ouput	For. Output
$N^{0}_{regla1}(r)$	$S := Y_r$		$U \setminus (N \cup T)$	$\{Y_r\}$	
$N^{k}_{regla1}(r)$	$\varepsilon := x^r_k$ (left)	$Y_r \cup \{x^r_{k+1} \ldots x^r_{p_r}\}$	x^r_k		
$N^{p_r+1}_{regla1}(r)$	$Y_r := \varepsilon$ (right)	$Y_r \cup \{x^r_1 \ldots x^r_{p_r}\}$		$\{x^r_1 \ldots x^r_{p_r}\}$	Y_r
$N^{p_r+1+k}_{regla1}(r)$	$x^r_k := x_k$	x^r_k	Y_r		x^r_k

NEPs computers

Context free grammar ➔ PNEP translation

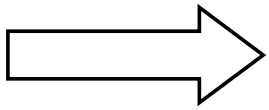
- A PNEP is built from a context free grammar in the following way:
  - We assume that each derivation rule in the grammar has a unique index that can be used to reconstruct the derivation tree.
  - There is a node for each non terminal (*).
    - Each node applies to its strings all the derivation rules for its non terminal.
    - The filters, as well as the graph layout, allow all the nodes to share all the intermediate steps in the derivation process.
  - An additional output node, is used to contain the ***parsed string***: a version of the input, enriched with information that will make it possible to reconstruct the derivation tree (the rules indices). This node will only allow enter this parsed string
  - The graph is complete.
  - The initial content of the node that corresponds to the axiom is the axiom itself.
- The designer could choose different contents for each node.
  - From just a node for all the derivation rules
  - To a node for each different right hand side (each rule) in the grammar

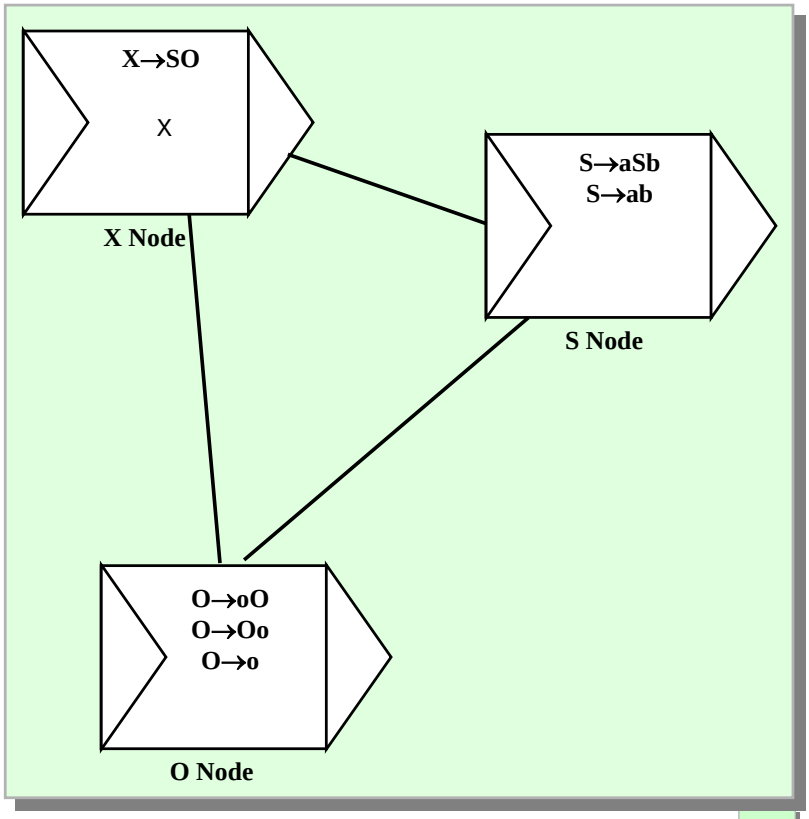
NEPs computers

Context free grammar ➔ PNEP translation

- For example (without the additional output processor, filters will be omitted for clarity):

1: X→SO
2: S →aSb
3: S →ab
4: O→oO
5: O→Oo
6: O→o

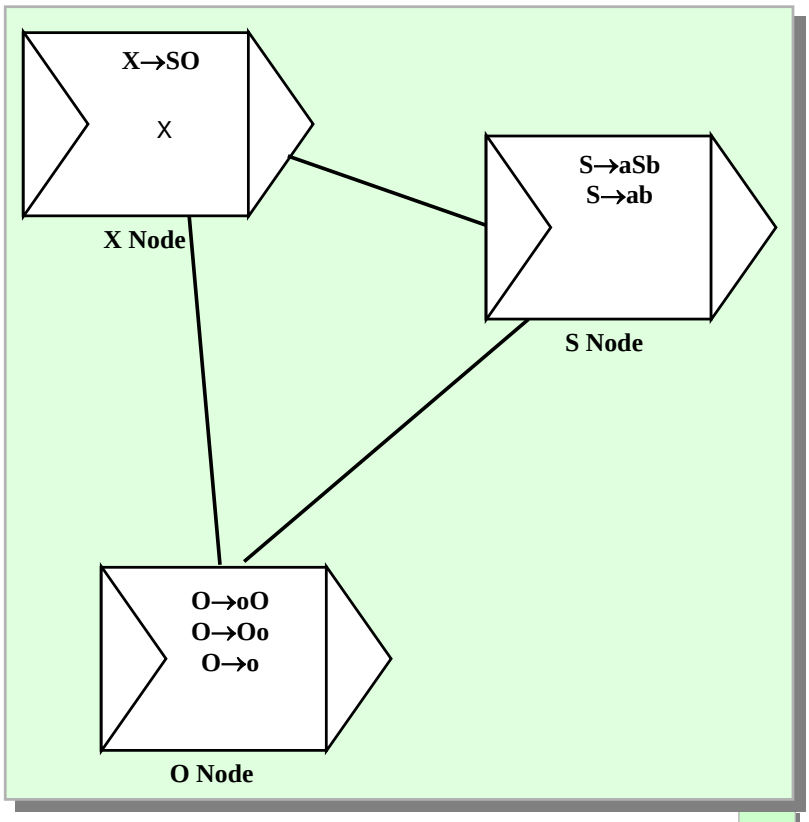




NEPs computers

Context free grammar ➔ PNEP translation

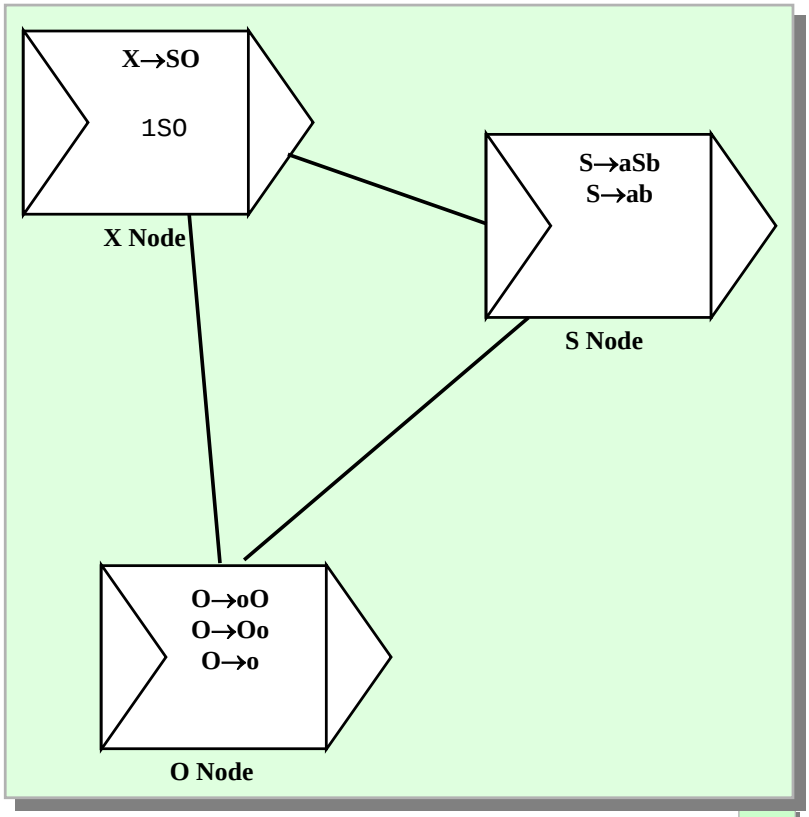
- The computation starts



NEPs computers

Context free grammar ➔ PNEP translation

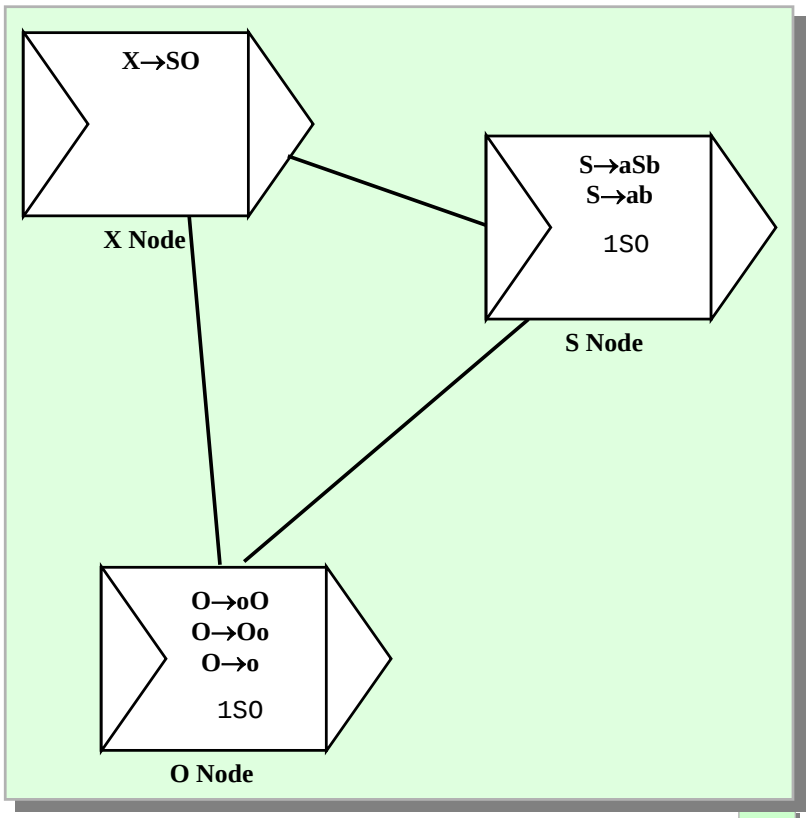
- The node of the axiom applies its rule



NEPs computers

Context free grammar ➔ PNEP translation

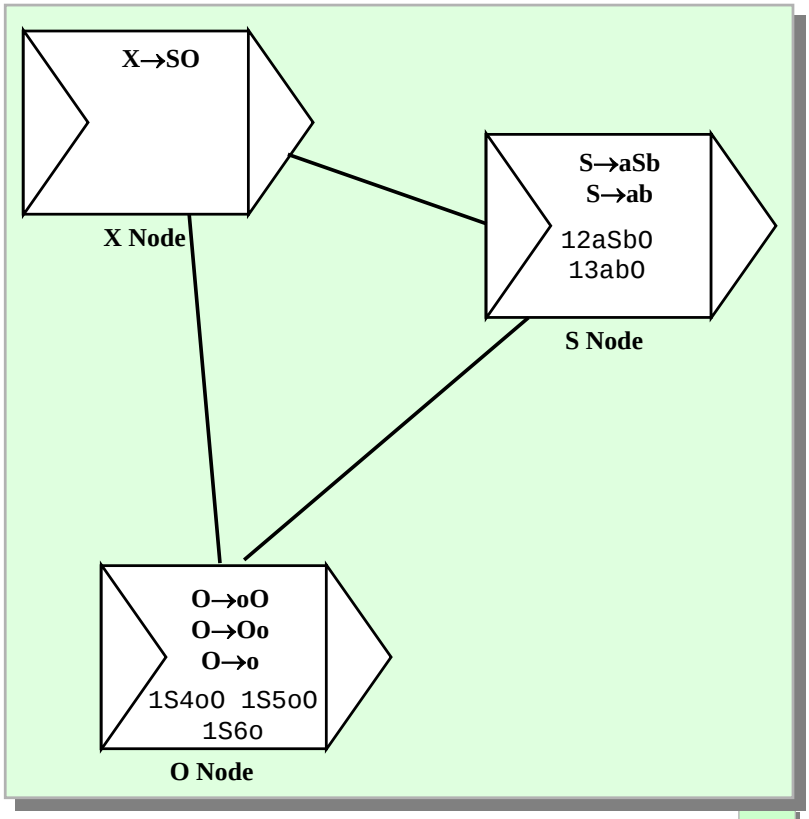
- Each node gets the strings it can modify



NEPs computers

Context free grammar ➔ PNEP translation

- And applies to them its rules

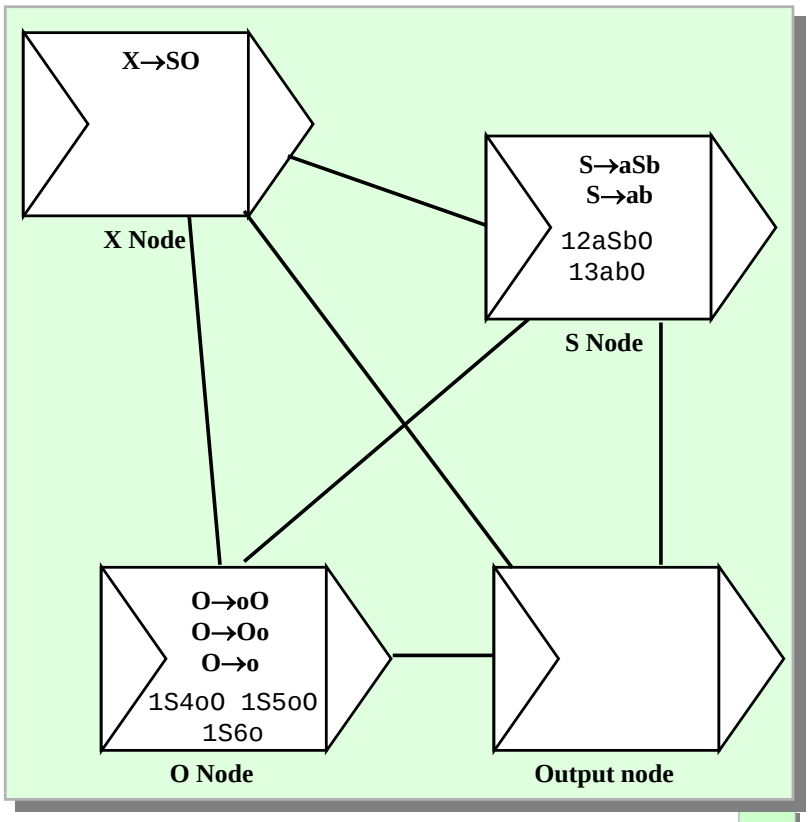


NEPs computers

Context free grammar ➔ PNEP translation

- The output node builds a regular expression from the string being parsed with the possible sequence of applied rules between its symbols.
- For example for the string abo we will write for this regular expression the following

r*ar*br*or*

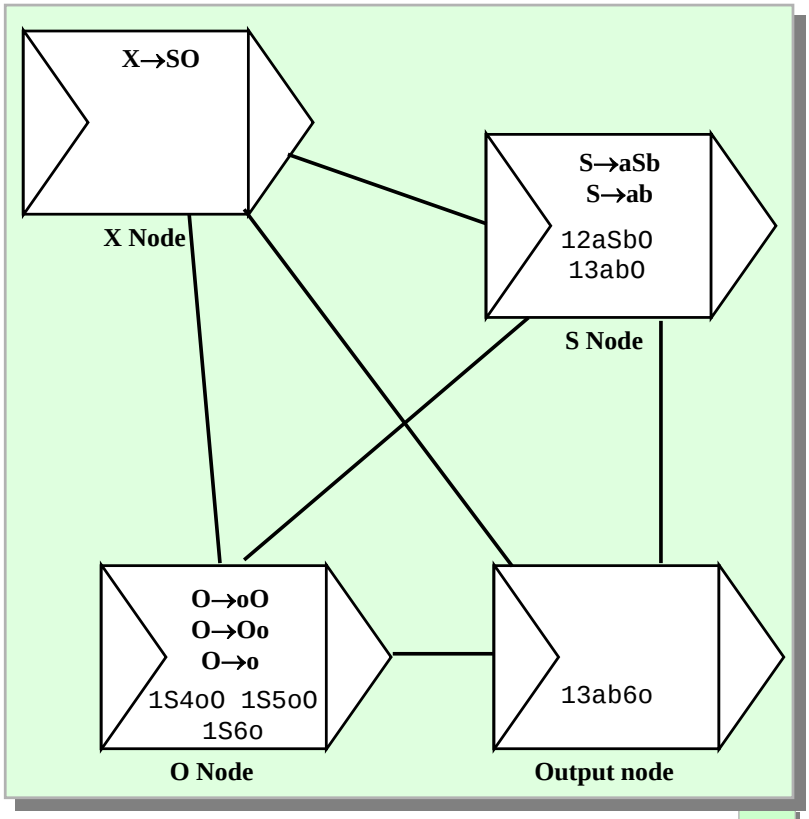


NEPs computers

Context free grammar ➔ PNEP translation

- After some additional steps computation finishes and the output node contains:

  `13ab6o`



NEPs computers

NEPsLingua for PNEPs

- Now, that we now NEPsLingua we can appreciate the simplicity and closeness of its programs to the grammar corresponding to a PNEP.
- PNEP for parsing de context free grammar of these derivation rules

  `X → SO`

  `S → aSb | ab`

  `O → Oo | oO | o`
- Can be written in NEPsLingua as follows

```
@A={X,S,a,b,o,O}
@N={final}+{n{symbol}:symbol->{X,S,O}}
@c{n{X}}={X}
@r{n{X}}= {X-->SO}
@r{n{S}}= {S-->aSb, S-->ab}
@r{n{O}}= {O-->o, O-->oO, O-->Oo}
@pif{n{X}}={1,{X}} @pif{n{S}}={1,{S}} @pif{n{O}}={1,{O}}
@C=@complete
@S={ @non_emtpy_node={final} }
```

NEPs computers

NEPsLingua for PNEPs

- Only for comparison purposes, a possible jNEP xml specification file for the same NEP is shown below

```
<?xml version="1.0"?>
<NEP nodes="5">
<GRAPH>
<EDGE vertex1="0" vertex2="4"/>
<EDGE vertex1="0" vertex2="3"/>
<EDGE vertex1="1" vertex2="4"/>
<EDGE vertex1="1" vertex2="3"/>
<EDGE vertex1="2" vertex2="4"/>
<EDGE vertex1="2" vertex2="3"/>
</GRAPH>
<EVOLUTIONARY_PROCESSORS>
<NODE initCond="" id="0">
<EVOLUTIONARY_RULES>
<RULE ruleType="substitution" actionType="ANY" symbol="S"
newSymbol="0-0_a_b"/>
<RULE ruleType="substitution" actionType="ANY" symbol="S"
newSymbol="0-1_a_S_b"/>
</EVOLUTIONARY_RULES>
<FILTERS>
<INPUT type="1" permittingContext="S" forbiddingContext=""/>
</FILTERS>
</NODE>
<NODE initCond="" id="1">
<EVOLUTIONARY_RULES>
<RULE ruleType="substitution" actionType="ANY" symbol="0"
newSymbol="1-0_o"/>
<RULE ruleType="substitution" actionType="ANY" symbol="0"
newSymbol="1-1_0_o"/>
<RULE ruleType="substitution" actionType="ANY" symbol="0"
newSymbol="1-2_o_0"/>
</EVOLUTIONARY_RULES>
<FILTERS>
<INPUT type="1" permittingContext="0" forbiddingContext=""/>
</FILTERS>
</NODE>
<NODE initCond="X" id="2">
<EVOLUTIONARY_RULES>
<RULE ruleType="substitution" actionType="ANY" symbol="X"
newSymbol="2-0_S_0"/>
</EVOLUTIONARY_RULES>
<FILTERS>
<INPUT type="1" permittingContext="X" forbiddingContext=""/>
</FILTERS>
</NODE>
<NODE initCond="">
<EVOLUTIONARY_RULES>
<RULE ruleType="deletion" actionType="RIGHT" symbol=""/>
</EVOLUTIONARY_RULES>
<FILTERS>
<INPUT type="2" permittingContext=
"a_a_b_b_o_o_o_0-0_0-1_1-0_1-1_1-2_2-0_S_0_X" forbiddingContext=""
/>
</FILTERS>
</NODE>
<NODE initCond="">
<EVOLUTIONARY_RULES>
<RULE ruleType="deletion" actionType="RIGHT" symbol=""/>
</EVOLUTIONARY_RULES>
<FILTERS>
<INPUT type="RegularLangMembershipFilter" regularExpression=
"[0-9\-]*(a)[0-9\-]*(a)[0-9\-]*(b)[0-9\-]*(b)[0-9\-]*(o)[0-9\-]*(o)
[0-9\-]*(o)"/>
<OUTPUT type="1" permittingContext="" forbiddingContext=
"a_a_b_b_o_o_o"/>
</FILTERS>
</NODE>
</EVOLUTIONARY_PROCESSORS>
<STOPPING_CONDITION>
<CONDITION type="NonEmptyNodeStoppingCondition" nodeID="4"/>
</STOPPING_CONDITION>
</NEP>
```


NEPs computers

PNEPs demo

- In the computer

NEPs computers

PNEPs implemented improvementes: bad terminals filtering

- The basic PNEP model generates strings from the axiom in a rather blind way.
- No terminal string is discarded until it is larger that the target string.
- Parsers usually discard parsing choices as soon as a terminal is not in its right place (usually in a left to right order).
- We have implemented a naive filter that removes those strings that contains terminals that do not belong to the target string.

NEPs computers

PNEPs implemented improvementes: depth first non terminal choice

- The basic PNEP model explicitly stores a different string for each different derivation that only differs in the order in which non terminals are chosen. All these derivations corresponds to the same derivation tree (we are not talking about ambiguity in this case)
- We actually only need a string to stand for each derivation tree. Top down techniques usually solve this difficulty by applying some given order (usually the left-most non terminal is chosen)
- In the NEP config file, this rules are described with the following syntax:

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

- For that porpuse we have defined a new NEP rule. Given the rule r: $A \rightarrow a$, the action of the rule on a word w, r(w) is defined by

$r(w)=\{t \mid w= w_1Aw_2$ and $t=w_1aw_2$ and only_contains_terminals(w_1), w_1,w_2 are words over V$\}$

NEPs computers

PNEPs non implemented improvementes: left and right-corner filtering

- We have previously described the naive “bad terminals” filter we have implemented
- We can add Left and right-corner filtering by checking both ends of the generated strings: those strings with misplaced terminals in any of the ends (left or right) could be discarded.
- This test could be extended to the left-most (respectively right-most) non terminal by means of the classic first (respectively last) sets.

NEPs computers

PNEPs further research lines

- We plan:
  - To add PNEP to our corpus and compare it with actual NLP parsers (Freeling, etc.)
  - To add semantics to PNEP to design a compiler tool
  - To tackle some semantic NLP problem by means of the previous model

THANK YOU FOR YOUR ATTENTION