Different approaches for development tools for natural computers: grammar driven vs. model driven approaches

David Fernández[1], Francisco Saiz[1], Marina de la Cruz[1], and Alfonso Ortega[1]

Departamento de Ingeniería Informática,
Escuela Politécnica Superior,
Universidad Autónoma de Madrid, Spain
david.fernandez@iic.uam.es, {Francisco.Saiz, Marina.Cruz, alfonso.ortega}@uam.es

Abstract. In this paper we will compare our first steps in two different approaches to define programming languages for NEPs (one bio-inspired model of computation in which our research group is interested). The classic approach proposed by the literature several decades ago is focused on the grammar of the syntax of the language being defined. Recently the focus is moved to a formal description (model) of the model of computation for which the programming language is being designed. This approach is called *model driven*. The designer simply adds syntax, semantics checks and translation routines to the different elements of the model that are applied. Programming is usually understood as *instantiating the model*. After introducing the main characteristics of each model for this particular case some conclusions and further research tasks are discussed.[1]

1 Motivation

Our research group is interested in providing the scientific community with powerful tools to develop software applications to solve problems by means of new computing devices (most of them inspired by the way in which Nature solves difficult tasks). One of these new computing paradigms are Networks of Evolutionary Processors (NEPs) that mainly consist of a set of processors each of which performs a very simple process on the set of strings that they contain. These computing processors are connected as nodes of a graph. NEPs are not allowed to dynamically change the topology of their nets. The complete system alternates computing and communicating steps. In the computing or evolving step the processors change their contents simultaneously. In the communicating step they share some of their strings with the rest of the processors to which they are connected in the net. An important component of NEPs are the filters used by the processors to decide which strings enter and leave the nodes. The classic family of NEPs uses two kinds of filters (input and ouput) each of which is defined by means of a

[1] Work partially supported by the Spanish Ministry of Science and Innovation under coordinated research project TIN2011-28260-C03-00 and research project TIN2011-28260-C03-02 and by the Comunidad Autónoma de Madrid under research project e-madrid S2009/TIC-1650

couple of components (forbidden and permitted strings) One of the main characteristics of NEPs is that they are intrinsically parallel and some instances of them have the same computational power of Turing's machine giving the possibility of designing algorithms for NP-problems that improve the temporal performance of their Turing counterparts. More detailed formal definitions and properties could be found in [1]

One of the most interesting features of bio-inspired computers, like NEPs, is their intrinsic parallelism. We can design algorithms for them that could improve the exponential performance of their classic versions, but, unfortunately there are neither no real computers nor programming languages and software engineering tools for almost any bioinspired model. So, running the algorithms usually involves simulating the model in a conventional (von Neumann) computer and their design depends on the simulator and is far for being a standard procedure: some simulators read configuration files, some others only offer the use of their graphical interface. This paper compares two different approaches (grammar and model driven) that our group is following to design programming languages and development environments for NEPs.

2 Introduction

2.1 Grammar driven approach

We could name the traditional approach to design programming languages and develop their processors (compilers and interpreters) as *grammar driven approach* (GDA)

GDA takes the following steps [2]:

- Design a context free grammar for the complete syntax of the language
- Add the semantics and code generator routines by means of an attribute system. In this way the initial context free grammar becomes an attribute grammar
- Choose the parsing technique more adequate to the complexity of the language. There are mainly two types of parsing families: top-down and bottom-up. The most popular and powerful of their algorithms are respectively LALR(1) (from *look ahead L(eft to right) R(ightmost derivation)* taking into account just 1 symbol from the input) and LL(*)[3](from L(eft to right) L(eftmost derivation) using as many terminals symbols (*) from the input as needed)
- Develop the compiler-interpreter usually with the help of automatic generators of compilers like bison [4] or CUP [5] (LALR) or ANTLR [3] (LL(*))

2.2 Model driven approach

One of the most promising and powerful approaches to software engineering, including the definition of program languages, is the so called *model driven approach* (MDA).

This approach is based on a formal description of the model of computation (for example, and typically, an extension of an UML [2] model). The model is enriched with the syntax that will be used by the programmer in his programs.

[2] Unified Modelling Language is a standard language widely used to model in computer science

There exists some software applications or developing environments that support this approach. These environments usually works as follows:

The system offers a module to describe the (UML) model. Different options, modules or plug-ins are available to add different kinds of syntax (textual, visual) to each component of the model. Once the model and its syntaxes are refined and finished, a new development environment is provided by the systems that is particularized and specialized to describe specific cases of the model of interest, that is, the user can write its own programs using its own new syntax. This new development environment is usually run under the control of the system (by means of, for example, a new graphic canvas or a new text editor window) offering, in this way, all the functionality of the system. When the user is writing his programs, the system is actually instantiating the model underlying and keeping in memory an image of the program being written. It can check the syntax at the same time the program is written. Semantic constraints could be added (and checked) in a similar way to syntax and it is also possible to add the procedure by means of which each element of the model is translated into a different representation. The reader can easily understand that the results provided by these kind of tools could include the compilers or interpreters for the programming language defined as well as development environment specialized for the language defined.

3 Grammar driven approach for natural computers: for example NEPs

Some of the authors of this paper have previously followed the first steps of this approach for NEPs. They have proposed NEPsLingua as a possible syntax for NEPs [6] and checked the viability of building its parsers with ANTLR.

NEPsLingua proposes a syntax for NEPs close to the mathematical notation used in their formal specification. Further details of the complete syntax can be found in [6]

The following listings show, as examples, some NEPsLingua versions of NEPs:

Examples of a NEPsLingua Program

```
@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}
```

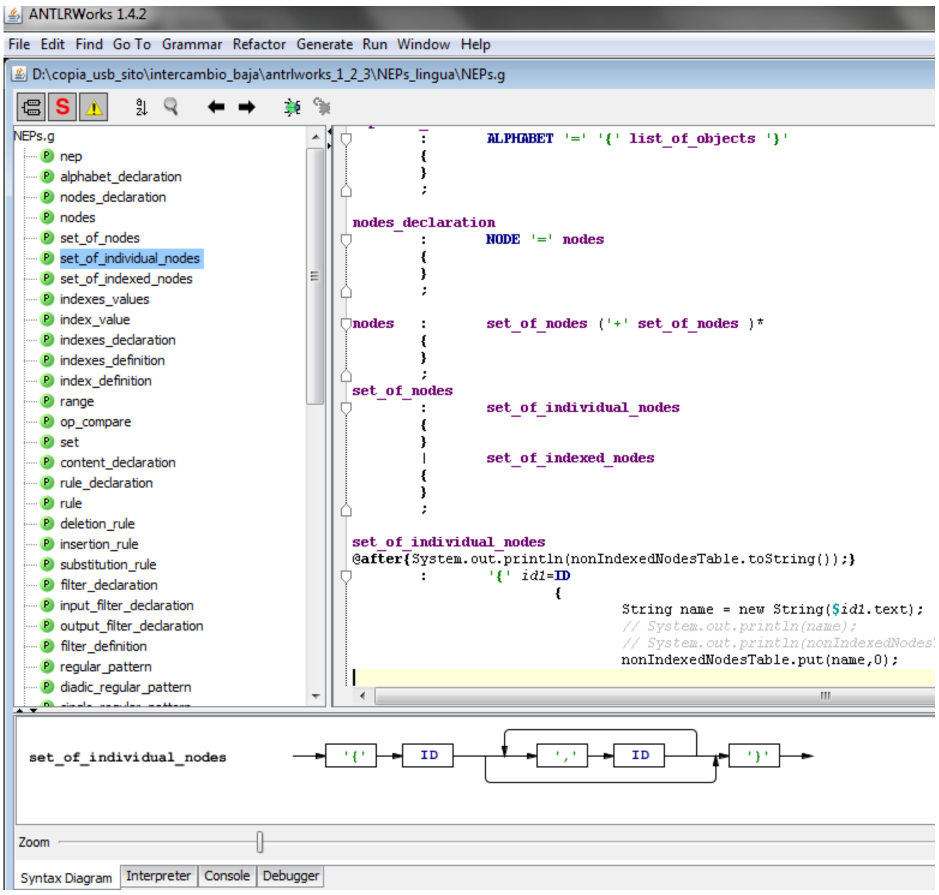
(This very simple example of NEP has two nodes that delete and insert the symbol B. The initial word AB travels from one node to the other. The first node removes the symbol B from the string before leaving it in the net. The other node receives string A and adds symbol B again. The resulting string comes back to the initial node and the same process takes place again.)

```
@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
```

(NEPsLingua program for a NEP for parsing the grammar implicit in the rules of the nodes, X is the axiom)

Figure 1 shows an example of ANTLR with the grammar defined for NEPsLingua. You can see the rules defining some of the elements of a NEP. Specifically you can see a fragment that describes the different classes of set of nodes in NEPsLingua: individual and indexed.



```
            :       ALPHABET '=' '{' list_of_objects '}'
            {
            }
            ;

nodes_declaration
            :       NODE '=' nodes
            {
            }
            ;

nodes       :       set_of_nodes ('+' set_of_nodes )*
            {
            }
            ;
set_of_nodes
            :       set_of_individual_nodes
            {
            }
            |       set_of_indexed_nodes
            {
            }
            ;

set_of_individual_nodes
@after{System.out.println(nonIndexedNodesTable.toString());}
            :       '{' id1=ID
                        {
                                String name = new String($id1.text);
                                // System.out.println(name);
                                // System.out.println(nonIndexedNodes
                                nonIndexedNodesTable.put(name,0);
```

Fig. 1. ANTLR window showing some of the elements of the LL(*) grammar for NEPsLingua

4 An example of model driven approach for NEPs

One of the platforms that supports MDA is conformed by the modelling plug-ins of Eclipse (`http://www.eclipse.org`). Although the reader could find in its website different options, this paper only describes the following:

– Emfatic, for the model extended with visual syntax
– Xtext, for adding textual syntax
– Some features added by Eclipse to the integrated development environment

We will show in this paper some fragments of Eclipse files describing our languages for NEPs. A complete description of the characteristics of these tools is out of the scope of this work and can be found in `http://www.eclipse.org`

Definition of the basic UML model

Eclipse provides a graphic interface to draw the basic UML model. It contains the typical elements of every UML classes diagram (classes, interfaces, relationships, attributes, etc.). This models are named *Ecore* diagrams by Eclipse.

In the present work we have not defined an explicit *Ecore* diagram. We describe above the technique used instead.

Definition of the UML model and its visual appearance

In this paper we will show how to define he model (with its visual appearance at the same time) by means of the *Emfatic* plug-in. When the model is defined as an Ecore diagram, the user has to manually add to the textual representation of the Ecore diagram some *emfatic annotations* to add the visual syntax.

In our example we will represent the processors, the filters and the rules by means of respectively rectangles, round rectangles and ovals. Permitting filters are coloured in green, while forbidden in red.

It is easy to find in the following fragment of *Emfatic* code the annotations that specify these visual appearances (in particular rules and colours of filters).

```
@gmf.node(label = "contentName", figure="ellipse")
class ContentRules {
    attr String contentName;
    @gmf.link(style="dash")
    ref Processor ownerRules;
    @gmf.compartment(foo="bar")
    val Rule[*]  rules;
}
...
@gmf.node(border.color = "0,255,0")
class PermittingInputFilter extends InputFilter{}

@gmf.node(border.color = "255,0,0")
class ForbbidingInputFilter extends InputFilter{}
```

(A fragment of the emfatic code with visual annotations for NEPs)

Adding the textual appearance

Eclipse provides a procedure to automatically associate the following default textual syntax for its models: each element should be preceded by its name and its possible contents appears between brackets.

It is easy to realize after reading the following fragment of the Xtext file for NEPs that it is very similar to the context free grammar associated with the default syntax. It is easy also to identify the *standard Xtext syntax*: words `'NEP'`, `'processors'`, `'rules'`, `'Processor'` and symbols `'{'`, and `","` have to be literally written in the programs and could be (easily) changed by simply editing this file.

```
NEP returns NEP:
    {NEP} 'NEP'
    '{' ('processors' '{' processors+=Processor
                         ( "," processors+=Processor)* '}' )?
        ...
        ('rules' '{' rules+=ContentRules
                         ( "," rules+=ContentRules)* '}' )?
        ...    '}';
Rule returns Rule:
    InsertRule | DeriveRule | DeleteRule | RegularExpresionRule
    | SubStitutRule;
Processor returns Processor:
    {Processor} 'Processor'
    name=EString  '{'        ...          '}';
```

(A fragment of the Xtext code for NEPs showing the similarities with the context free grammar for the default syntax)

Although it is very easy to modify some aspects of the Xtext syntax (you can, for example, freely change the delimiters to use other symbols different from brackets or the tags used to identify each element) it is difficult to include big changes without modifying the model. This is one of the main drawbacks we encounter in this approach. GDA gives total freedom in the design of the syntax but in MDA (because of the tight relationship between the model and the syntax) some syntactic constructs should be translated into new elements in the model. This elements have obviously only syntactic meaning, that is, they do not actually belong to the model itself because a different syntax could exclude them.

The following examples show the difficulties to make the Xtext default syntax more similar to NEPsLingua.

In NEPsLingua, for example, it is not mandatory to put together all the components (initial contents, rules, filters) of each node, while the default Xtext syntax put all these elements inside their node.

You can compare in the following listings how the contents of node n_0 (of the simplest example of NEP described previously) have to be together in the default Xtext version while this is not mandatory in NEPsLingua.

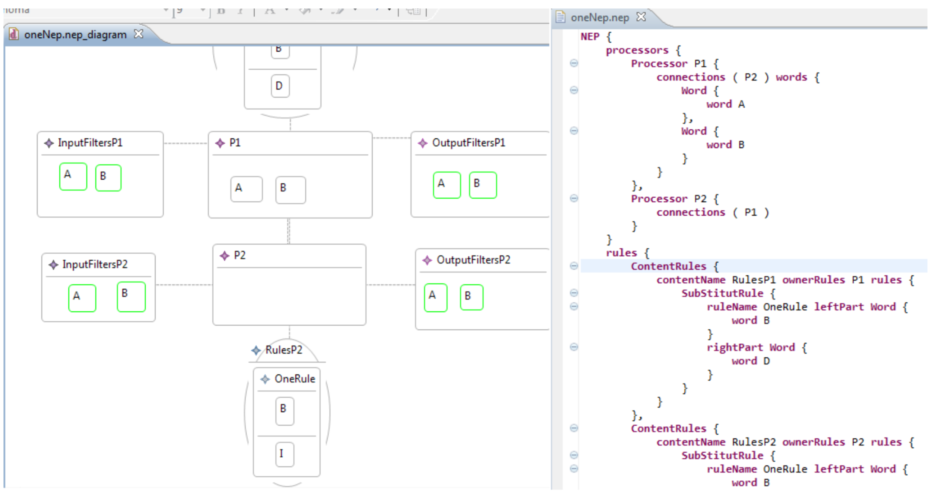
```
@c{n{0}}={A,B}
@r{n{0}}={B-->#}
```

```
@r{n{1}}={#-->B}

@c{n{0}}={A,B}
@r{n{1}}={#-->B}
@r{n{0}}={B-->#}
```

(Two equivalent ways of specifying the contents of a couple of nodes using NEPsLingua)

In figure 2 the simplest example of NEP is (re)described with our visual syntax and also with the default textual Xtext syntax. It is easy to find the differences between NEPsLingua and the default Xtext syntax, and the difficulties to make them similar.



```
NEP {
    processors {
        Processor P1 {
            connections ( P2 ) words {
                Word {
                    word A
                },
                Word {
                    word B
                }
            }
        },
        Processor P2 {
            connections ( P1 )
        }
    }
    rules {
        ContentRules {
            contentName RulesP1 ownerRules P1 rules {
                SubStitutRule {
                    ruleName OneRule leftPart Word {
                        word B
                    }
                    rightPart Word {
                        word D
                    }
                }
            }
        },
        ContentRules {
            contentName RulesP2 ownerRules P2 rules {
                SubStitutRule {
                    ruleName OneRule leftPart Word {
                        word B
```

Fig. 2. Eclipse windows showing both, the visual and Xtext default textual appearance of the simplest example of NEP

Other interesting features of Eclipse modelling plug-ins

Eclipse provides us with very interesting features. We will comment two of them: getting a specific development environment for writing code with the new language and the possibility of integrating different views of the programs in a synchronized way.

One of the goals of Eclipse is to generate an *instance* of the development environment (Eclipse *itself*) that recognizes the languages just defined for the new model as one of *its* languages. This new environment offers all the features to the programmer that Eclipse has, for instance, it highlights and checks the syntax, auto-completes the code and suggests by means of pop-up menus different options to the programmer while he is working.

If the new instance of Eclipse includes the modelling plug-ins in such a way that allows the synchronization of different views (we have described in this paper a textual and a visual appearance for NEPs) of the same program, the user (programmer) of this development environment could change from a view to the other (between the visual and the textual editor) and, when he changes a view, the environment will automatically update the other.

Figures 3, 4 and 5 show this circumstance: firstly (fig. 3) you can see the new instance of Eclipse specific for NEPs showing a new (empty) NEP both in the textual and in the visual editor.

Once the programmer adds a new processor to the NEP by means of the visual editor (fig. 4), after saving this file, the environment automatically updates the textual view accordingly (fig. 5). We could have proceeded in the opposite way, modifying and saving the textual view and getting an automatic update of the visual view.

5 Conclusions

Grammar driven approach Among the advantages of GDA, we can highlight the solid theoretical model on which it is based, the power of its results and the soundness of the tools that have been checked for years. From our viewpoint, its main disadvantages are that its results do not usually include to offer a friendly development environment. The available tools are often libraries that are conceived to be called from the host code of the application in which the language will be included.

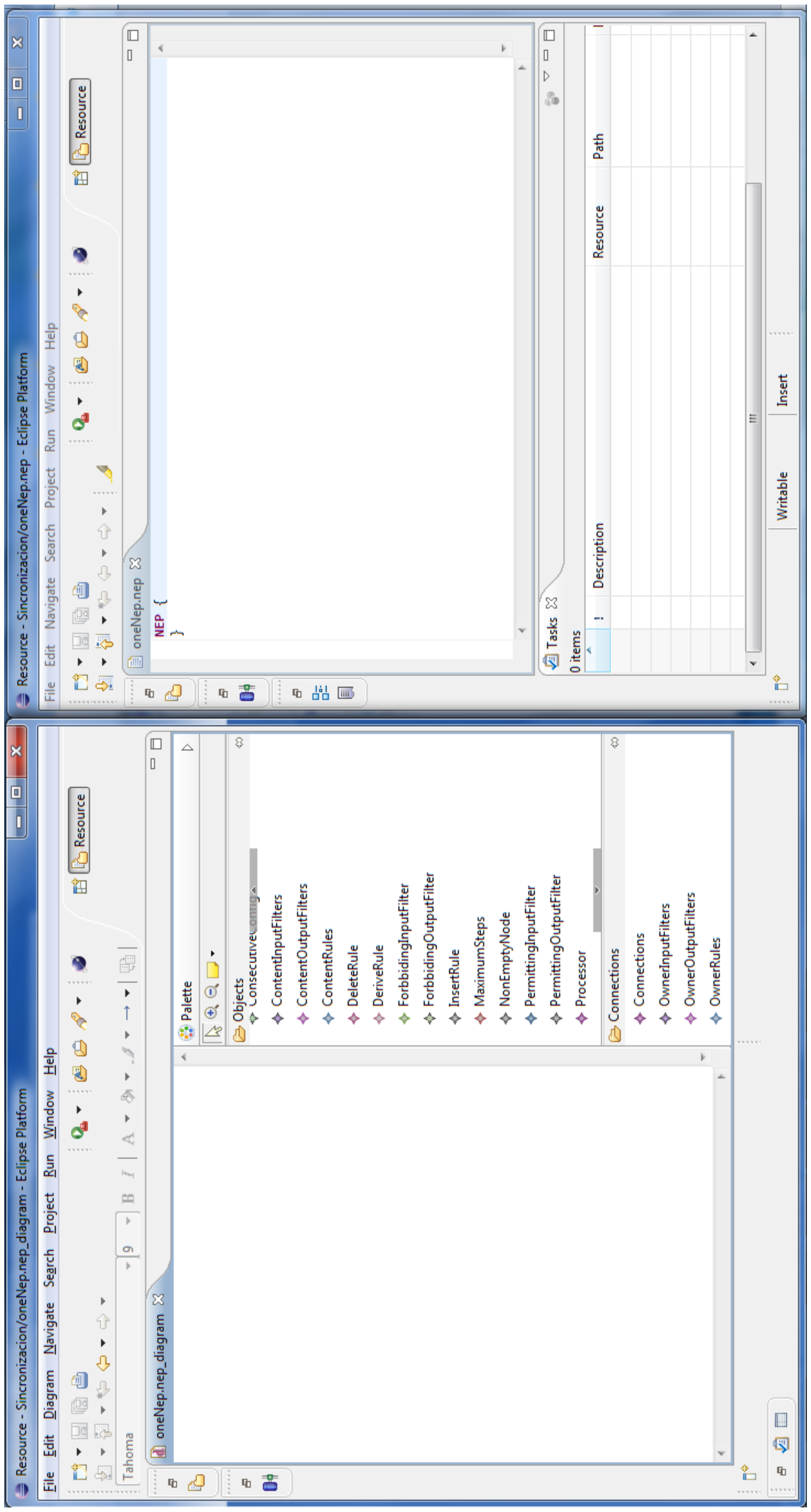
Model driven approach MDA offers to the designer a higher level of abstraction to define the model. The formalism (usually based on UML) could be more intuitive than grammars. Platforms that support MDA usually include providing the programmers with development environments, among their goals. Nevertheless, the novelty of the approach actually causes, in our opinion, its main drawbacks. The big amount of research effort that is being done in this domain produces lots of software tools that are neither always well documented nor properly supported. They are often tricky and obscure to handle and (what could be worse) nobody guarantees that the tools will be still supported after their learning process.

6 Further research lines

Regarding the MDA, in the future, we plan to focus our efforts in the design of a Xtext textual syntax more similar to NEPsLingua but which allows, in addition, to keep the model as free of syntactic artefacts as possible. Once we finish the design of the syntax (both textual and visual) we have to add semantics checks and translator routines. With respect to GDA we have to complete also the semantic analyser and the code generator. Both approaches (MDA, GDA) could be compared then for programming NEPs. Our conclusions will be useful when facing other bio-inspired models of computations. We are currently interested, for instance, in grammar systems, linguistics grammar systems or membrane based systems.

References

1. J. Castellanos, C. Martin-Vide, V. Mitrana, and J. M. Sempere.: Networks of evolutionary processors. Acta Informatica, Vol. 39(6-7): 517-529, 2003.



```
NEP {
}
```

Fig. 3. Eclipse windows showing both, the visual and textual editors for a new (and hence, empty) NEP

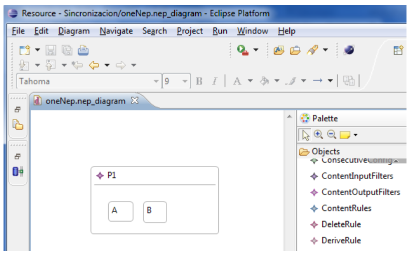
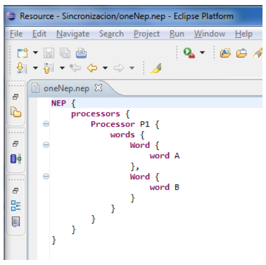


Fig. 4. Visual editor window showing a NEP with only one processor: it contains the strings "A" and "B"


Resource - Sincronizacion/oneNep.nep - Eclipse Platform
File Edit Navigate Search Project Run Window Help
oneNep.nep

```
NEP {
    processors {
        Processor P1 {
            words {
                Word {
                    word A
                },
                Word {
                    word B
                }
            }
        }
    }
}
```

Fig. 5. Textual editor window automatically updated by the environment

2. A.V. Aho, M.S. Lam, R. Sethi, J.D. Ullman, : Compilers: Principles, Techniques, and Tools, 2/E Prentice Hall 2007
3. T. Parr: The Definitive ANTLR Reference: Building Domain-Specific Languages The Pragmatic Bookshelf Raleigh. Norh Carolina. Dallas. Texas 2007
4. http://www.gnu.org/software/bison/
5. http://www.cs.princeton.edu/ appel/modern/java/CUP/manual.html
6. de la Cruz, M., Jiménez, A., del Rosal, E., Bel-Enguix, G., Ortega, A.: NEPs-lingua: a new textual language to program neps. In: Proceedings of ICAART 2011 (2011)