NiMo Syntax. Part 1

Silvia Clerici Guillermo Prestigiacomo
Cristina Zoltan
Departament de Llenguatges i Sistemes Informàtics
Universitat Politècnica de Catalunya Barcelona, Spain

July 15, 2014

Abstract

Many formalisms for the specification for concurrent and distributed systems have emerged. In particular considering boxes and strings approaches. Examples are action calculi, rewriting logic and graph rewriting, bigraphs. The boxes and string metaphor is addressed with different levels of granularity.

One of the approaches is to consider a process network as an hypergraph. Based in this general framework, we encode NiMo nets as a class of Annotated hypergraphs. This class is defined by giving the alphabet and the operations used to construct such programs. Therefore we treat only editing operations on labelled hypergraphs and afterwards how this editing operation affects the graph. Graph transformation (execution rules) is not covered here.

Resumen

Hay diversos formalismos para la especificación de sistemas concurrentes y distribuidos. Algunos de ellos basados en en el modelo de cajas y cuerdas. Ejemplo de ello es action calculi, rewriting logic y reescritura de grafos. Basado en este esquema general se presenta la sintaxis de NiMo como una clase de hypergrafos con anotaciones. Esta clase se describe dando el alfabeto y las operaciones para construir los programas NiMo. Por lo tanto tratamos solamente operaciones de edición de grafos sobre grafos con etiquetas y luego como las operaciones de edición afectan al grafo. Operaciones de transformación de grafos, resultado de la ejecución no son tratados en este documento.

1 NiMo Programming Language

When designing a programming language, the problem of exploiting multiple processors is in general addressed following two approaches: processing the program written to be run in a single processor architecture in order to identify the tasks that can be executed simultaneously, or include constructions in the language for the programmer to signal those opportunities.

The first approach is very costly and in general not very effective. The second approach places too much responsibility on the shoulders of the programmer.

A third approach in to relay on a coordination language that can use legacy code. NiMo is a programming language because is a coordination language for a set of basic processes specially designed for stream processing.

NiMo is a graphical programming language, where programs are process networks. Processes communicate and synchronize via unbounded channels mixing pure functional and data flow paradigms, and also have roles. Communication is decoupled and data can be shared.

However, there are no data races and the semantics of a program is the same regardless of the number of processors used in its execution.

The programmer's responsibility is reduced to finding solutions which can better exploit the available processors. Due to the evaluation model, it suffices to show the correctness of the solution on a single processor semantics, in order to ensure the correctness of the same solution exploiting multiple processors.

The language is graphical and highly interactive. When executing a program all the internal states are exhibited. The programmer can visualize step by step the parts of the system where the parallelism can be improved and changes can be made and tested directly on partially evaluated programs.

The model has proven to be highly efficient for problems where data flows (data intensive) and can be processed without having to be completely in memory.

In particular, NiMo is a "safe" language, for two reasons: is strongly typed and portable. On the portability side "a safe language is completely defined by the programmer's manual"[7]. This paper intends to show that NiMo syntax (Part 1) and its graphical type system (Part 2) can be described in a very simple and concise way. NiMo execution and programming will be treated in separate papers.

2 Introduction

Many formalisms for the specification for concurrent and distributed systems have emerged. In particular considering boxes and strings approaches. Examples are action calculi, rewriting logic and graph rewriting, bigraphs [3]. The boxes and string metaphor is addressed with different levels of granularity.

One of the approaches is considering a process network as an hypergraph [6]. Based in this general framework, we encode NiMo nets as a class of hypergraphs that can be constructed using the construction operations.

In [5] a model for shared graphs is developed, and also a graph presentation of action calculi.

We define *Directed Hypergraph* as in [8], and extend the definition to *Annotated Hypergraph* to be used in describing the syntax of NiMo programs, which are a subclass of annotated hypergraphs. This class is

defined by giving the alphabet and the operations used to construct such programs. *Decorated Hypergraph* are defined for encoding NiMo programs with types. We give the operators to construct NiMo programs *hypergraph construction* and how an annotated hypergraph is transformed under hypergraph construction operations.

NiMo programs are process networks that not only reactive: A reactive system is a system that responds (reacts) to external events. In NiMo, input external nodes are the ones that accept messages from the environment. NiMo programs may have processes that react by themselves or due to their neighbors. Most of the modeling in the literature is addressed to reactive systems. NiMo processes, having the lazy ingredient, can react if there is a process that requires its computation, using a decoupled protocol. Its for this reason that NiMo programs do not fit exactly in [6] or [4]. The main difference is that in [6] modeling, a node can be shared by any two hyperedges without restrictions acting as the mean to communicate the two. NiMo programs have a structure close to Petri Nets, having channels instead of places for communicating between transitions (processes)[1]. But each channel has a single provider. This ingredient assures that non-determinism could be present in a net if non-deterministic processes are present. Another essential difference is that NiMo nets can be dynamic.

Here we treat NiMo syntax as process networks, with ports decorated with their type. Therefore we treat only editing operations on labelled hypergraphs and afterwards how this editing operation affect the node labeling.

We are not covering graph transformation (execution rules), which preserves type information on the transformed graph.[2]

2.1 Hypergraphs

Directed Hypergraph An hypergraph $H = (V, E)$ consists of a set of nodes V, a set of edges E, a connection mapping $s : E \to (\mathcal{P}(V), \mathcal{P}(V))$ and $\chi : \mathcal{H} \to \mathcal{P}(V)$ of external nodes.

There are two mappings $hd, tl : E \to V^*$ that give respectively the *head* and the *tail* of an edge.

$s(e) = (tl(e), hd(e))$

The hypergraph in Figure 1 has

$$V = \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12\},\ E = \{E_1, E_2, E_3, E_4, E_5\}$$

$$s(E_1) = (\{1, 2\}, \{4, 5, 6\})$$
$$s(E_2) = (\{3, 4\}, \{7, 8\})$$
$$s(E_3) = (\{6\}, \{9, 10\})$$
$$s(E_4) = (\{10, 11\}, \{12\})$$

[1] As Petri Nets, NiMo programs are bipartite graph, therefore processes do not communicate directly, but via a channel.

[2] This is true only free ports in definitions are not allowed

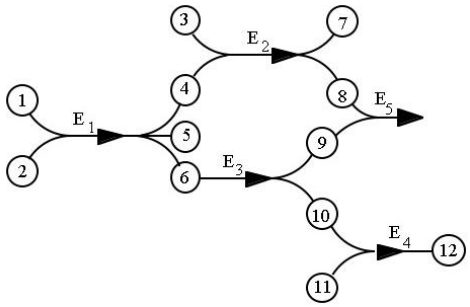

Figure 1: Example of an hypergraph

$$s(E_1) = (\{8, 9\}, \{\})$$

$$\chi(H) = \{1, 2, 3, 5, 7, 12\}$$

Forward edge or F-edge Is a directed hyperedge e, such that $|tl(e)| = 1$

In Figure 1 E_3 is the only F-edge.

Annotated Hypergraph An Annotated Hypergraph $AH = (H, L, \iota)$ where H is a directed hypergraph, L be a fixed set of labels and $\iota : E \to L$

Symetric image Given the hypergraph $H = (V, E)$, a symetric image is the hypergraph $H' = (V, E')$, where $E' = \{e' |\ \exists e \in E$ and $tl(e) = hd(e')$ and$hd(e) = tl(e')\}$

3 NiMo programs as Hypergraphs

The rest of the definitions will constrain the general definition given for annotated hypergraphs, to a the special class: NiMo programs.

In a NiMo program, vertices are called ports.

The set of vertices is the union of three disjoint sets:

V^{in}, V^{out}, V^F respectively in-ports, out-ports and f-ports.

For notation simplicity, instead of having sets as the range of s, hd, tl, and χ we will use sequences of ports, without repetitions.

$s : E \to ((V^{in})^*, (V^{out})^*, V^F)$ is a 3-tuple of strings of nodes and

$\chi : H \to (V^{out} \cup V^{in} \cup V^F)^*$ gives the external nodes of the hypergraph.

$hd, tl : E \to V^*$

According to this presentation, for each hyperedge the mappings s will return 3-tuples of sequences, each one taken from the three disjoint sets of ports.

Interfaces as hypergraphs The hypergraph $H = [l]_{(i,o,f)}$ having exactly one edge e with label l ($\iota(e) = l$). i is the sequence of in-ports, o is the sequence of out-ports and f is an f-port or is empty. $V = \chi(H))$ are the ports of an Interface. Is an hypergraph, with a single hyperedge and all ports are external and $hd(e) \in (V^{in})^*,\ tl(e) \in (V^{out})^* + + V^F$.

Irreducible graph Let G be a hypergraph. A (non-trivial) decomposition of G is a pair of inclusions A $\rightarrow$G $\leftarrow$ B such that G is the union of A and B and G $\neq$ A or G $\neq$ B. A hypergraph G is irreducible if it has no non-trivial decomposition.

Fact 3.1 *The only irreducible hyperqraphs are the sinqle vertex graph and the interfaces.*

Process and non process Interfaces The interface hypergraph

$$H = [l]_{(i,o,f)}$$

contains exactly one edge e, $s(e) = (i, o, f)$ and all the vertices are external ($\chi(H) = V_H$).

Non process interface have $f = \epsilon$.

Process interfaces have two roles: as functional data (using the out-port f to connect it to other processes. $f \in F$) or just a process (connected to other interfaces via ports in i and/or ports in o).

3.1 NiMo Programs

NiMo programs have two types of edges E_{ch} and E_I. The first ones connect out-ports with in-ports, while the second group connect in-ports to out-ports. The ports in the graph are the union of the ports in E_I.

E_{ch}: This group of edges is characterized by having a single source, which is an element of V^{out}, $tl(e) \in (V^{in})^+$. Given e such that $hd(e) \in (V^{in})^+$ and $tl(e) \in (V^{out})$ is the ordered pair formed by the sequence of in-ports and a single out-port. Therefore E_{ch} edges in NiMo programs are F-edges.

This class of edges are divided into two groups:

- *Simple edges* are edges having $s(e) = (i, o, \epsilon)$ $|i| = |o| = 1$ and a label taken from $\{red, white, green\}$[3]
- *Red Hyperedges* $s(e) = (i, o, \epsilon)$ $|o| = 1$ $|i| > 1$ and a label taken from $\{red, white, green\}$ [4]

$E_I = E_I^N \cup E_I^P$ are the edges corresponding to interface hypergraphs present in the toolbox. The elements in the toolbox correspond to the alphabet of he language[5]. Given e such that $tl(e)$ is the sequence obtained by clockwise enumeration of the in-ports as they are shown in the toolbox[6]. Similarly $hd(e)$ is the ordered pair formed by the sequence of out-ports in the west side of the interface, counterclockwise, and the f-port if present.

- *Interface non process* E_I^N $s(e) = (i, o, \epsilon)$ $|i| + |o| \geq 1$, the label taken from the set of types or a value in a given type
- *Process interface* E_I^P $s(e) = (i, o, f)$ $|i| + |o| \geq 1$ label taken from the set of pairs (name, mode)

[3] In the graph this type of edges are painted in black or blue

[4] In the graph this type of edges are painted in red

[5] This alphabet is dynamic as new processes can be defined and include in the toolbox. In this paper we do not cover the operations for adding new E_I^P to the toolbox

[6] In the working graph an interface can be rotated

In the next section we will give the operators to construct NiMo programs and will see that the hyperedges E_I^P correspond to processes, while E_I^N are other program elements. Simple edges and red hyperedges appear as a result of applying a constructor operators $\odot$. Simple edges relate pairs of elements in E_I. Therefore the constructed graph will be a bipartite graph[7].

4 NiMo Program Construction

In the toolbox, all the interfaces for building NiMo programs are present[8]. Some are processes that can be used as functional values. NiMo program are constructed starting from an empty hypergraph as working graph, and applying a sequence of operators taken from:

drag bringing an interface from the toolbox into the working graph. The toolbox holds the alphabet. Those are elements of E_I.

paint setting the opposite label to a simple edge or Red hyperedge E_{ch}. Red is opposite to white and white is opposite to red

$\odot$ connecting two nodes present in the graph by using a simple edge or a Red hyperedge

mode Set the label of a process interface E_I^P.

So, the graph, i.e. the NiMo program, is constructed by dragging interfaces from the toolbox, by adding simple edges or red edges (E_{ch}) using the operation $\odot$, by setting or unset the demand on an expression or setting the label to a process interface.

In NiMo the set L labelling the edges in the set E_I^P are pairs formed by interfaces names and the Modes of process. Modes are elements of the set: (*Disable*, *Demand Driven*, *End Driven*, *Data Driven*, *Weak Eager*, *Autoexpand*). The labels for edges in E_{ch} are $\{white, red, green\}$.

$$L = \{(process\ name, process\ mode)\} \cup \{white, red, green\}$$

A program under construction is an annotated hypergraph in the working space called the working graph.

In this sequence starting in the empty graph, and obtained by applying the operators every element is a retract of the precedent one[9].

Connections between nodes (one out-port and one in-port) add or change an edge of the working hypergraph i.e. alters the set E_{ch}, keeping the ports (under certain circumstances an F port may disappear). Dragging an interface adds a new hyperedge to the working graph and therefore new nodes are added to it (the interface ports). All the newly added nodes add up into the set of external nodes. Connecting two nodes, reduce the number of external nodes and eventually adds an edge to the

[7] As are Petri Nets

[8] Cannot be name clashes in the toolbox

[9] A retract of a graph H' is a subgraph H of H' such that there exists a homomorphism $r : H' \to H$, called retraction with $r(x) = x$ for any vertex x of H

working graph.[10]. $s(e)$ gives the in-ports, the out-ports and the F-port of the interface having the only edge e. For non-process interface, the F-port is not present. In a connection, the in-ports and the out-ports of an interface do not change, the F-port may vanish. A connection changes the χ function, because ports became bound.

In describing the construction operators, as graph transformers, we will use

$$H = (V, E, s, \iota, \chi)$$

for the hypergraph being transformed by the operator and

$$H' = (V', E', s', \iota', \chi')$$

for the resulting hypergraph. We use the expression $c \in \chi(H)$ to indicate that c is an external node. $c \in s(e)$ to indicate that $c \in hd(e) \vee c \in tl(e)$ i.e. we don't need distinguish if c is a in-port or an out-port. χ_e are the open ports of an interface having the only edge e.

4.0.1 $(H \xrightarrow{\mathbf{drag}(H''=(V'',\{e\},s'',\iota'',\chi''))} H')$

Rule 1 states that dragging an hyperedge from the toolbox, all its elements are added to the working graph.

$$\frac{H'' \in \text{toolbox}}{\begin{array}{c} V' = V \cup V'' \qquad s' = s \cup s''(e) \qquad E' = E \cup \{e\} \qquad \iota' = \iota \cup \iota''(e) \\ \chi^{'} = \chi \cup \chi^{''}_{e} \end{array}} \tag{1}$$

In Fig.2 an interface is dragged into the working space, taken from the toolbox[11]. The interface has one in-port, two out-ports and a f-port.
The following rules are for connecting an out-port b, which is not an f port, to an in-port a. $e, e_1 \in E_I$, $a \in s(e)$, $b \in s(e_1)$.

4.0.2 $(H \xrightarrow{\odot(\mathbf{a},\mathbf{b})} H')$

This operations connects an out-port (a) to an in-port (b). There are four cases: if a is an f-port (rules 4,5) or not (rules 2,3).

The first one, rule 2 is for constructing simple edges, while rule 3 for red edges.

$$\frac{\begin{array}{c} a, b \in \chi(H) \\ s(e) = (i, o, f) \qquad b \in i \qquad s(e_1) = (i_1, o_1, f_1) \qquad a \in o_1 \qquad a \notin F \end{array}}{\begin{array}{c} V' = V - \{f_1\} - \{\text{if } |i \cap \chi| = 1 \text{ then } \epsilon \text{ else } f\} \qquad E'_{ch} = E_{ch} \cup \{(a, b)\} \\ s'(a \odot b) = (a, (b, \epsilon)) \qquad s'(e) = (i, o, \text{if } |i \cap \chi| = 1 \text{ then } \epsilon \text{ else } f) \\ s'(e_1) = (i_1, o_1, \epsilon) \qquad \iota'((a, b)) = \text{ if } e_1 \in E^P_I \text{ then } \textit{white} \text{ else } \textit{green} \\ \chi^{'} = \chi - \{a, b\} \end{array}} \tag{2}$$

[10]In the case where the out-port was not an external node, the edge will be a red hyperedge. In any case if a interface involved in the connection had a f-port, it may disappears

[11]In the implementation, the f-port is not shown for interfaces in the toolbox

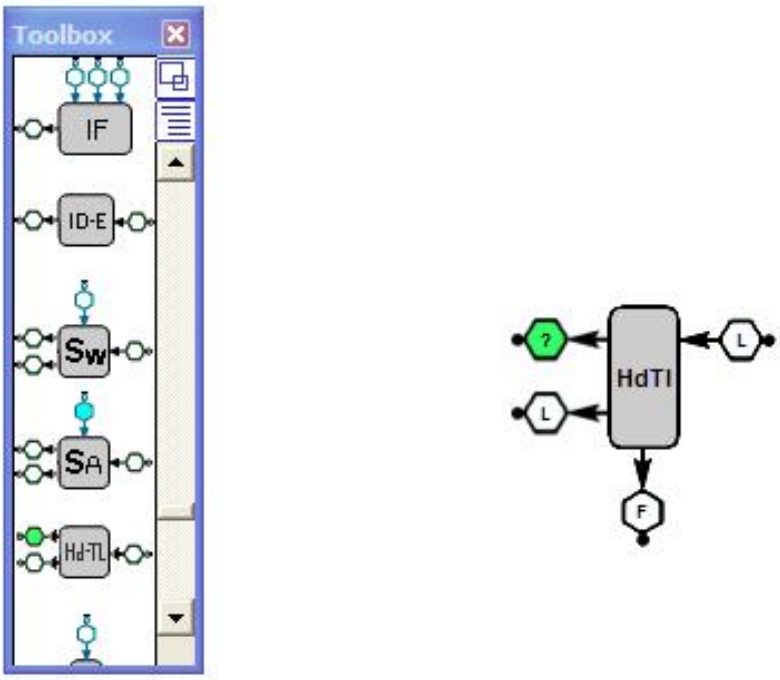

Figure 2: Drag an interface to the working space

On top right of Fig. 3, an edge is created between the two $HdTl$ interfaces. Both f ports vanish, because all the in-ports of the one of the left became bound and an out-port of the other interface is also bound. In the bottom right of Fig. 3, the connection leaves open in-ports in the interface $ifBool$, therefore the f port remains.

Rule 3 covers the case of connecting an out-port already connected.

$$\frac{\begin{array}{c} a \notin \chi(H) \\ a \notin F \qquad s(e) = (i, o, f) \qquad s(e_1) = (i_1, o_1, \epsilon) \qquad b \in i \qquad (a, c) \in E_{ch} \end{array}}{\begin{array}{c} V' = V - \{\text{if } |i \cap \chi| = 1 \text{ then } f \text{ else } \epsilon\} \\ E'_{ch} = E_{ch} - \{(a, c)\} \cup \{(a, bc)\} \; \} \qquad s'(e_1) = (i_1, o_1, \epsilon) \\ s'(a \odot b) = (a, (bc, \epsilon)) \qquad \iota'((a, b)) = \text{ if } e_1 \in E_I^P \text{ then } \mathit{white} \text{ else } \mathit{green} \\ \chi' = \chi - \{b\} \end{array} \tag{3}$$

By the $\odot$ operation a new edge $((a, b))$ is created. Both ports a, b are no longer external ports. F ports, if present, can be lost in this operation: The F port of the interface of the second operand, if this connection is done using the interface only remaining open in-port. Always the interface of the first operand (a) looses its F port (if present) when using a port in o_1.

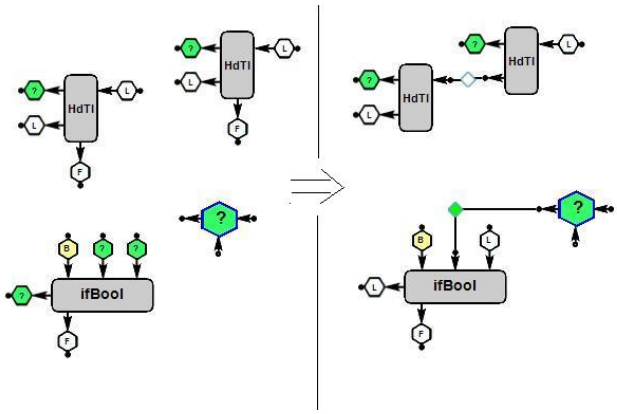

Figure 3: Two simple edges added to the hypergraph with different labels

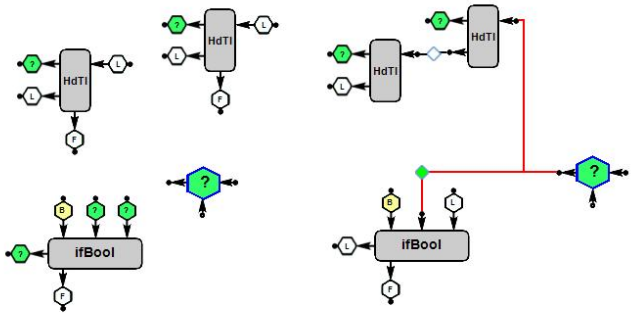

Figure 4: Sharing values

The next two rules, apply when the out-port in the operation is an F-port. The first rule (rule 4) is for adding a simple edge and rule 5 for connecting an F-port, the result not being a simple edge. These rules describe how a functional value becomes an input to a process.

$$\frac{\begin{array}{ccc} a \in \chi(H) & a \in V^{out} & a \in F \end{array}}{\begin{array}{ccc} V' = V & E'_{ch} = E_{ch} \cup \{(a,b)\} & s'(a \odot b) = (a,b,\epsilon) \\ & \iota'(a \odot b) = green & \chi' = \chi - \{a,b\} \end{array}} \quad (4)$$

$$\frac{\begin{array}{cccc} a \notin \chi(H) & a \in V^{out} & a \in F & s(e) = (i,a,\epsilon) \end{array}}{\begin{array}{ccc} V' = V & s(e'') = (bi,a,\epsilon) & E'_{ch} = E_{ch} - \{e\} \cup \{e''\} \\ & \iota'(a \odot b) = green & \chi' = \chi - \{b\} \end{array}} \quad (5)$$

4.0.3 $quad(H \xrightarrow{paint(e)} H')$

This operation changes the edge label. Green labels cannot be changed. Only red and white ones can be changed.

$$\frac{e \in V_{ch} \qquad \iota(e) \neq \ green \qquad \iota(e) = \ red \vee white}{\iota'(e) = \neg\iota(e)} \tag{6}$$

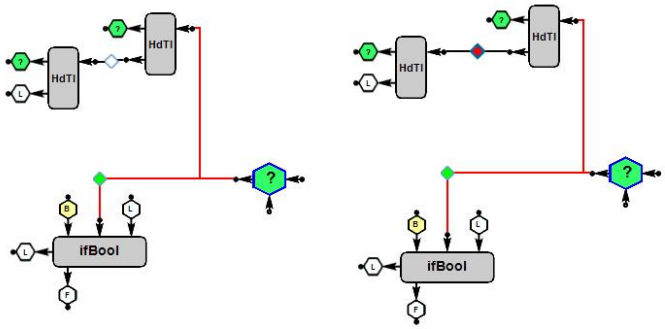

Figure 5: Changing labels: White → Red, Red → White

4.0.4 $quad(H \xrightarrow{mode(e,mod)} H')$

This operation sets the mode for process.

$$\frac{e \in E_I^P \qquad \iota(e) = (name, b)}{\iota'(e) = (name, mod)} \tag{7}$$

The *drag* operation adds a new edge to the working graph, an element from E_I.

The $\odot$ operator is defined only for pairs of nodes, one in-port and one out-port, the in-port must be external. This operation gives as a result a new edge in E_{ch}. The rules above cover the five possible cases.

The case where all the inputs are instantiated, is missing.

The *paint* operation, is a partial operation, changes the label of an edge which is not an interface edge.

The *mode* operation, is a total operation, changes the label of the interface edge.

As syntactic sugar, edges are painted in different colors. E_I edges are painted in different colors depending on they are basic processes (white), net processes (grey) or net processes that have been executed at least once(?). Also E_{ch} are red if they are incident to more than one in-port. Are blue if they are incident to a single port and is one on the top of the interface or are black if they are incident to a in-port on the side of a interface.

Lemma 4.1 *In a NiMo program a node has at most two edges, one is in E_I and one (if present) in E_{Ch}.*

Proof

The only way new nodes show up in a program is by means of a $drag(e)$ operation. The dragged hypergraph a siingle edge in E_I and all the nodes are in $\chi(H)$.

A node is removed from $\chi(H)$ by the operation $\odot$, which creates an edge $e' \in E_{ch}$.

If the node is in V^{out} and not in $\chi(H)$, the $\odot$ operation will create a new red edge, incident to the node, which is in E_{ch}.

Lemma 4.2 *NiMo programs are bipartite hypergraphs*

Proof *As edges go from in-ports to out-ports (E_I) or from out-ports to in-port (E_{ch}), NiMo programs are bipartite, directed hypergraphs.*

Corollary 4.3 *In a NiMo program there are no self loops*

Proof

Being a bipartite graph, there are no edge, e such that $hd(e) = tl(e)$

5 Graphic Syntax

As described in previuos sections, NiMo programs are modeled by hypergraphs. As being a graphical language the program must be drawn. Also, by being an interpreted language, then program execution must be displayed in any execution step.

A drawing of a graph is its representation on the plane. Formally, the drawing of a graph is a function which maps vertices of the graph to distinct points of the plane and edges to simple curves with ends in adjacent vertices.

5.1 Aesthetic Criteria

A good layout can be a picture worth a thousand words; a poor layout can confuse or mislead. Graphs are used to represent information and structure in various areas of the software engineering. To achieve the readability of the information presented, properties of a drawing are specified, the so called aesthetics criteria [1].

1. Minimization of edge crossings; Ideally, there is a planar drawing so that there is no edge crossing, but not every graph admits one. If there is no planar embedding the goal is to find a drawing with a minimal total number of crossings between edges.
2. Minimization of the drawing area; It is essential in practical visualization systems to save screen space. Furthermore, it is relevant if one cannot arbitrarily scale the graph down.
3. Minimization of the edge length; This criterion is divided into three similar minimization concepts:
    - Total edge length: Minimize the sum of the edge lengths.

- Maximum edge length: Minimize the maximum edge lengths.
- Uniform edge length: Minimize the variance of the edge lengths.
- Minimization of the bend number; This criterion contains three concepts likewise

the criterion for the edge length:

- Total bend number: Minimize the total number of bends along the edges.
- Maximum bend number: Minimize the maximum number of bends on an edge.
- Uniform bend number: Minimize the variance of the number of bends on an edge.

4. Minimization of the aspect ratio; Aspect ratio is defined as the ratio of the length of the longest side to the length of the shortest side of the smallest rectangle with horizontal and vertical side covering the drawing. Drawings with high aspect ratio may not be conveniently placed on a screen, even if it has small area.

Additionally to the aesthetics criteria mentioned, there is another important criterion, the so called user mental map. In an interactive visualisation system, changes to a drawing are made constantly, sometimes by the user and sometimes by the application. These changes frequently spoil the layout, since node overlaps might happen. A layout algorithm that rearranges the layout preserves the user mental map criterion if it destroys the mental map of the user as less as possible, by minimizing changes to the layout [9]. Most of the existing layout algorithms are designed for layout creation, and so is the Topology-shape metrics(TSM) approach. Such layout algorithms may completely rearrange the layout and thus destroy the mental map of the diagram.

Is crucial that the program visualization of remains understandable for the programmer during all the visualization faces. A measure of this understandability, is that the graphical presentation remains always close to the mental model the programmer has of the program. In achieving this goal, several presentations of the same program must be possible.

In the graph drawing literature the basic approach for drawing a graph works with three phases: the planarization, the orthogonalization, and the compaction. The planarization tries to minimize the number of edge crossing, the orthogonalization tries to draw the graph based on a grid and the third phase aims to shorten the length of the edges without loosing the properties of the graph. In an interactive visualization system, changes to a drawing are made constantly, sometimes by the user and sometimes by the application.

In our case graph drawing is subject to a set of constrains: some soft and some hard. One of the constrains is to preserve, whenever possible, the mental map the user has of the program. In the literature there are several work on the problem of drawing directed hypergraphs with port constraints[2]. Their goal is to draw the graphs, preserving the flow, having optimizing the number of cross edges. No matter their goal is different from ours, we will use their notion of hypergraph[12] and constrains.

[12] NiMo programs are a sub-family of their graphs

In Sect. 6.2, we present the treatment to the edge corssing problem, whilst in Sect. 6.1 the describe the constrains in drawings due to the port constraints. Next we give some definitions of concepts to be used.

6

A topological numbering of G is numbering of the vertices of G such that the numbering fulfills the condition $(u, v) \in E(G) \Rightarrow number(u) < number(v)$. A backward is an edge that goes from a lower numbered hyperedge to a bigger numbered hyperedge and forms a cycle in the graph.

A backward parameter, is a parameter connected with a backward simple edge.

External hyperedge Are those hyperedges that are able to produce results to the external world and the Expression interface

Productive hyperedge An hypergraph is a Productive Hypergraph if there is a path from it to an external hyperedge

6.1 Port Constraints

In this part we will used the NiMo program syntax given in Section 3.1.

The ports of a hyperedge e are the points in the drawing connected to the hyperedge by a small arrow (some autors call them dendrites). This points are visible whenever open. When a port is closed, the diamond hypergraph is drawn attached to the (end point). In NiMo these ports have a specific semantic interpretation, such as being inputs or outputs for data tokens in the program. As mentioned in [2] the strictest variant of port constraints is the one where the exact position of each port, relative to the respective node, is prescribed. This implies that each port has an associated side of the node where it is drawn, which is the case in NiMo programs, where hyperedges have a corresponding icon and the icon is present in the toolbox. Icons in the toolbox are used as reference, having ports on the north (top), south (bottom), west (left) and east (right). Due to the icon representation, nodes in the program can not be moved and are always in the icons border. This produces a rigidity for controlling edge crossing.

6.2 Edge crossing

Consider a topological numbering for the NiMo program, constrained to the lower values are assigned to external hyperedges. Ignoring back hyperedges, every NiMo program can be drawn from left to right guided by a topological ordering. This ordering may be need to be recalculated after each editing operation and execution operation because new nodes show up and maybe some disappears. Keeping the mental model stands for minimizing the variation in the drawing guided by two successive topological numberings of the program.

Back edges must be drawn, once the other edges are already displayed, in such way that the cycle keeps inside all the elements attached to hyperedges present in the cycle. This strategy, in the general case minimizes edge crossing.

7 Conclusions

In this paper we presented a syntactical aspect of the NiMo graphical language. First a formal definition is given for the graph that are NiMo programs. Several hypergraphs models serve for describing the syntax of NiMo programs. Different models emerge under the perspective of implementing the language. In this paper we present a single one.

Part II of this paper will define *Decorated Hypergraph* for encoding NiMo programs with types. We will present the way decorations in the graph evolve due to edition operations.

In the previous section we sketched some remarks signaling the difference between the problem of drawing a graph having some constrains like port constrains and drawing a sequence of graphs. In the literature the problem of drawing a graph is limited to the drawing of a single and final graph. In drawing NiMo programs the problem is incremental and in general is aided by the programmer during edition face.

The sequence of graph drawings correspond to the program evolution in edition and/or execution. This graph drawing must preserve the mental model the programmer has of the program being edited/executed and also allow the programmer to understand the program evolution. The degree attained of this objective is crucial to usability of the language as a debugging workbench.

References

[1] Giuseppe Di Battista, Peter Eades, Roberto Tamassia, and Ioannis G. Tollis. *Graph Drawing: Algorithms for the Visualization of Graphs*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1st edition, 1998.

[2] Markus Chimani, Carsten Gutwenger, Petra Mutzel, Miro Spönemann, and Hoi-Ming Wong. Crossing minimization and layouts of directed hypergraphs with port constraints. In *Proceedings of the 18th International Conference on Graph Drawing*, GD'10, pages 141–152, Berlin, Heidelberg, 2011. Springer-Verlag.

[3] Troels Christoffer Damgaard, Arne J. Glenstrup, Lars Birkedal, and Robin Milner. An inductive characterization of matching in binding bigraphs. *Formal Asp. Comput.*, 25(2):257–288, 2013.

[4] Andrei Dorman and Tobias Heindel. Structured operational semantics for graph rewriting. In Alexandra Silva, Simon Bliudze, Roberto Bruni, and Marco Carbone, editors, *ICE*, volume 59 of *EPTCS*, pages 37–51, 2011.

[5] Masahito Hasegawa. Models of sharing graphs: A categorical semantics of let and letrec. Technical report, 1997.

[6] Barbara König. A general framework for types in graph rewriting. *Acta Inf.*, 42(4):349–388, 2005.

[7] Benjamin C. Pierce. *Types and programming languages*. MIT Press, Cambridge, MA, USA, 2002.

[8] Alberto Torres and Julián Aráoz. Directed hypergraph models of knowledge bases in expert systems. *Acta Cientfica Venezolana*, pages 387–394, 1988.

[9] D. J. Walmsley. Mental Maps, Locus of Control, and Activity: A Study of Business Tourists in Coffs Harbour.