NiMoToons: a Totally Graphic Workbench for Program Tuning and Experimentation

Silvia Clerici [a,1], Cristina Zoltan [a,2] and
Guillermo Prestigiacomo [a,3]

[a] *Dept. Llenguatges i Sistemes Informàtics*
Universitat Politècnica de Catalunya
Barcelona, Spain

Abstract

NiMo (Nets In Motion) is a Graphic-Functional-Data Flow language designed to visualize algorithms and their execution in an understandable way. Programs are process networks that evolve showing the full state at each execution step. Processes are polymorphic, higher order and have multiple outputs. The language has a set of primitive processes well suited for stream programming and supports open programs and interactive debugging. The new version of the environment NiMo Toons includes: an also graphic and incremental type inference system, multiple output processes as higher order parameters, symbolic execution, five evaluation modes that can be globally or locally set for each process and dynamically changed, and facilities to measure the used resources (parallelism level, number of steps, number of processes, etc.)

Keywords: graphic language, functional, data flow, stream programming, parallelism, visual type inference, symbolic computation

1 Introduction

NiMo (Nets in Motion) [3,8] is a graphic programming language inspired on the Data flow representation of some lazy programs (first proposed by Turner [13]), where functions are viewed as processes and channels are (usually infinite) lists. Its main objective is to provide the user a full control over its application development, debugging and optimization. The fact of being totally graphic is the key point because all the execution internals can be visible. The net is the code but also the computation state. Edition and execution are interleaved, and any partially defined net is an open program that can be executed and modified step by step. All together allow incremental development even during execution, because the initial code can evolve and be stored at any step as a program that can be recovered later. On the other hand, execution steps can be undone, acting as an on line tracer and

[1] Email: silvia@lsi.upc.edu
[2] Email: zoltan@lsi.upc.edu
[3] Email: gpresti@gpresti.com

debugger. The NiMo graphic syntax is simple and concise. Solutions of growing complexity can be built using a small set of graphic primitives, which allow dealing with higher order, partial application, different evaluation policies, and polymorphic type inference. The net architecture shows the chains of function compositions and exhibits the implicit parallelism. Back arrows give an insight of the "recurrence laws", i.e. how new results are obtained from the already calculated ones. This bi-dimensional view of the algorithm facilitates the reasoning about it and its possible improvements.

The full semantics of NiMo was defined by means of graph grammars and implemented in a graph transformation system [12]. This first prototype (NiMoAGG) was the basis for the development of NiMoToons, the NiMo graphic environment. The current version presented here, is a substantial development of earlier work [6]. The system overall design and implementation issues were described in that paper. Here we focus on the main newly added features and their application possibilities. The novelties are:

- A static and incremental graphic type inference system that guarantees type safeness by construction and allows identifying easily the origin of error messages. It is described in section 3.
- A more flexible use of multiple output processes as higher order parameters, and processes with configurable arity.
- Symbolic execution that allows computing with symbolic constants of any type including polymorphic ones.
- Visualization features like interactive separation of shared expressions, non expanded net-processes execution.
- Measures of program behavior for comparing versions in terms of time, space and parallelism.
- A customizable evaluation policy with five modes that can be global or locally set for each process and dynamically changed, covered in section 4.

This last feature gives a very flexible way to experiment different strategies to exploit implicit parallelism, allows subnet synchronization and, together with symbolic execution, provides the means for generative and multistage-programming as it is discussed in section 4.2.

2 NiMo Overview

2.1 Graphic Syntax

Figure 1 illustrates the main graphic elements. Processes are represented by *rectangles* with two kinds of parameters, horizontal entries for flowing data (streams), and vertical entries for parameters that are not channels in the data flow sense. Their names are shown in different colors according to their evaluation modes (see section 4). Data are represented by *hexagons*, which are placeholders that can be bound to a value. If not, they can be considered as free anonymous variables. Hexagons are colored and labeled according to their type, polymorphic data are green "?". *Black dots* are duplicators for multiple uses of a value or channel and *circles* repre-

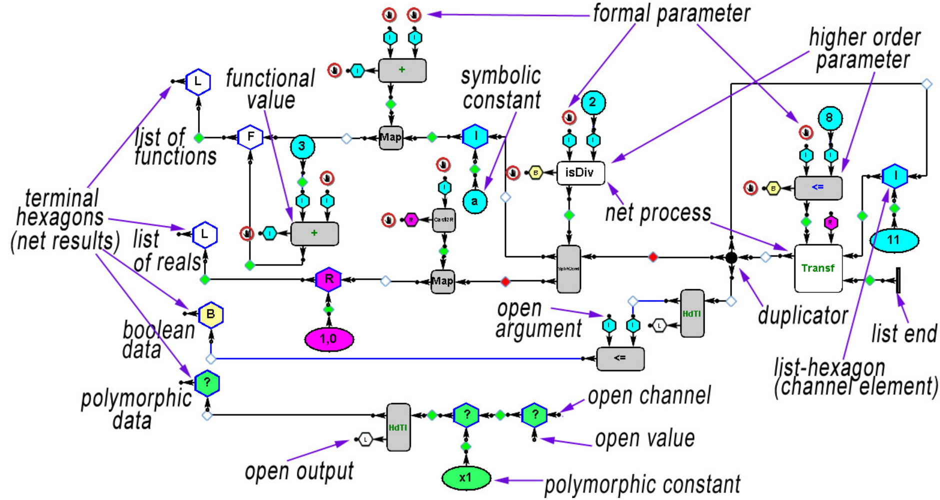

Fig. 1. A NiMo program example

sent constant values (also colored according to their type). In figure 1, the net has four outputs (the leftmost hexagons): a list of *functions*, a list of *reals*, a *boolean* value and a *polymorphic* value. All the mentioned nodes are interfaces having typed (in/out) connection ports. Interfaces are dragged from a toolbox (see left side of figure 2) and dropped into the workspace where the new net is being built.

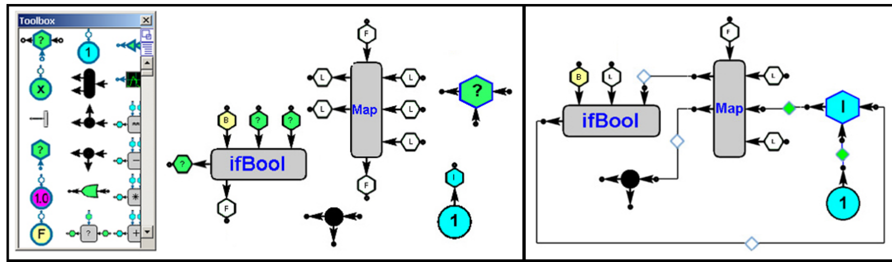

Fig. 2. Interfaces

Afterwards they are connected by clicking on the respective in and out ports, if both types are compatible. In this case an *edge* is constructed or else an error message is generated, and therefore nets are type safe by construction (see section 3). The edges have a state reflecting data evaluation degree or process activation. It is shown by means of a colored *diamond*. When a process output is connected the diamond is white, and incoming data items are connected with green diamonds as shown in the right side of figure 2. The user can change to red the white diamond of a process output to request the process to act (or it can be changed in execution as is discussed in section 4).

2.2 Processes are more than functions

Multiple output processes. Processes as functional data: In NiMo, processes are functional in the sense of referential transparency and can be also a valid data. However according to the data flow paradigm they can have no entries, multiple outputs, and even none (think of a sink-process that consumes its inputs) but the number of inputs and outputs cannot be both zero. Let's observe that on the

left of figure 2, process interfaces have an out port at the bottom. It is not one of their outputs, but their value as a functional data. This out port disappears whenever one of the outputs or all the inputs of the process are connected (it is acting as a process, not as a functional value). Higher-order parameter processes are connected by this port as can be seen in figure 1. When this port is connected, all the open ports of the process interface become blocked (red circle) to prevent that new connections can be made. Otherwise its value as a functional constant (and of course its type) would be different. See for instance the first element of the first output list in figure 1, which is the NiMo equivalent for the function (3+) in Haskell notation.

Curried/uncurried Implicit Behavior: Multiple inputs of a process in NiMo can be interpreted in curried or uncurried way depending on the context. This is a kind of implicit conversion from one functional type to other type. On the other hand, partial application can be made in any order. In the new version the effective arguments for the application can also be delayed. In figure 3 the first higher order parameter *ifBool* has its first and third parameters already applied, and therefore it has one single parameter. The green arrow at the first place indicates that the argument value will be completed later.

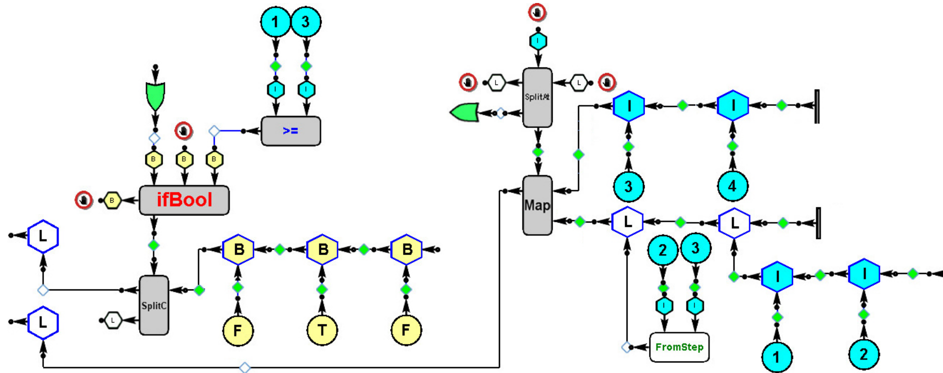

Fig. 3. Partial application and resulting

Partial Resulting: The toolbox provides a repertory of basic processes (gray rectangles) including multiple output versions of many Haskell prelude functions. For instance the process *SplitAt* is analogous to the *splitAt* function returning a pair of lists, but it can behave also as *take* or as *drop* just by leaving one or the other output disconnected (three processes for the price of one). We will refer to this multiple behavior as *partial resulting*, in analogy with the notion of partial application i.e. there is a symmetry in parameters and results regarding partiality. In the first version only single output processes were admissible as higher order parameters. Now multiple outputs and also partial resulting are allowed. In figure 3, the higher order parameter of *Map* is the process *SplitAt* acting as a *take*; to leave its second output open, it must be explicitly indicated by means of a green arrow before connecting it to *Map*.

Net Processes: Are user defined components, their interfaces (the white rectangles) are defined by means of a net that has to be parameterized and given a name. The user binds the in/out ports of a configurable interface with the open in/out ports of the net to be considered the formal parameters and results. This

mechanism is the graphic equivalent to bound variables definition in a lambda abstraction.

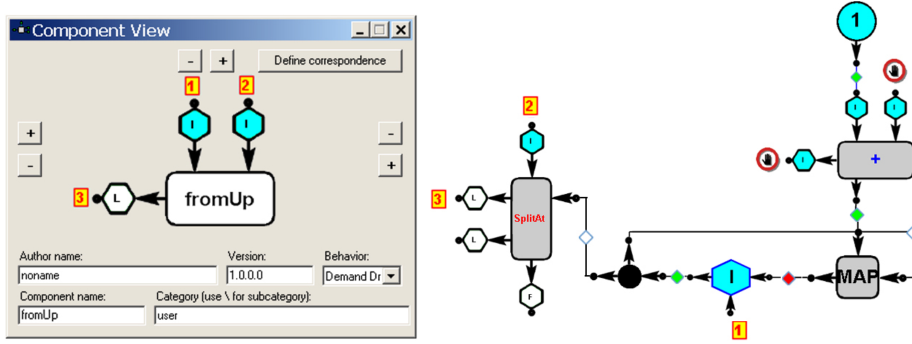

Fig. 4. Net Process definition

Figure 4 shows an example for the process *fromUp* that generates a list with k consecutive integers from the value n, where n and k correspond respectively to the parameters labelled 1 and 2. The equivalent Haskell code for this net is

fromUp n k = x where (x, y) = splitAt k z ; z= n: map (1+) z

Afterward it can be imported to the toolbox for being used in a new net and so on, allowing incremental net complexity up to any arbitrary degree. In execution, when the net process has to act the interface is replaced by the net setting the connections according to the corresponding bindings. Though some of the in/out ports could have been left open, NiMo allows to make "parameter passing" anyway. In terms of graph transformation this replacement mechanism implements what we call the *expansion rule* of the process.

Configurable arity processes and library nets: The system provides several basic processes with configurable arity, as a *Map* with any number of inputs and output channels, *Takewhile* and *Filter* with the same number of inputs and outputs, and an *Apply* process. Also, for top down development a new interface (with the desired name, in/out ports and also their types) can be created bringing the *generic process* interface from the toolbox. Its net definition can be postponed.

In addition, some useful parameterized nets as *fromStep* are provided in the system library. Given that the net size is critical for visualization, some of them are considered to be included in the basic processes repertory.

2.3 Symbolic execution

Along with the ability to run with free variables (open in ports) and the postponement of the process definition, in NiMo it is also possible to use symbolic constants of any type, even polymorphic. Symbolic manipulation is useful to compare equivalent codes in a more abstract way, and it can be used as a proof assistant. For instance in figure 5 it can be seen the initial results of evaluating $map(f \cdot g)\ [x1, x2, x3, x4, x5]$ and $map\ f(map\ g[x1, x2, x3, x4, x5])$. In this case the net definition of f and g interfaces is not really postponed; they are being used as symbolic functional values. The user can prevent a process to be expanded by assigning it a disable mode (see section 4). Figure 6 shows the execution of the classical Fibonacci net example, where the initial values 0 and 1 are replaced by a pair of symbolic integer constants

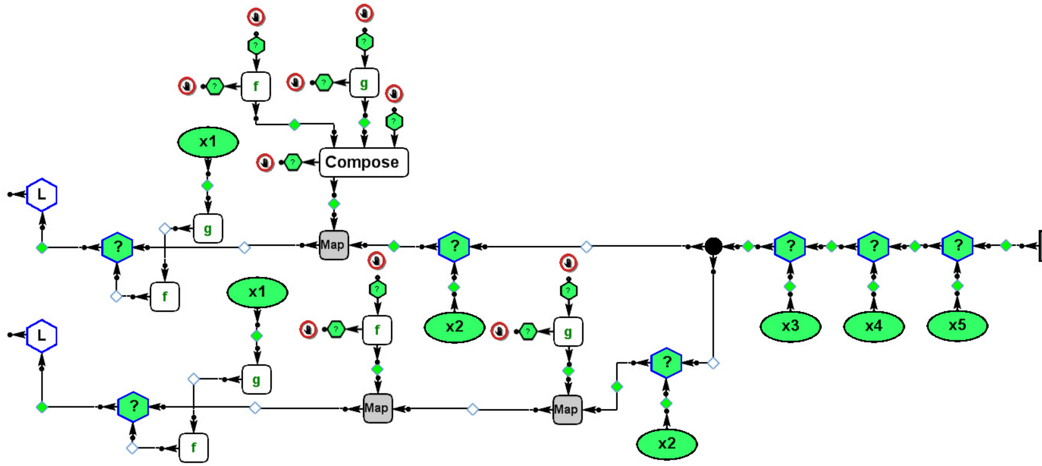

Fig. 5. Symbolic execution with polymorphic values

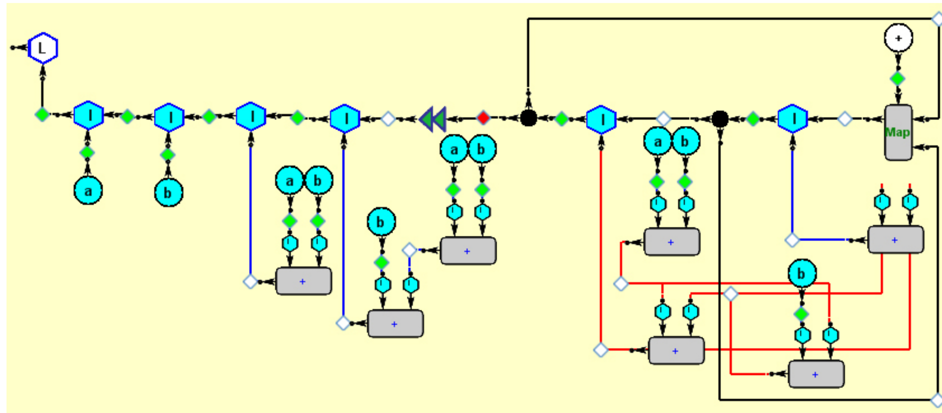

Fig. 6. Fibonacci's symbolic execution

a and b. The red multi-edges correspond to the shared expressions produced by the duplicators. In NiMo only constant values (circles and list-ends) are physically duplicated; any other subnet is shared. However, since sharing can obscure the understanding of the results, duplication can also be enforced. It can be performed locally on an in port, or by preceding the output of a subnet having shared expressions with a special purpose process: the double-green-triangle that can be seen on the left of the duplicator. In the previous version, where the evaluation policy was lazy, it was used in the net outputs to set a continued demand on its provider. Now it also has this mentioned functionality for enforcing duplication of shared values. This algorithm is not trivial due to the existence of multiple outputs processes.

3 Visual type inference

In NiMo type inference is static and incremental. During the net construction each connection is type consistent. The type inference system is a graphic generalization of the classical Hindley/Milner algorithm to deal with multiple outputs and curried/uncurried conversion. The net has an associated *type graph*. It is incrementally built during the net construction and its evolution is optionally visible.

Each process in/out port is tied to a node in the type graph. This feature allows identifying easily the origin of the type error messages. The net type graph is constructed starting from the *type descriptor* of each interface. In Figure 7 we can see the interfaces on the left of figure 2 with their type descriptors and the resulting type graph after connecting them. Let's observe that the type of the out port at the bottom of *ifBool* and *Map* interfaces, describes their types. In NiMo a process type is a graph rooted with a hexagon F whose outgoing edges are labeled *From* and *To*. To describe textually this type we use the $\|$ notation to signal that we are neither using the curried nor the uncurried type as it was discussed in 2.

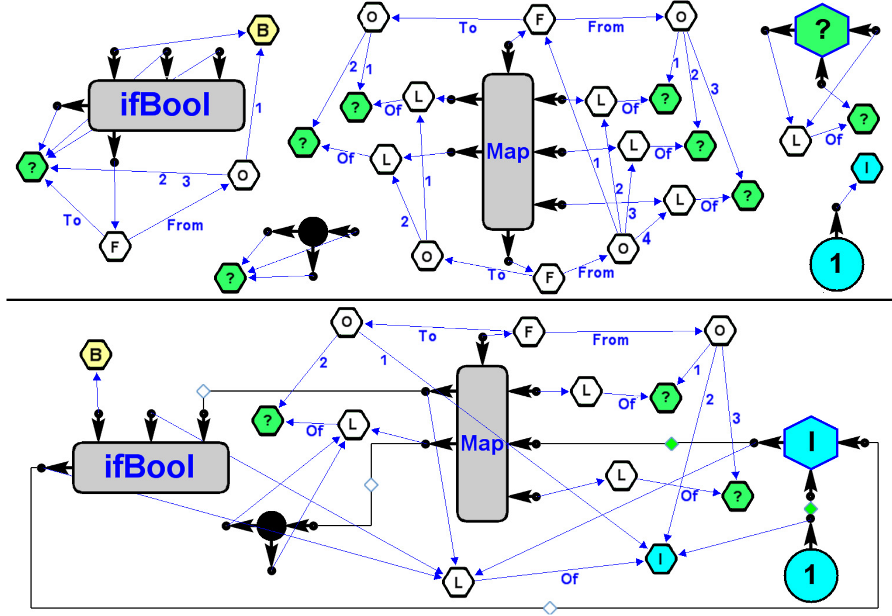

Fig. 7. Type descriptors

It denotes the type constructor for ordered parallel inputs or results corresponding to graphs with an O-hexagon root. The textual denotation for the *ifBool* interface type is $bool\|a\|a \to a$. In this case, the hexagon labeled O is the root of the graph corresponding to $bool\|a\|a$. It should have as many outgoing edges as the input parameters, but multiple occurrence of the same type variable a is represented by a single polymorphic hexagon and a single edge with two labels (2 and 3). The *Map* type is $((a\|b\|c \to d\|e) \parallel [a] \parallel [b] \parallel [c]) \to [d] \parallel [e]$. After connecting the interfaces, the type graph is the final result of unifying each pair of in/out port types. Intuitively each connection produces the "fusion" of both port types when the corresponding hexagons are superposed (if the types are compatible), i.e. the minimal graph that has the same nodes and edges but eventually a *?* node in one of them has been replaced by the more refined type graph of the other. This structural unification of graphs is enough to graphically model type inference in functional languages. But the existence of multiple outputs and the curried/uncurried casting required a particular procedure for unifying process types. It is detailed in [4].

In figure 8 the higher order parameter has a curried interpretation. Let's observe that before connecting process +, its type is $int\|int \to int$ and the parameter of *Map* is $int \to a$. After connecting them the unified type is $int \to (int \to int)$.

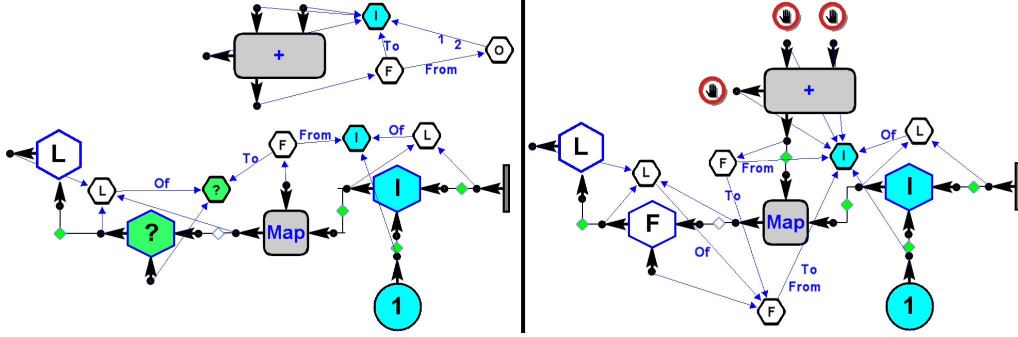

Fig. 8. Process type unification

4 Evaluation Modes

Most languages admitting parallel computation, have specialized constructs to enforce o suggest parallelization. In NiMo parallelism is implicit, in the model all processes selected to act (the analogous to *selected redexes*) are supposed to execute in parallel at the same execution step. Every selected process is explicitly marked by means of a red frame around its interface. An *execution step* is a transition from one net to the next where all the marked processes have produced the corresponding graph transformation. From the user point of view they all have acted in parallel. In the previous version NiMo followed a parallel lazy policy, all processes acted only under demand (only when some of their outputs had a red diamond), except a distinguished one that continuously forced its provider to act. A required process without enough data to act required the needed inputs providers and so on. The user could only change the evaluation order setting a demand on a given process (by changing to red some of its output diamonds). Now the user control on the selection criterion and therefore on the process execution scheduling is substantially upgraded. Each process has its own evaluation mode that can be set globally (for all) or locally for each process and can also be changed during execution.

The modes for basic process are: *Disable*: the process is not able to execute (even if it is requested). *Demand-Driven*: The process is able to execute only if it is requested. *End-Driven*: The process is able to execute as soon as it can end and disappear (for instance a non-required map can execute whenever any of its input channels ends). *Data-Driven*: The process is able to execute as soon as it has enough data. *Weak-Eager*: the process is always able to execute or to request its needed input providers.

Net processes have three possible modes: *Disable*: will never expand; *Demand-Driven*: only when requested; *Auto-Expand*: always applies its expansion rule.

4.1 The evaluation model

In [1], a labeling procedure is used for selecting threads to be active in each synchronized execution step. The labels are: Inactive, Blocked, Runnable and Active. We do not have an equivalent for Runnable because in the NiMo model the number of processors is unbounded, however exists a similarity with the underlying concepts of the other three ones, but in our case labels relate to processes, not to threads. A

basic process having all its needed inputs evaluated enough is Active if: its mode is Data-driven or Weak-Eager, or its mode is End-driven and can terminate, or its mode is Demand-driven and is requested.

A basic process with any of its needed inputs not enough evaluated is Blocked if it is required or its mode is Weak-Eager. All the other basic processes are Inactive.

Net processes are never Blocked. A net process is Active if its mode is Demand-Driven and it is requested, or if its mode is Auto-Expand. Otherwise it is Inactive.

The NiMo execution scheme for each execution step is the following:

1 All the Active processes that were marked with a red frame are executed.

2 Disconnected subnets are erased by the garbage collector rules.

3 Labeling of processes and marking actions closure: Active processes are marked with a red frame. Blocked processes require their needed input providers, each new requested process becomes Blocked or Active and so on until no new Blocked or Active processes are found.

Execution ends when no Active processes are found. Initially step 1 has an empty set of marked processes.

The graph operational semantics for NiMo is detailed in [5].

4.2 Customizing Evaluation

Different semantic models could be implemented in NiMo using modes. When only the outermost processes (the ones nearest to the net outputs) are set to Weak-Eager, and all the other processes are set to demand-driven the net has a total lazy parallel behavior. Setting non outermost processes as End-Driven allows net simplification. Setting all processes as Data-Driven gives the usual semantics in data flow approach. An eager semantics cannot be emulated in NiMo by changing modes, because Weak-Eager processes require only their needed inputs. Combining modes allows increasing the implicit parallelism, dealing with synchronization and also regulating channel population. Also the evaluation modes could be used for deactivating subnets during experimentation or promoting speculative calculations, and to prevent evaluation of symbolic values. Modes alter the process scheduling without changing the code (only the color of process names changes).

The system provides tools for measuring the execution behaviour of a program. A step counter and two statistical viewers give the figures of resource consumption. The first viewer shows the number of processes and data items per step, which correspond to the total memory usage. The second one shows graphically the number of processes acting at each step, and also the time the step takes (in the current NiMo implementation). We have observed that starting with a lazy policy, a few strategic changes in some processes modes result in a substantial improvement in the program performance. For example, the net in figure 9 calculates the prefixes of its input channel. The equivalent Haskell code is

prefixes x = y where y = [ ]: zipWith (++) y (map (: [ ]) x)

Starting with all the processes being Demand-Driven, except the duplicator that is Weak-Eager, execution takes twenty-five steps. The result is shown in figure 10. It can be seen that the applications of both HO parameters ++ and (:) remain unevaluated and their results are shared.

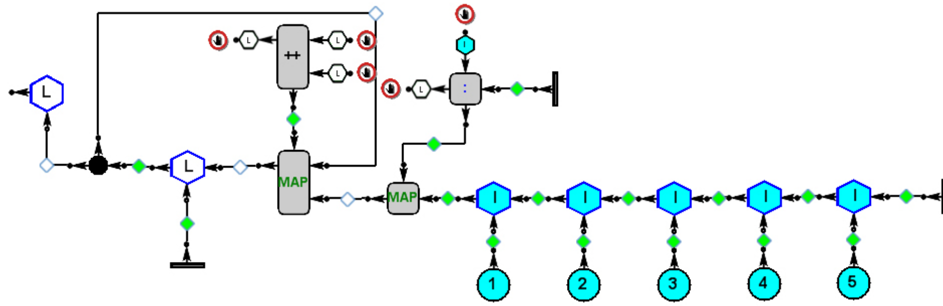

Fig. 9. Net prefixes

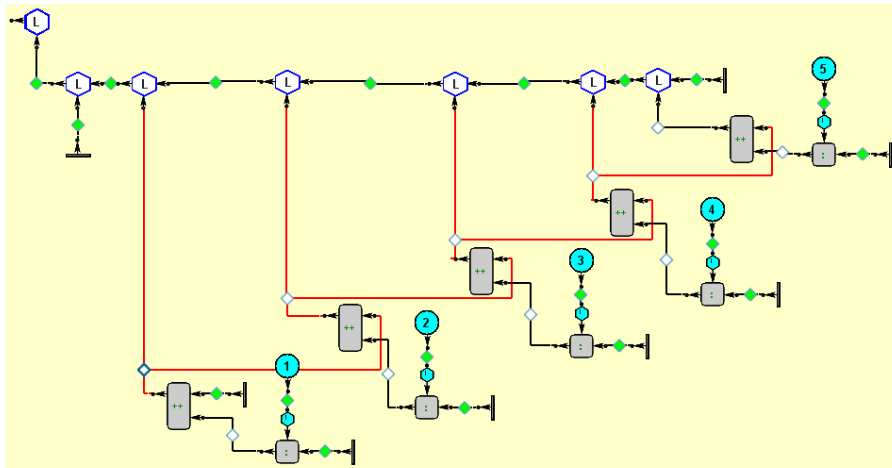

Fig. 10. Prefix execution with shared results

If now we set all of them as Data-Driven (using the command for globally setting a mode) the execution can continue, and after nine steps the net on the left of figure 11 is obtained.

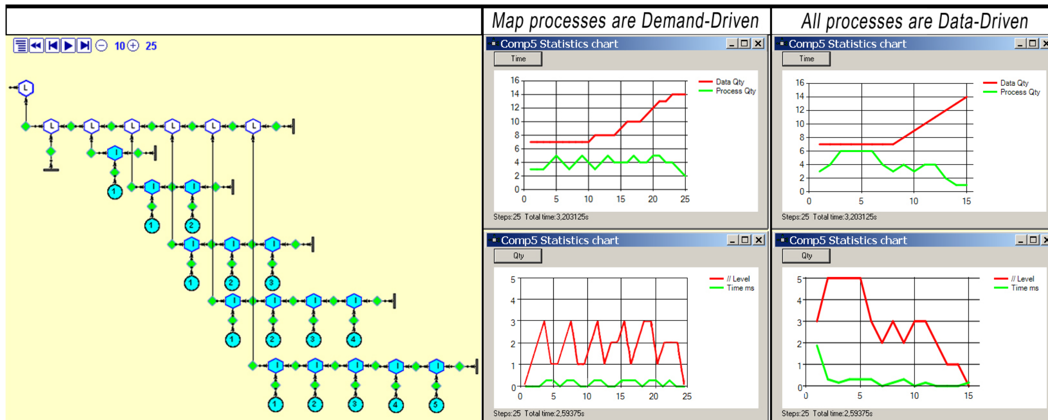

Fig. 11. Prefix reults and statistical viewers

This final result could have been directly obtained in twenty-five steps if ++ and (:) had been set to this mode from the beginning. And it would have taken only fourteen steps if we had started the execution with all the processes set to Data-Driven. On the right of figure 11 we can see the statistical viewers for both cases.

When the Map processes are Demmand-Driven, the red line in the bottom viewer shows that at most three processes execute in parallel, thus the mean of processor usage is less than two. But setting all the processes as Data-Driven the maximun is five and the average goes up beyond three, getting a 40 % gain in processing time, while keeping a similar usage of memory as the top viewers show.

On the other hand, a very interesting consequence of having disabled processes is that generative programming becomes very natural. A net can be defined to generate another one containing disabled processes. The net evolves until no more processes can act; this final result is the desired program. To be executed it only requires to globally set the disabled processes to another mode, and execution proceeds in the next step, or it can be stored to be run later. This technique was used (in collaboration with the UPC team of the WISEBED project [14]) to generate different topologies for sensor networks of variable size and afterward simulating their behavior [2].

5 Related works and final remarks

There are several languages or tools sharing some common characteristics with NiMo. In [3] we related some tracers, debuggers, and visual representation tools for functional languages. GemCut [9] is a graphical viewer for functions in the Haskell like language CAL, the editor uses the inference system of the CAL compiler to prevent type errors. In NiMo the type inference is also made graphically and gives the user on line visibility of the type inference process. TypeTool [10] and System I [11] are web-based tools for visualizing type inference of lambda terms, they are oriented to teaching the basis of type inference algorithms for functional languages.

Regarding the different evaluation modes we can mention Ptolomy [7]. It is a visual language based on actors having ports as communicating interfaces. Actors have parameters that are not visualized on the actor but shown in a separate windows. In NiMo all the program state is visualized. Ptolomy deals with continuous data, while NiMo is only discrete. In Ptolomy there is a variety of domains with an uniform evaluation policy, for instance process network domain correspond to a Data-Driven policy.

However, beyond partial similarities, to the best of our knowledge there is no other work that can be globally comparable, neither in approach nor in the integration of all the mentioned features in a single graphic system.

The graphic-functional-dataflow characteristics of NiMo result in a very powerful computation model. The mixed paradigm opens several possibilities not yet explored in pure dataflow or functional approaches. Integrating both worlds has supposed a big challenge and it has been necessary to find many creative solutions for making both models compatible. In particular, dealing with multiple outputs and curried/uncurried compatibility has required a non-trivial generalization of the usual notions of polymorphic type inference to handle the process type. On the other hand, the fact of being graphic and the underlying graph-transformation operational semantics, give the user a full control on the execution state at any step in a way impossible to imagine in textual languages. But at the same time total visibility (nothing can be hidden under the carpet) also required to find a graphic

notation for many "internals" to make them understandable. Besides, the visualization aspects are critical when nets grow, and it is a really difficult problem to solve that textual languages do not have to face.

We are currently working on several aspects of net visualization like improving the non expanded view of net process and channel viewers.

At the same time, a distribution version of NiMo Toons including an interactive tutorial is now in preparation. In this way we hope to extend the range of possible users beyond the academic environment.

Acknowledgement

We want to thank David A. Turner and Ricardo Peña for useful discussions while visiting us.

References

[1] Baker-Finch, C., D. J. King and P. Trinder, *An operational semantics for parallel lazy evaluation*, in: *ICFP '00: Proceedings of the fifth ACM SIGPLAN international conference on Functional programming* (2000), pp. 162–173.

[2] Clerici, S., A. Duch and C. Zoltan, *Implementing static synchronus sensor fields using NiMo*, Technical Report LSI-09-13-R, Dept. Llenguatges i Sistemes Informàtics, Universitat Politècnica de Catalunya (2009).
URL http://www.lsi.upc.edu/dept/techreps/llistat_detallat.php?id=1052

[3] Clerici, S. and C. Zoltan, *A graphic functional-dataflow language*, in: H.-W. Loidl, editor, *Trends in Functional Programming*, Trends in Functional Programming **5** (2004), pp. 129–144.

[4] Clerici, S. and C. Zoltan, *Graphical type inference. a graph grammar definition*, Technical Report LSI-07-24-R, Dept. Llenguatges i Sistemes Informàtics, Universitat Politècnica de Catalunya (2007).
URL http://www.lsi.upc.edu/dept/techreps/llistat_detallat.php?id=970

[5] Clerici, S. and C. Zoltan, *A dynamically customizable process-centered evaluation model*, in: *PPDP '09: Proceedings of the 11th international ACM SIGPLAN conference on Principles and practice of declarative programming* (2009), pp. 37–47.

[6] Clerici, S., C. Zoltan, G. Prestigiacomo and J. G. Sanjulián, *Diseño de un entorno integrado de desarrollo para NiMo*, in: *VI Jornadas de programación y lenguajes-Prole 2006*, 2006, pp. 233–242.

[7] II, J. D., C. Hylands, B. Kienhuis, E. A. Lee, J. Liu, X. Liu, L. Muliadi, S. Neuendorffer, J. Tsay, B. Vogel and Y. Xiong, *Heterogeneous concurrent modeling and design in java*, Technical Report Technical Memorandum UCB/ERL M01/12, Electronics Research Laboratory, Dept of EECS, University of California at Berkeley (2001).
URL http://ptolemy.eecs.berkeley.edu/

[8] NiMo-Home page (2009).
URL http://www.lsi.upc.edu/~nimo/Project

[9] Resources (2009).
URL http://resources.businessobjects.com/labs/cal/gemcutter-techpaper.pdf

[10] Simões, H. and M. Florido, *TypeTool - a type inference visualization tool*, in: *In Proceedings of the 13th International Workshop on Functional and (Constraint) Logic Programming*, 2004.

[11] System I (2009).
URL http://types.bu.edu/modular/compositional/system-i/

[12] Taentzer, G., *Agg: A graph transformation environment for modeling and validation of software*, in: *Applications of Graph Transformations with Industrial Relevance* (2004), pp. 446–453.

[13] Turner, D. A., *Miranda: a non-strict functional language with polymorphic types*, in: *Proc. of a conference on Functional programming languages and computer architecture* (1985), pp. 1–16.

[14] WISEBED (2009).
URL http://www.wisebed.eu/