

A Dynamically Customizable Process-Centered Evaluation Model

Silvia Clerici Cristina Zoltan *

Departament de Llenguatges i Sistemes Informàtics
Universitat Politècnica de Catalunya
Barcelona, Spain
{silvia,zoltan}@lsi.upc.edu

Abstract

We present the graph operational semantics approach used for defining NiMo nets execution, which mix lazy, data-driven and a weak form of eager evaluation, all in parallel. NiMo is a totally graphic language from the family of higher order typed languages but with a strong data-flow inspiration. Programs are process networks that evolve showing the full state at each execution step and can be dynamically changed or completed. In NiMo parallelization is implicit. The conjunction of two kinds of tags determine which processes are able to act in the same execution step. According to their tag, processes can behave in five modes that can be globally or locally set for each one, and can be also dynamically changed. Combining modes gives a very flexible way to experiment different strategies to increase processor usage, decrease channel population, and achieve subnet synchronization. Together with symbolic execution it also provides the means for generative and multi-stage programming.

Categories and Subject Descriptors D.3.3 [Programming Languages]: Language Classifications—Data-flow languages, Functional languages, Multiparadigm.
D.3.1 Programming Languages, Formal Definitions and Theory, [Semantics]

General Terms Languages

Keywords Computation model, functional languages

1. Introduction

NiMo (Nets In Motion) is a graphic-functional-data flow language designed to visualize algorithms and their execution in an understandable way. NiMo is not a visual interpreter for a textual language because there is no textual code at all. It is a true graphical programming language. 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

* Partially supported by the ICT Program of the European Union under contract number 215270 (FRONTS).

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

PPDP'09, September 7–9, 2009, Coimbra, Portugal.

Copyright © 2009 ACM 978-1-60558-568-0/09/09...\$10.00

a set of primitive processes well suited for stream programming and supports open programs and interactive debugging. The system provides an also graphic and incremental type inference system that guarantees type-safeness by construction. There are also facilities to measure the used resources (parallelism level, number of steps, number of processes, etc.).

NiMo nets are graphs that the user builds with a specialized graph editor whose kernel is a graphical type inference system. The interpreter is a specialized graph transformation system, and therefore the NiMo operational semantics is given in terms of graph transformations. The full semantics of the NiMo initial version [Clerici and Zoltan 2004] was defined by means of attributed graph grammars and implemented in a graph transformation system [AGG Home page]. This first prototype (NiMoAGG) followed a parallel lazy evaluation policy; nevertheless, the user could modify locally the evaluation order by setting an explicit requirement on any process.

In the current version the user control on the process action scheduling is substantially greater and easier to set. There are five evaluation modes that can be globally or locally set for each process and dynamically changed. The process mode determines the activeness level of the process according to which the process behavior can range from “absolute passivity” up to “being always trying to act”. The action rule of each process determines the context conditions under which the process *may* act, and produces the corresponding transformation when the process must act. The context conditions are recognized by looking at the process input/output edge’s tags, and together with the process mode determine whether or not it is the case. This process-centered approach gives the user a very intuitive and notably flexible way for customizing evaluation order.

The overall approach is outlined in section 3 and its advantages and possible applications are discussed in section 3.2. In section 4 the evaluation model of NiMo and the elements to define its graph operational semantics are presented. We focus on the scheduling to handle the different modes in which the processes can act. Since the textual language nearest to NiMo is Haskell, and in particular GpH, the description of the scheduling is related to the semantics given in [Baker-Finch et al. 2000] and the differences between both approaches are discussed. Section 5 introduces the main concepts on attributed graph grammars and graph transformation, and then presents the set of attributed graph grammars used to define the operational semantics of an execution step. In section 6 other languages and approaches are also discussed.

In the next section we give a brief description of the main NiMo elements and constructs, and then we summarize its most significant differences with Haskell.

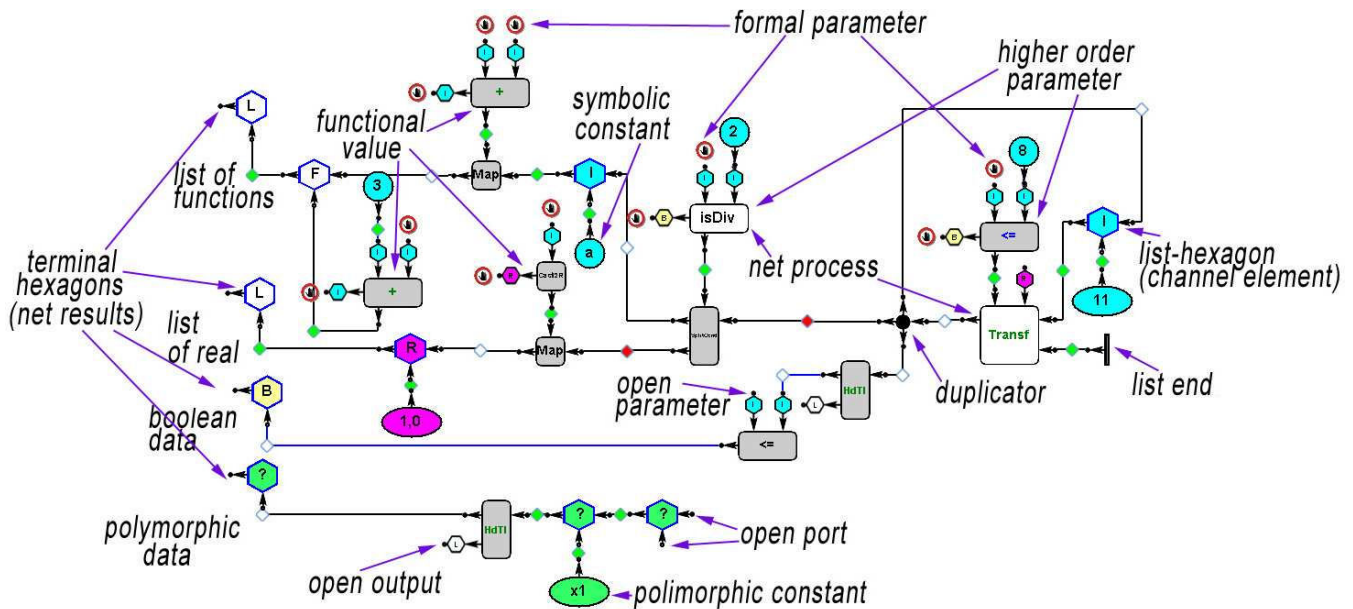


Figure 1. A NiMo Program example

2. NiMo elements and peculiarities

2.1 Graphic syntax

The NiMo graphic syntax is simple and concise. Figure 1 illustrates the main graphic elements: *rounded rectangles* represent processes 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.

Data are represented by *hexagons*, which are placeholders that can be bound to a value. If not, they can be interpreted as free anonymous variables. Hexagons are colored and labeled according to their type, polymorphic data are green "?". Terminal hexagons are the net outputs; the hexagons connected by horizontal arrows are list-elements. *Black dots* are duplicators for multiple uses of a given entry (as the multiple uses of a variable in an expression) and *circles* represent constant values (also colored according to their type)¹.

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. Clicking on a pair of ports connects them by means of an edge if both types are compatible, or else an error message is generated. Therefore nets are type safe by construction.

Edges have a state that is shown by means of a colored *diamond*. When a process output is connected the diamond is white, incoming data items are connected with a green diamond. 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 by its precedent process.

Let's observe that on the left of figure 2, process interfaces have an out port on the bottom. It is not one of their outputs but their value as a functional data. This out port disappears whenever one output of the process, or all its inputs are connected (it is now a potentially active process that cannot be considered a data). Higher-order parameter processes are connected by this port as

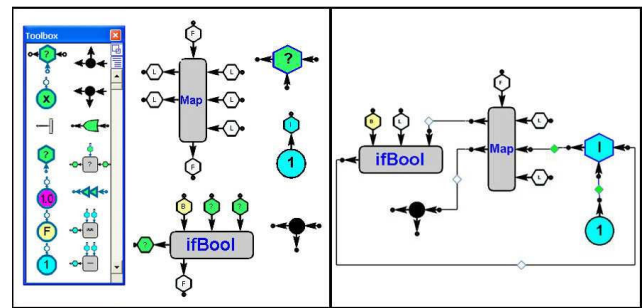


Figure 2. Interfaces and connections

can be seen in figure 1). When the connection is made, all the not yet connected 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.

2.2 Basic Processes

(bP): There is a set of predefined processes (grey rounded rectangles) for basic types and stream processing, 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. There are also some bPs with configurable arity as a *Map* with n input and m output channels, some of whose instances are analogous to functions *map*, *zipWith* and *zipWith3*. Also a *TakeWhile* and a *Filter* with as many inputs as outputs and an *Apply* process.

2.3 Net processes

(nP): They are user nets that have been parameterized and given a name to be used as a process in a new net and so on, allowing incremental net complexity up to any arbitrary degree. Their interfaces (the white rounded rectangles) are defined by means of a *net definition*. This mechanism is the graphic equivalent for Haskell

¹ The language syntax heavily includes the color, making it difficult to show it in a black and white paper.

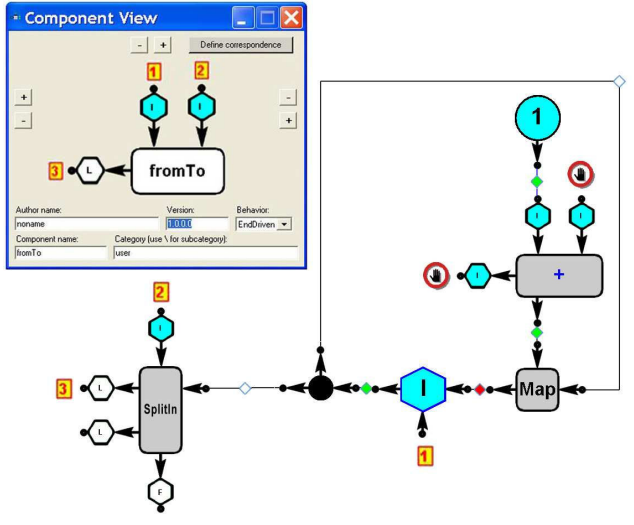


Figure 3. Net Process Definition

definitions with one non-conditional equation and only variables on the left. The association of bound variables is made by binding the in/out ports of a configurable interface with those open in/out ports of the net considered the formal parameters and results. Figure 3 shows the *net definition* for the process *fromTo*, that when $n \geq k$ is analogous to the Haskell's notation $[n .. k]$ (n and k are the parameters labeled 1 and 2). The process *SplitIn*, given a value and a channel, produces in its first output the initial elements of the channel until finding the value (this included). The second output returns the rest of the channel. The analogous tuple-returning function in Haskell can be defined as:

$$\text{SplitIn } v \text{ channel} = (o1 ++ [head\ o2], tail\ o2)$$

where

$$(o1, o2) = \text{break}(==\ v)\ \text{channel}$$

and then the Haskell code equivalent to the net definition is:

$$\text{fromTo } n\ k = \text{fst}(\text{SplitIn } k\ \text{list})$$

where $\text{list} = n : \text{map}(+1)\ \text{list}$

In execution, when the nP has to act, the interface is replaced by the net, maintaining the connections according to the corresponding bindings. A distinctive characteristic of NiMo is that any of the in/out ports could have been left open and nevertheless the application can be made anyway. This graph replacement mechanism implements the *expansion rule* of the process (section 5.4).

In addition, a not yet defined process can be built from a *configurable generic interface* and its net definition can be delayed, allowing top down development. Undefined nPs can be used also for symbolic execution.

2.4 Graphical type inference

The type inference system is a graphic generalization of the classical Hindley/Milner algorithm to deal with multiple outputs and curried/uncurried conversion [Clerici and Zoltan 2007]. The net has an associated type graph. It is incrementally built during the net construction and its evolution is optionally visible. Each process input/output is tied to a node in the type graph. This feature allows to identify easily the origin of the type error messages.

The net type graph is constructed starting from the *type descriptor* of each interface. In Figure 4 we can see the interfaces on the left of figure 2 with their type descriptors, and the resulting type graph after connecting them.

In NiMo a *process type* is a generalization of a functional type with multiple outputs (not a tuple), and whose multiple inputs are

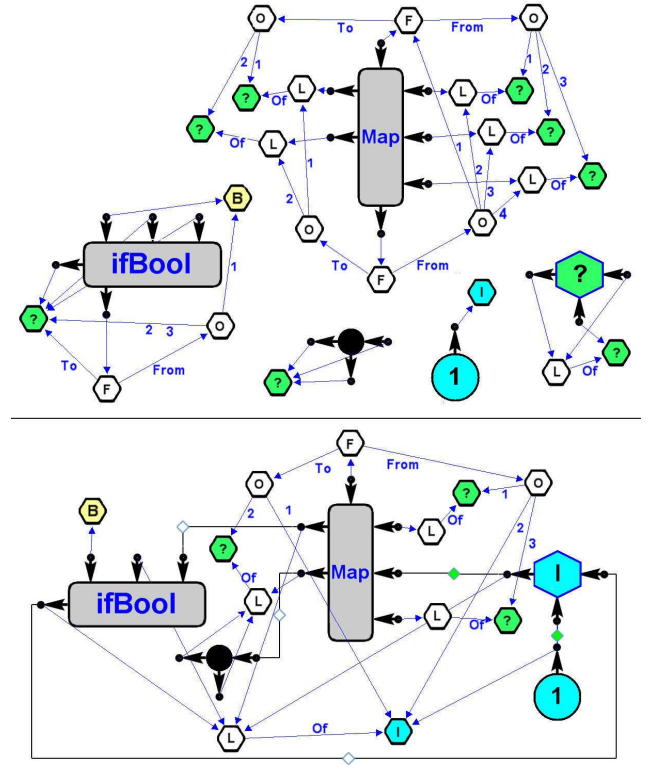


Figure 4. Type Descriptors and Type graph

interpreted in a curried or uncurried way depending on the context. Its graphical representation is a graph rooted with a hexagon F with outgoing edges labeled *From* and *To*. In order to describe this type textually, we use the \parallel notation to denote the type constructor for ordered parallel inputs or results, corresponding to the subgraphs with an O -hexagon root. In figure 4 the type graph tied to the out port at the bottom of the *ifBool* and *Map* interfaces describes their type. The textual denotation of the *ifBool* type is $\text{bool} \parallel a \rightarrow a$, where multiple occurrence of the same type variable a is graphically represented by a single polymorphic hexagon and a single edge with two labels (2 and 3). The *Map* type is $((a \parallel b \parallel c \rightarrow d \parallel e) \parallel [a] \parallel [b] \parallel [c]) \rightarrow [d] \parallel [e]$.

2.5 Main differences with Haskell

The main differences obviously arise from the fact of NiMo being totally graphic, which allows some features that cannot be imagined in textual languages. But it is also its main handicap because the visualization aspects, especially when nets grow, raise problems of difficult solution that textual languages do not have to face. The other big difference is that NiMo is not a production language; it is intended to be a workbench for incremental development, testing, debugging and tuning. Users have total control over their code and the state of its execution, they can interactively modify any program element and even undo execution steps, therefore the system acts as an online tracer and debugger. On the other hand, despite NiMo is a general-purpose language the data flow ingredient makes it especially suitable for stream programming style. Haskell is not oriented to a specific programming style, but it is also suited for stream handling because its Prelude provides a wide set of higher order list functions. A tool for translating NiMo programs to Haskell has been partially developed [Clerici et al. 2006].

Specific differences are the following:

- The net is the code, but also the computation state.
- Partially defined nets are open programs (having free variables) that can be executed.
- Execution is step by step and between steps any net element can be modified, deleted or added.
- Each step produces a new program that can be stored and recovered later.
- Type inference is incremental and only type safe nets can be constructed (type errors cannot take place in execution either).
- Execution errors can be ignored if they do not affect the net results.
- Symbolic constants of every type, included polymorphic, are admitted allowing symbolic execution.
- Not yet defined nPs do not inhibit execution.
- Functions are showable and then first level functional results are always valid.
- Processes are neither curried nor uncurried but they can behave in one or the other way depending on the context. Their polymorphic type is:

$$(?i_1 || \dots || ?i_n \rightarrow ?o_1 || \dots || ?o_m)$$

with $n, m \geq 0$ and $n + m > 0$

- Partial application with out-of-order parameter binding in multiple input processes is allowed.
- Processes can execute even if any of their non-needed inputs is left open.
- Multiple output processes may have multiple behaviors depending on which of their outputs are left open.
- There are multiple-arity processes.

Limitations (in the current version):

- User definitions cannot be made on patterns.
- There is neither overloading nor subtyping.
- Polymorphism is parametric (Miranda style, without inclusion of classes).
- There are not user-defined types.
- There is no input/output peripheral interfacing (open ports and terminal hexagons play this role. Input files are modeled as a nP producing a channel).

3. Executing nets

NiMo is a non-strict parallel language, but there are not explicit constructs to indicate parallelization. In the NiMo model all processes *selected* to act are supposed to execute in parallel at the same execution step. An execution step is a transition from one net to the next, where all the selected processes have produced the corresponding graph transformation. From the user point of view all of them have acted in parallel.

A bP *may* act only when all its *needed inputs* are *evaluated enough* which depends on each process. For example, the conditional process *ifBool* needs its first in port to be connected to a constant value; the others can be connected to any subnet or even be open. The result of the process execution is the elimination of the *ifBool* interface, and the reconnection of the subgraph that was connected to its out port with the subgraph connected to its second or third in port. If this port is open the in port connected to the *ifBool* out port also becomes open, and in any case the subnet connected

to the other in port becomes disconnected. This graph transformation is defined by means of the *execution rule* of each bP (Section 5.3).

On the other hand, (as was said in section 2.3) a nP always *may* act because it only requires applying its expansion rule, which can be made even in case of having all its in ports open.

Determining which of the processes that may act are selected depends on two factors: the existence of an explicit request and the evaluation mode that the process has, because in NiMo the evaluation order is not a uniform policy, each process has its own evaluation mode that can be set globally (for all) or locally for each process. Moreover, modes can also be changed during execution.

A request on a process is signaled by means of a red diamond in any of its output ports, and selected processes are explicitly marked by means of a red frame around their interface.

3.1 Process Modes

A bP can have five modes. They are (from more passive to more active): Disabled, Demand-Driven, End-Driven, Data-Driven and Weak-Eager.

Demand-Driven and Data-Driven are the usual ones in lazy functional and data-flow models respectively. A Demand-Driven process only acts when it is required and a Data-Driven process as soon as it has enough data to act.

An End-Driven process has almost the same lazy behavior as a Demand-Driven one but it also acts if the action produces the process elimination. For instance, whenever an input of *Map* is empty it can be reduced; operators on basic types reduce as soon as their parameters are constant values, etc. This mode allows net size simplification and therefore a better use of the screen real estate.

In Weak-Eager mode a process produces its results in a continued way, if it does not have enough data then it requests the needed values. The process behaves as if it were always requested to act.

Oppositely, Disable is a mode to “freeze” a process; it will not act under any circumstance. It is useful to isolate subnets still incomplete or being delayed for any reason, and also for symbolic execution as well as other interesting applications that are outlined in section 3.2. This mode is the only exception to the slogan common to all the other ones:

If the process is required it executes if it may, or else it requires the provider processes for its not sufficiently evaluated inputs.

On the other hand, a nP only has three possible modes: Disabled (unable to expand), Demand-Driven (only expands if it is requested), and Auto-Expand (always expands).

Modes attached to processes establish different degrees of activity for them. If we consider the domain of contexts where the process can act there is an inclusion relation associated to the following ordering on modes²:

For bPs:

Disabled \leq Demand-Driven \leq End-Driven \leq Data-Driven \leq Weak-Eager
and for nPs

Disabled \leq Demand-Driven \leq Auto-Expand

In figure 5 we can see that two processes are marked to act. One is a *Map* process in Data-Driven mode; it is not required (white diamond in front of it) but has all its needed parameters to act. The other is *INCR*, being Demand-Driven and required (a red diamond in front of it). For a nP it is enough to be expanded (even in case of having had its parameter open).

²This ordering is used in the attribute context condition of rules in section 5.2.

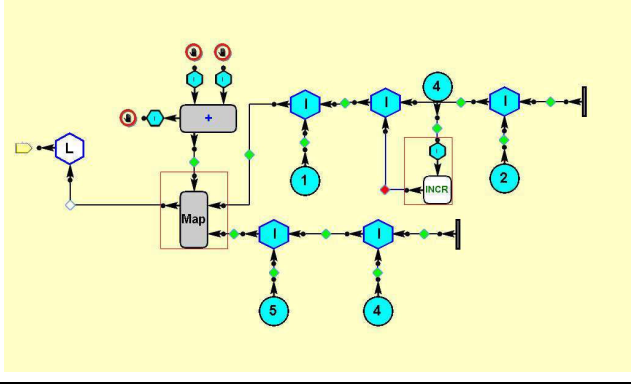


Figure 5. *Map* and *INCR* are ready to act

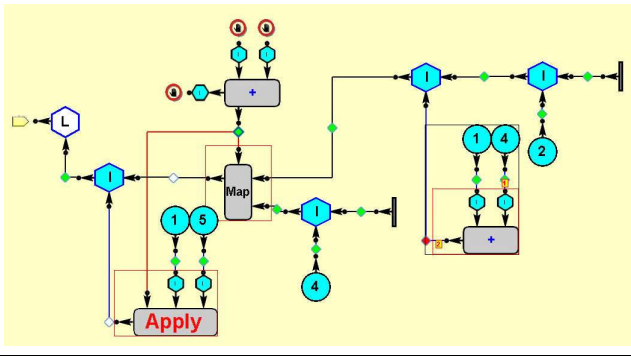


Figure 6. Graph after the execution step

In figure 6, we see the net in the next execution step. *Map* has acted using its execution rule, *INCR* using its expansion rule, and three processes are marked to act.

3.2 Customizing parallel evaluation

Different semantic models can be implemented in NiMo by using modes. A total lazy parallel behavior is attained by setting to Demand-Driven all the processes (at the first level and within the nPs) except the outermost ones (those nearest to the net outputs) which are set to Weak-Eager. If an outermost process is a nP its mode is Auto-Expand, and its own outermost processes are also set to Weak-Eager or Auto-Expand and so on. This policy is the most conservative one; only the really necessary processes are selected to act avoiding channel population and non-desired or even erroneous values. But in most cases it is also the slowest way to produce the results or to reduce the net size.

Setting the non-outermost bPs as End-Driven instead of Demand-Driven allows net simplification. For this reason in NiMo the default mode for a bP is End-Driven, in spite of the fact that in most cases a Data-Driven policy is also safe and more efficient. Setting all processes as Data-Driven gives the usual semantics in data flow approach. On the other hand, an eager semantics cannot be emulated in NiMo by changing modes, because Weak-Eager processes require only their needed inputs. By setting all processes as Weak-Eager, the net behaviour is somewhat similar but not the same because the analogous to the most external redexes here can be reduced before its not necessary internal redexes. In fact, in an eager semantics the notion of needed parameters does not apply.

Combining modes can be used to increase the implicit parallelism, deal with synchronization and to regulate channel population. For instance, in the example of binomial in figure 7 the pro-

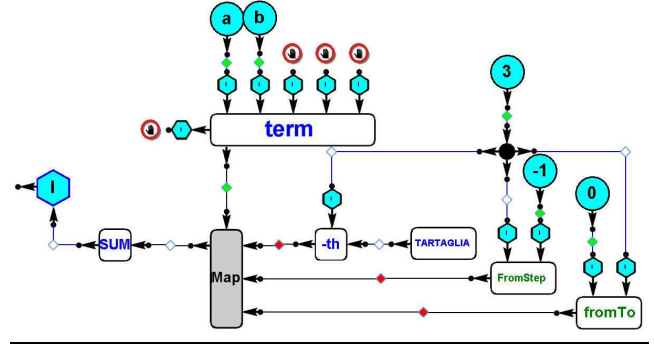


Figure 7. Newtons binomial

ductive rhythm of *Tartaglia* is by far slower than *fromTo* and *fromStep*, but *Map* consumes the three heads at the same time. Setting the processes of *Tartaglia* to a more active mode than that in the processes of the other two is the way of synchronizing them, and also avoiding unnecessary values to be produced in the unbounded output of *fromStep*. Of course they will be erased as soon as one of the other two inputs of *Map* ends, and therefore the final result would be the same even setting Weak-Eager mode to every process.

Besides, the evaluation modes could be used for activating/deactivating subnets during experimentation. This possibility allows partial testing of the already complete subnets, visualizing speculative calculations of intermediate results, and it is also the way to transform by execution some subnets into their results to produce a simpler version of the program. Disable mode delays the execution of the incompletely defined parts and prevents evaluation of symbolic values.

Also, a very interesting consequence of having disabled processes is that generative and multi-stage programming becomes very natural [Taha 2003]. 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 to generate different topologies for sensor networks of variable size and afterward simulating their behavior in [Clerici et al. 2009].

Another possibility somewhat related though not properly multi-stage programming, is using modes to execute an algorithm in successive stages to reason about the intermediate levels. To illustrate this form of “multi-stage execution” let’s consider again the binomial example. Figure 8 shows the first time execution stops when setting a Demand-Driven mode to *term* instead of Auto-Expand. Now if we set all nPs’ mode to Auto-Expand, execution continues up to producing the full expansion of the binomial and then stops. But in case of having run the program with numeric constants, say 2 and 3 for *a* and *b*, and now changing again the mode for all processes to End-Driven (or Data-Driven or Weak-Eager), the final numeric result (125) is obtained.

4. Graph Operational semantics

The NiMo operational semantics is given in terms of graph transformations. The transformation of the net at each execution step is produced by the consecutive application of three kinds of rules: execution/expansion, cleaning and marking. Rules of the same kind are applied in parallel until the closure of the rule set application. They are described in section 5. In the rest of this section the main elements to define the graph operational semantics are presented.

Scheduling parallel execution is based on coordination tags set on processes and on edges. To describe it we will refer to the se-

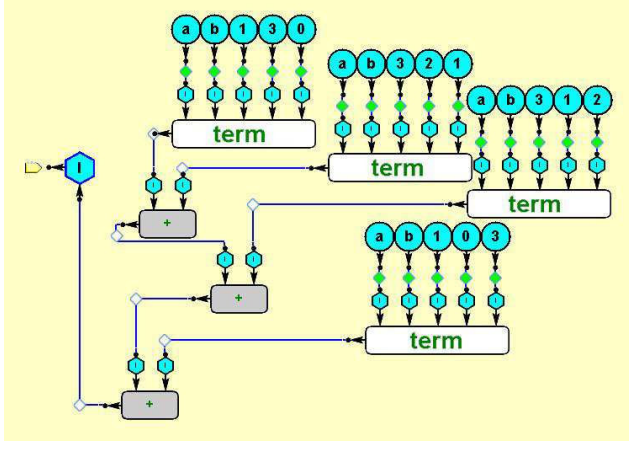


Figure 8. Binomial intermediate result

mantics given in [Baker-Finch et al. 2000], though the differences between both languages and approaches are significant. The operational semantics for GpH, is defined on a restricted subset of the language named GpH-core. It is a λ -calculus extended with numbers, recursive lets, and sequential and parallel compositions. Definitions are modeled as local in a let-expression and the let-rule generates the bindings. In our graph grammar semantics, definitions correspond to the set of execution and expansion rules, and the expression to be evaluated is the working graph. Regarding the scheduling the main difference obviously lies on the variety of modes in NiMo.

On the other hand, in this paper the operational semantics describes a reduction sequence from an initial global configuration to a final global configuration where *main* turns into a WHNF (analogous to say that the terminal hexagon is connected with a green diamond). Here we only define the state transition for a single execution step. This small step semantics approach is taken on one hand because in NiMo, when executing step by step, the user can interactively modify the program, so the final state of one step is not always the initial one for the next. But even in a continued execution the goal is not necessarily to reach an output result. The result could be any interesting final state. For instance, in generative programming the final result is a new executable net, and in general, the existence of disabled processes may produce the execution stop, which can be continued by only changing their modes. So the notion of final result strongly depends on the computation goals.

On the other hand, the NiMo process type is a generalization of functions and therefore a process cannot be equivalent to a λ -abstraction. Processes have multiple inputs that are neither tuples nor curried parameters and can be applied in any order. Therefore to describe them in that way we would need an extended notation where bound variables on the left of a λ -abstraction should be a kind of $\lambda x_1 || \dots || \lambda x_n$. The same occurs with the multiple body expression $e_1 || \dots || e_m$ where any of them can be left disconnected in the application.

But even in cases of single output and curried interpretation of inputs there are other significant differences because NiMo can execute having free (anonymous) variables (the open ports), while in [Baker-Finch et al. 2000] only closed terms are considered. In particular NiMo allows processes with open-parameters to execute. It is like saying that β -reduction not necessarily requires having the argument to be performed. As an example $(const\ 3)_-$ reduces to 3 (provided that its mode is not Disabled).

4.1 Connection States. Edge Tags

Tags on edges (the diamond color) indicate, on output edges, whether the process is requested, i.e. if any of its results is demanded. On input edges they allow a bP not to be aware of its input edges' neighbors, except in a few cases. In general, just knowing if all the diamonds in its needed inputs are green is enough to know if a bP may execute or not.

White or red diamonds precede subnets that are equivalent to non-WHNF expressions, because the head is a process providing one of its outputs, and in NiMo only data items (hexagons, circles or bottom port connected processes) can be WHNF (all of which are preceded by green diamonds). But sometimes it is not enough information, for instance if the mode of *Map* is End-Driven, it is necessary to know if the neighbor is a list-hexagon or a list-end to know if it can die. Also, some bP may need more than one list element in an input channel to apply its execution rule (corresponding to the Haskell pattern $(a:b:x)$).

Diamond colors reflect the state of the connections between port interfaces. Consequently, in the graph grammar rules diamonds might have been modeled as attributes on the edges that connect interface nodes. But in this case the pattern should have included necessarily the other extreme of the edge, requiring many similar rules, one per each kind of neighbor instead of abstracting all of them in a common pattern (look for instance at the first rule in figure 14).

A connection between a pair of in out ports is modeled as a pair of edges (out port-to-diamond, diamond-to-in port); then connection between interface ports is the composition of both connections. As an important consequence of modeling port connections in this way, the graph patterns in the rules have at most one process node and diamonds as outermost nodes, a helpful trait to attain rule independence.

4.2 Parallel Execution Scheduler

[Baker-Finch et al. 2000], describes a labeling procedure for qualifying bindings (from variables to expressions) with an activity degree. Labeled bindings correspond to threads. The labels are: *Inactive* (I), *Blocked* (B), *Runnable* (R) and *Active* (A). We don't have an equivalent for Runnable because in the NiMo model the number of processors is unbounded, however there exists a similarity with the underlying concepts of the other three ones, but in our case labels are assigned to processes, not to threads. For us:

- A stands for a process that according to its mode and context conditions can apply its transformation rule.
- B stands for bPs that are required or Weak-Eager and have not enough evaluated inputs to act; they must require to the corresponding providers.
- All the remaining processes are labeled I.

The label of any bP depends on its own definition, mode, and context conditions. The later are determined looking at the bP inputs, and the diamond's tags of the outputs. On the other hand, the labeling function for a nP only depends on the process being required or not and on its mode. A nP is never blocked because its expansion rule can be applied no matter the state of its inputs.

Therefore a bP can be labeled with A, B or I, and a nP can be labeled A or I.

4.2.1 Labeling processes

The signature of the labeling function for bPs is:

$$LAB : OD^+ \times P \times I^* \rightarrow \{A, B, I\}$$

Where od_k in OD are the diamonds in front of a p process in P (leaving the outports) and i_k are the inputs of p (values or channels connected to its in ports).

LAB is defined using the predicates $FORCED$ (that is a general condition for every bP), and MAY and DIE (that depend on each process).

$$\begin{aligned} \text{FORCED}(p, \langle od_1, \dots, od_n \rangle) &\Leftrightarrow \exists i od_i = red \\ \vee \text{MODE}(p) = \text{Weak-Eager} & \\ \text{MAY}(p, \langle i_1, \dots, i_n \rangle) &\Leftrightarrow \forall \text{needed } i_k \\ \text{are evaluated enough to act } p & \\ \text{DIE}(p, \langle i_1, \dots, i_n \rangle) &\Leftrightarrow \forall \text{needed-to-die } i_k \\ \text{meet conditions to die } p & \end{aligned}$$

LAB definition is the following:

$$\text{LAB}(\langle od_1, od_2, \dots, od_n \rangle, p, \langle i_1, i_2, \dots, i_m \rangle) = \begin{cases} B & \text{FORCED}(p, \langle od_1, od_2, \dots, od_n \rangle) \\ & \wedge \neg \text{MAY}(p, \langle i_1, i_2, \dots, i_m \rangle); \\ A & \text{MAY}(p, \langle i_1, i_2, \dots, i_m \rangle) \wedge \\ & (\text{MODE}(p) > \text{End-Driven} \\ & \vee (\text{MODE}(p) = \text{End-Driven} \\ & \wedge \text{DIE}(p, \langle i_1, i_2, \dots, i_m \rangle)) \\ & \vee (\text{MODE}(p) > \text{Disabled} \\ & \wedge \exists i od_i = red)); \\ I & \text{otherwise.} \end{cases}$$

The labeling function for a nP is:

$$\text{NLAB} : P \times OD^+ \rightarrow \{A, I\}$$

$$\text{NLAB}(p, \langle od_1, od_2, \dots, od_n \rangle) = \begin{cases} A & \text{MODE}(p) = \text{Auto-Expand} \\ & \vee (\text{MODE}(p) = \text{Demand-Driven} \\ & \wedge \exists i od_i = red); \\ I & \text{otherwise.} \end{cases}$$

4.3 Execution Step

Each computation step is the finite state transition resulting from the composition of derivations produced by rules in four disjoint grammars:

- G1: *Execution rules* defining the bP execution actions for Active processes (the left side of the rule contains a red frame node connected to the process). There might be several rules for each bP according with the possible patterns on the process inputs. They are described in 5.3.
- G2: *Expansion rules* for an active nP to produce the replacement of its interface for its net definition. The left side of the rule contains a red frame node connected to the process. There is one rule for each process. They are described in 5.4.
- G3: *Garbage collection* rules to clean the net removing the non-productive subnets. They are described in 5.5.
- G4: *Marking* rules to produce a red frame for Active processes or to request the process providers for Blocked processes. According to the patterns on their inputs and attribute context conditions on their tags one or the other kind of rule will be applicable. There might be several rules for each process. They are described in 5.6.

The application order of each set of rules is the following:

1. Rules in G1 and G2 are applied in parallel.

Final state: In the resulting net all the processes marked with a frame in the previous step have done their execution action. All frames has been removed.

2. Closure of the parallel application of rules in G3.

Final state: All the non productive elements produced in stage 1 have been removed from the net.

3. Closure of the parallel application of rules in G4.

Final state: all Blocked processes have a red diamond in all their needed inputs not evaluated enough, and all Active processes are marked to be executed in the next step.

One execution step is the transition $S_i \Rightarrow S_{i+1}$ where \Rightarrow is defined as $\xrightarrow{G4} \circ \xrightarrow{G3} \circ \xrightarrow{G1 \cup G2}$ and \xrightarrow{G} denotes the closure of rule application for grammar G which is defined below.

Let's note that at the first execution step there are no previously marked processes therefore $\xrightarrow{G1 \cup G2}$ is the identity transformation.

Closure of rule application : If $S \xrightarrow{G} S'$ is the transformation of a graph S into S' by means of the parallel application of rules in G , the closure of rule application is defined as:

$$S \xrightarrow{G} S' \text{ if:}$$

1. $S \xrightarrow{G} S'$ and
2. there is no S'' such that $S' \xrightarrow{G} S''$

5. Graph transformation rules

5.1 Attributed graph grammars concepts

Graph Grammars consist of a set of rules, which are applied to an initial working graph. Rules are pairs of graphs named *Left* and *Right* sides, whose elements are put in correspondence by means of a set of *mappings* relating nodes or edges present on both sides. A rule can be applied when the pattern-graph Left can match the working graph, meaning that there exists an inclusion from an instance of Left into the working graph that fulfills the application conditions of the rule. Attributed graph grammars associate a set of tags to each node or edge. Some of these tags are visual as the shape and color of a node or the kind of an edge line, and the others can be variables of any allowed type. Most of the graph transformation systems fix some of these tags defining the kind of node or edge, and the other ones are called attributes, but conceptually all together define the node or edge type. Rules can also change the attribute value of a related node or edge in Right, and a particular configuration of attribute values can be used as a precondition on the left side (attribute context condition).

Matching with Left is the positive pre-condition for rule application, but there are also negative conditions expressed by a second pattern-graph Neg that excludes those instances of Left also matching with Neg; these rules are triples (see for instance rule in figure 11). The rule application result is the replacement of the found occurrence of Left by the corresponding instance of Right; the graph transformation is also called *derivation* or *graph transformation step*.

Mapping from elements on both sides of the rule is usually indicated with labels. Same label on both sides mean that the element remains in the transformation result, on the contrary the element is removed in the result. Elements on the right not related to any one on the left are new fresh elements appearing in the result. The elements not present in the rule are not transformed. In terms of algebraic theory, matches and rules are *graph-homomorphisms* and the formal semantics of rule application is given by a categorical construction known as *pushout* [Rozenberg 1997].

5.2 Graphical notation used in the rules

In the figures we have used a NiMo style syntax. The fix tag for nodes (interfaces or diamonds) is the shape, and for edges their horizontal or vertical orientation. The color for diamonds, processes and hexagons and the name of process or type label in

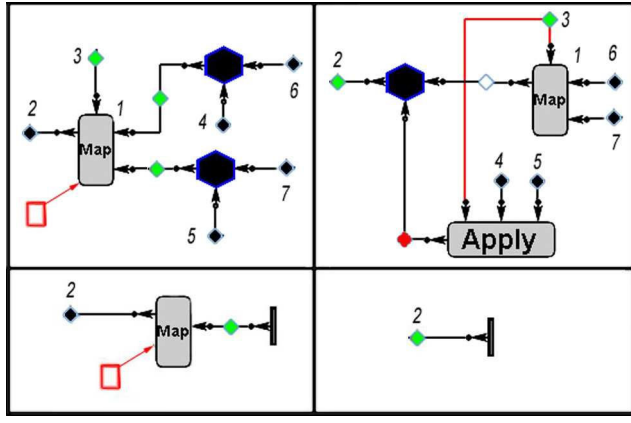


Figure 9. Execution rule of *Map*

hexagons are not fix. They are painted or named when the value must be one in particular, when any value is allowed they are painted in black, and when the attribute value is a context condition it appears as a variable in a dashed rectangle (see rule 14). The process-mode tag is represented in this way. For edges, the non fixed tag is the number of the port they connect. When the rule is non ambiguous, we omit the attribute. If an attribute in a dashed rectangle does not appear in Right its value does not change. On the other hand, edges from an out port interface to a diamond are drawn with the arrowhead at the beginning instead of at the end.

5.3 bP execution rules

The execution rules of a bP are the analogous to function definitions in Haskell, i.e. the equation/equations defining a function, and they are also defined by means of patterns. When applied on the current working graph the sub graph matched with the pattern on the left side of the rule is transformed. The resulting graph is analogous to the resulting expression after reducing a beta or delta redex.

As an example figure 9 shows (a simplified version of ³) the pair of rules for a process *Map* with one output and two input channels. The red-rectangle node connected to the *Map* interface on the left side of both rules, corresponds to the red frame shown in figure 5. It is the label A produced in the previous execution step by the corresponding marking rule in figure 14. Notice that it is removed in the result.

The first rule is the equivalent to the equation

$$\text{map } f (a : x)(b : y) = f a : \text{map } f x y$$

The application of this rule erases both input list-hexagons and re-connects their values (the subgraphs a and b connected to diamonds 4 and 5) as second and third inputs for a new interface *Apply*, whose first input is connected by a red edge to the diamond 3. The parameter process *f* (not present in the rule) tied to diamond 3 is shared by *Map* and *Apply* processes. A new list-hexagon is inserted in the output channel of *Map* and the preceding diamond 2 changes to green. The new output diamond of *Map* is white because *Map* has already produced its result. The input channels of *Map* are connected to diamonds 6 and 7, i.e to the respective remaining channels *x* and *y*.

The screenshot in Figure 6 shows the result of applying the implementation of this rule on the net in figure 5.

The execution rule of *Apply* will produce the graphical application in the next step (giving *f a b*). This is the way a higher

³ In fact *Map* has a more generic execution rule to handle multiple input and output channels.

order process as *Map* applies its functional parameter (i.e. a process connected to it by the bottom out port). By means of *Apply* the parameter process acquires its own parameters, and its output port produces the value of the new list element generated by *Map*. In the general case, *Apply* has as many outputs as the higher order process, and its process parameter *f* must also have at least as many outputs (some of them could be marked for being left open in the result of the application).

The second rule is a pattern for the case when at least one of the two input channels ends. Corresponding to equations $\text{map } f x [] = [] \vee \text{map } f [] y = []$

The *Map* process disappears in the result and also the list-end: the list-end on the right is a new list-end ⁴. The other elements not present in the rule, i.e. the process parameter and the other channel, remain in the result. But then they will be eliminated by the garbage collector rules because they have lost their consumer and therefore are non-productive subnets.

It is important to remark that all the rules have been defined to be sequentially independent, a property equivalent to say that two rules can be applied in either order with the same result. Let's observe that diamonds are the most external nodes, so the only case of pattern overlapping of two execution rules could be the same diamond acting as first and last node of each respective pattern. This trait simplifies very much the mechanism to obtain parallel independence ([de Lara et al. 2004] for details on this subject).

5.4 nP expansion rules

This set of rules produce the substitution of a single interface node by the sub graph corresponding to the nP net definition (described in 2.3). Mappings on diamonds preserve the bindings established by the user. In figure 10 the expansion rules for the nP *FromTo* defined in figure 3 are shown. As happened with execution rules, the pattern in Left includes the red rectangle, it was produced by the rule in figure 15.

Let's observe that according to the rules, expansion can take place even in the absence of one or both parameters. The one on the top corresponds to the case where both parameters are connected. The remaining ones have a negative condition to discriminate the other cases.

5.5 Garbage collection

The garbage collection rules are cleaning rules to erase all the non useful elements from the working graph. Figure 11 shows an example for list-hexagons. The rule has a negative precondition to prevent its application when this pattern matches. In this case connected list-hexagons are not removed; only those with their output port open can be erased.

Once the list-hexagon is eliminated its own providers have to be eliminated also, and so on.

The only immune interfaces are the output hexagons because they are the net outputs. For all the other interfaces there are garbage-rules whose closure eliminates all the subnets that do not contribute for the net outputs production.

Once more, the existence of multiple outputs makes a difference because "closed" cycles can happen. Let's suppose that the leftmost process in figure 12 was the provider for the non-empty channel of *Map*. The process garbage rule doesn't match because the process remains connected by the other output, but the subnet it heads has lost the connection with the net outputs, and therefore must be erased. This issue requires a more sophisticated algorithm for detecting such cases ⁵.

⁴ It cannot be the same because both types may be different.

⁵ In The NiMo environment implementation the algorithm is applied every 10 execution steps and can be optionally deactivated.

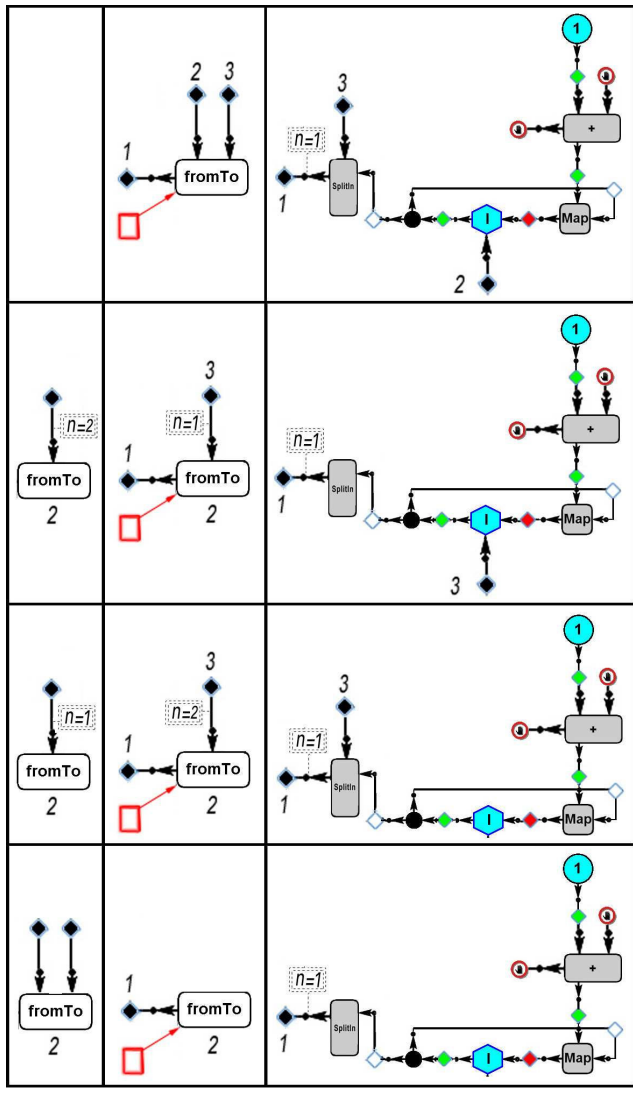


Figure 10. FromTo expansion rule

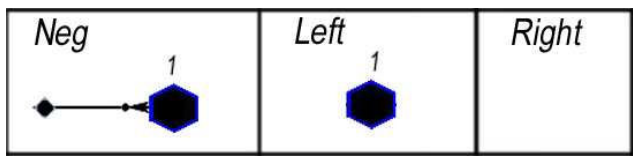


Figure 11. Garbage collector rule for list hexagons

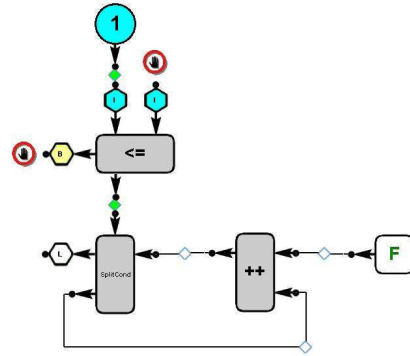


Figure 12. Non-productive subnet

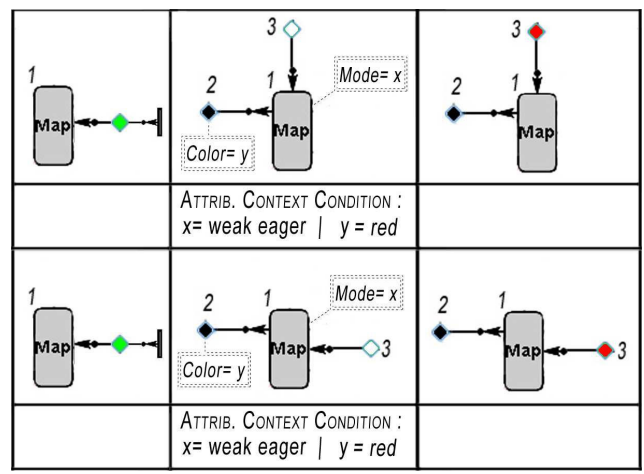


Figure 13. Blocked Map Action

5.6 Marking rules

Rules in this set define the conditions for labeling a process as Active or Blocked and perform the corresponding transformations.

B-processes must require their providers for all needed inputs that are not sufficiently evaluated. The corresponding graph transformation is to set these diamonds to red. In figure 13 an example of rules to perform this action, for the bP *Map* is shown.

A-processes must be marked to be executed in the next step. The corresponding graph transformation paints a red frame around their interface.

As it happens with the execution rules, the conditions for a bP to be active depend on each process and are also defined by patterns on its inputs. Figure 14 shows the rules for labeling a *Map* process as active, and the rule in figure 15 applies to any nP. In both cases rules have a negative condition, to ensure that only not yet active processes are activated.

The pattern of the rule on the bottom of figure 14 reflects the configuration enough to fulfill the DIE predicate: having a list-end connected, it does not matter what happens with the other in ports that might even be open. In the rule on the top, the pattern ensures the MAY predicate: all the inputs have a green diamond, which means that the higher order process and both channels are evaluated enough. The context conditions in both rules correspond to the attribute values that meet the conditions defined in function LAB.

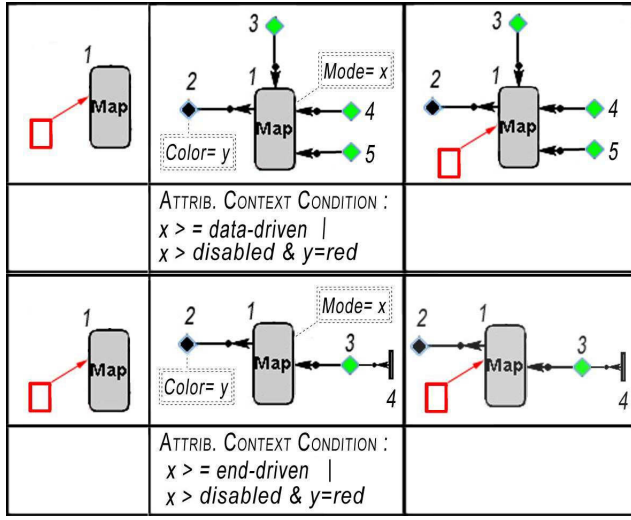


Figure 14. Map activation rules

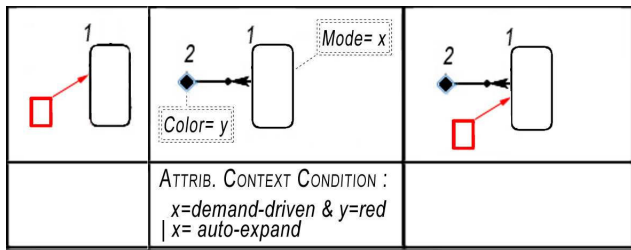


Figure 15. nP activation rule

The rule for marking any nP process as active is shown in figure 15. The context conditions correspond to the ones described in function NLAB .

6. Related Works

Along the article we refer in extenso to GpH, so in this section we will mention other languages. In general, languages have a fixed model of computation: Lazy, Data-Driven, etc., and may have additional constructs to change the mode expressions are evaluated.

StreamIt It is designed to facilitate the programming of large streaming applications. Alike NiMo, StreamIt exploits pipeline parallelism. In the compiler shown in [Cheng et al. 2008] a technique to fuse stateless actors is implemented. One example of fusion could be to replace the composition of *Map*'s by a single *Map* with the composition of both higher order parameters. We have found that depending on the data rate, fusion can decrease pipeline parallelism.

Strategies [Nakamura and Ogata 2001] is a mechanism that can be attached to symbols to force or delay the evaluation of certain parameters and change the default evaluation model. In NiMo, user components (nP) do not have an associated strategy. The strategy is defined by the modes of their internal processes. Regarding the bPs, in NiMo each one has its own fixed strategy. (For instance *ifBool* always evaluates its first parameter only). Strategies are integrated in **Maude** [Maude Home Page].

Ptolomy [Davis II et al. 2001] is based on actors having ports as communicating interfaces. Ports can be input, output or both. Ptolomy deals with continuous data, while NiMo only with discrete. Actors in Ptolomy somehow correspond to processes in

NiMo. Actors have parameters that are not visualized on the actor; they are shown in separate windows. In NiMo all the program state is visualized. In Ptolomy there is also a notion of mode: a domain that defines the communication semantics and the execution order among actors. This is implemented by two classes: a director (which is a scheduler) and a receiver class. Both are taken from an extensible library (determining how the input values are consumed or observed). There is a variety of domains and corresponds to a uniform evaluation policy for the whole actor. For instance Ptolomy process network domain corresponds to NiMo's Data-Driven policy.

Of the languages mentioned previously, the only visual one is Ptolomy. Ucello [Ellner and Taha 2007] and [Dami and Vallet 1996] languages share some visual characteristics with NiMo and are also functional.

Ucello is a graphic language, that uses lambda graphs. Edges either connect a term with a sub-term, or tie each use of a bound variable to the bound variable definition. The core language includes function abstraction and function application. It was extended with staging constructs, \sim , $\langle \rangle$ and $!$, the usual ones for textual multi-staged languages. For example NiMo's Disable mode for a process corresponds to a bracketed function with no preceding \sim at any stage. The language does not have the notion of flowing data.

[Dami and Vallet 1996] present a graphical language of interconnected boxes. Grey boxes are used as an apply box. Rewrite rules describe execution rules. The rules used have three elements: the left graph, the right graph and a third component that holds, in a graphical way, the binding between left and right graphs. Alike in NiMo, boxes without outgoing arrows are deleted by a garbage collector.

7. Conclusions

We have presented the execution model for a true graphic language that integrates the functional and dataflow paradigms. The mixture results in a very powerful computation model because the net is both the code and the computation state, giving full access to a run-time modifiable computation graph. Every element can be dynamically completed or changed, and the user is also enabled to set and even to modify the evaluation order interactively in a simple and very intuitive way.

This total interaction is unthinkable for textual languages, and some aspects coming from the integration of paradigms are not present in pure functional languages. Therefore some of the mentioned NiMo features are not easily exportable. However, a number of the ideas here presented are innovative and may result of interest. In concrete, regarding the main subject of this paper, there are two outstanding aspects: the variety of modes defining several levels of increasing activity, (in contrast with the usual dichotomy laziness vs. eagerness), and mainly the process-centered approach allowing customizable evaluation scheduling. Both together provide a notable flexibility in program tuning for resource usage optimization.

Regarding the five modes definition criteria, we have not also included an Eager mode because NiMo has a non-strict semantics. Weak-Eager provides the eagerness enough to get the subnet the process heads to produce its results in a continued way. Except for simulating the eager semantics in a direct way, we have not found any advantage in having a mode for blocking the process until the non-necessary inputs are produced.

Inasmuch as the overall aspects of NiMo development, the paradigms fusion has meant a big challenge that required to figure out many creative solutions to make 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, total visibility has also required to find a graphic

notation for many “internals” to make them understandable. Besides, the visualization aspects become critical when nets grow, which is a great difficulty that textual languages do not have to face. We are currently working on several aspects of net visualization such as a non-expanded view of net processes. Alongside, an extension of the bP catalog (with other built-in processes like a pseudo-random number generator) as well as a distribution version of NiMoToons (including an interactive tutorial) are now underway. On the other hand, NiMo’s current nP library has been developed mainly to show the language potential and programming peculiarities. Now we are looking for applications in an industrial setting to extend the range of possible users beyond the academic environment

Acknowledgments

We thank the anonymous reviewers for their helpful comments.

References

- AGG Home page. Agg home page, 2009. URL <http://user.cs.tu-berlin.de/~gragra/agg/>.
- Clem Baker-Finch, David J. King, and Phil Trinder. An operational semantics for parallel lazy evaluation. In *ICFP '00: Proceedings of the fifth ACM SIGPLAN international conference on Functional programming*, pages 162–173, New York, NY, USA, 2000. ACM. ISBN 1-58113-202-6. doi: <http://doi.acm.org/10.1145/351240.351256>.
- Chihhong Patrick Cheng, Teale Fristoe, and Edward A. Lee. Applied verification: The ptolemy approach. Technical Report UCB/EECS-2008-41, EECS Department, University of California, Berkeley, Apr 2008. URL <http://www.eecs.berkeley.edu/Pubs/TechRpts/2008/EECS-2008-41.html>.
- S. Clerici, A. Duch, and C. Zoltan. Implementing static synchronus sensor fields using nimo, 2009. URL <http://www.lsi.upc.edu/dept/techreps/>.
- Silvia Clerici and Cristina Zoltan. Graphical type inference. a graph grammar definition. Technical Report LSI-07-24-R, Dept. Llenguatges i Sistemes Informàtics, Universitat Politècnica de Catalunya, July 2007. URL http://www.lsi.upc.edu/dept/techreps/11listat_detailat.php?id=970.
- Silvia Clerici and Cristina Zoltan. A graphic functional-dataflow language. In Hans-Wolfgang Loidl, editor, *Trends in Functional Programming*, volume 5 of *Trends in Functional Programming*, pages 129–144. Intellect, 2004. ISBN 1-84150-144-1.
- Silvia Clerici, Cristina Zoltan, Guillermo Prestigiacomo, and Javier García Sanjulián. Diseño de un entorno integrado de desarrollo para nimo. In *VI Jornadas de programación y lenguajes-Prole 2006*, 2006.
- Laurent Dami and Didier Vallet. Higher-order functional composition in visual form, 1996. URL <http://cuiwww.unige.ch/OSG/publications/00-articles/TechnicalReports/96/ho-visual.ps.Z>.
- John Davis II, Christopher Hylands, Bart Kienhuis, Edward A. Lee, Jie Liu, Xiaojun Liu, Lukito Muliadi, Steve Neuendorffer, Jeff Tsay, Brian Vogel, and Yuhong 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, March 2001. URL <http://ptolemy.eecs.berkeley.edu/>.
- Juan de Lara, Claudia Ermel, Gabriele Taentzer, and Karsten Ehrig. Parallel graph transformation for model simulation applied to timed transition petri nets. *Electr. Notes Theor. Comput. Sci.*, 109:17–29, 2004.
- Stephan Ellner and Walid Taha. The semantics of graphical languages. In *PEPM '07: Proceedings of the 2007 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, pages 122–133, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-620-2. doi: <http://doi.acm.org/10.1145/1244381.1244402>.
- Maude Home Page. Maude, 2009. URL <http://maude.cs.uiuc.edu/>.
- Masaki Nakamura and Kazuhiro Ogata. The evaluation strategy for head normal form with and without on-demand flags, 2001.
- Grzegorz Rozenberg, editor. *Handbook of Graph Grammars and Computing by Graph Transformations, Volume 1: Foundations*, 1997. World Scientific. ISBN 9810228848.
- Walid Taha. A gentle introduction to multi-stage programming. In *Domain-Specific Program Generation*, pages 30–50, 2003.