

# Diseño de un entorno integrado de desarrollo para NiMo

Silvia Clerici <sup>a</sup> Cristina Zoltan <sup>a</sup> Guillermo Prestigiacomo <sup>b</sup>  
Javier García Sanjulián <sup>b</sup>

<sup>a</sup> *Universitat Politècnica de Catalunya, Dept. Llenguatges i Sistemes Informàtics  
Barcelona*

<sup>b</sup> *Universitat Politècnica de Catalunya, Facultat d'Informàtica de Barcelona*

---

## Resumen

NiMo es un lenguaje totalmente gráfico ejecutable para la programación en redes de procesos, del cual existe un prototipo completo desarrollado en el sistema de transformación de grafos AGG. NiMoAGG es muy limitado en los aspectos de visualización e interfaz con el usuario, fundamentales para ser una herramienta realmente útil. La construcción de un entorno integrado de desarrollo que mantenga la funcionalidad del prototipo y corrija esas deficiencias es una tarea compleja. En este trabajo se discuten las características que debería tener este IDE y los componentes necesarios para lograrlas. Se delinear los criterios seguidos en el diseño de NiMoToons, el núcleo del IDE actualmente en construcción, y también algunos aspectos de su implementación. Existe ya una primera versión de la interfase de usuario para la construcción de redes y facilidades de visualización. Por otro lado, se han desarrollado también herramientas de conversión entre programas en NiMoToons y NiMoAGG, y de traducción de ambos a Haskell y viceversa

*Palabras clave:* IDE - Lenguaje de Programación -Gráfico-  
Funcional- DataFlow

---

## 1. Introducción

NiMo (**N**ets **i**n **M**otion) [3,4] es un lenguaje totalmente gráfico inspirado en las representaciones en redes de procesos de programas funcionales perezosos, donde las funciones son vistas como procesos conectados por canales (listas usualmente infinitas), implementando la metáfora Data-flow de proveedor-consumidor. El objetivo primario de NiMo es facilitar la comprensión de los programas y servir como banco de prueba y experimentación. El usuario visualiza todas las transformaciones sufridas por la red en ejecución y tiene pleno control sobre cada una de sus componentes, pudiendo interrumpir la ejecución,

*This paper was generated using LATEX macros provided by  
Electronic Notes in Theoretical Computer Science*

retroceder pasos, y modificar la red o incluso alterar el orden predeterminado de ejecución, y retomarla desde ese punto.

El actual prototipo, NiMoAGG, implementado mediante gramáticas de grafos tiene la funcionalidad completa del lenguaje pero es muy pobre en cuanto a la visualización y la interfaz con el usuario. En este trabajo se presenta el estado alcanzado en la construcción de un entorno integrado de desarrollo que incluya la funcionalidad del prototipo actual, permita dibujar los grafos según las convenciones NiMo de ubicación y disposición relativa de nodos y arcos, y conserve lo mejor posible la estructura de la red que va siendo transformada. Además de las facilidades de visualización de ventanas, imprescindibles para tratar con gráficos que crecen mucho, el entorno debe incluir opciones de visualización compactada de la red. El entorno debe también facilitar el manejo de librerías y proyectos, y permitir la traducción a Haskell.

En la sección siguiente se resumen las características principales del lenguaje y su modelo de ejecución, a continuación se describe el prototipo existente y sus limitaciones. En la sección 4 se delimitan los objetivos a cumplir por un IDE capaz de salvar estas limitaciones. La sección 5 presenta el diseño del núcleo principal del IDE, llamado NiMoToons, y la siguiente las herramientas de conversión entre éste y NiMoAGG, y las de traducción entre NiMo y Haskell. Finalmente se describen los resultados alcanzados hasta el momento y la continuación prevista.

## 2. Características del lenguaje NiMo

Los programas NiMo son grafos orientados donde los arcos son canales de capacidad no acotada, por donde viajan en forma FIFO los datos (que pueden ser valores simples, estructurados, o funcionales). Los únicos identificadores son los de los procesos, éstos pueden tener más de una salida y estar parametrizados por valores que no son canales, e incluso por otros procesos. El lenguaje dispone de un conjunto de procesos básicos inspirados en los transformadores de listas de lenguajes como Haskell [9] y especialmente adaptados al procesamiento Data-flow.

El repertorio de símbolos gráficos es muy reducido como puede verse en la figura 1. Los arcos horizontales se utilizan para canales de datos, y los arcos verticales entrando a un proceso para parámetros que no son canales en el sentido Data-flow. El flujo de datos tiene orientación de derecha a izquierda, y por lo tanto las salidas finales de la red son los arcos de más a la izquierda.

NiMo sigue un modelo de ejecución Data-flow-perezoso. La activación de un proceso se produce por la demanda de algún otro del cual es proveedor, o del exterior si el proceso produce una salida de la red. Si el activado es un proceso básico y ya dispone de los parámetros necesarios producirá la transformación correspondiente, de lo contrario propagará la demanda sólo a los proveedores de estos parámetros. Si el proceso activado no es básico se expandirá, siendo reemplazado en el grafo por el subgrafo de la red que lo implementa.

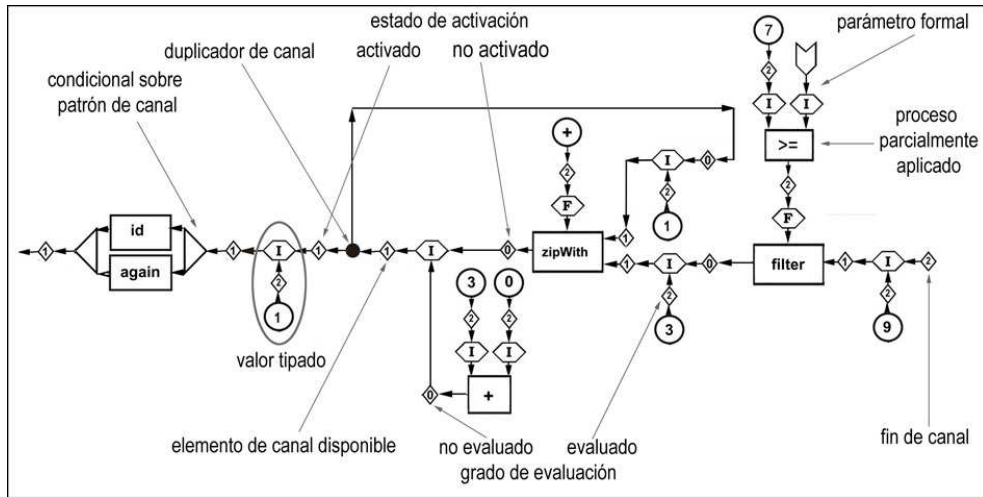


Figura 1. Sintaxis gráfica

La activación de procesos se visualiza en la red mediante diamantes coloreados que marcan los caminos donde se espera flujo de respuesta. Todos los procesos en condiciones de actuar se pueden ejecutar en paralelo por lo que no es necesario poner anotaciones para indicar ejecución paralela, o sincronización, ni establecer un orden de ejecución externo a la red. El control de ejecución viene dado por la política dirigida por demanda y regulado por la disponibilidad de datos. Sin embargo el usuario puede establecer una activación inicial de procesos que no obedezca a la demanda por propagación, e incluso cambiar el estado de activación de cualquier proceso deteniendo para ello la ejecución.

La ejecución muestra paso a paso todas las transformaciones de la red inicial. El ambiente de desarrollo permite interrumpir la ejecución, retroceder pasos, e incluso interactuar cambiando partes o completando redes incompletas y retomar la ejecución directamente. Esta es una de las características más peculiares de NiMo, posible gracias a que desde el punto de vista del usuario el código es a la vez el código y el estado de la computación.

### 3. El prototipo NiMoAGG

El punto de partida para el diseño del IDE es el prototipo desarrollado en AGG[1]. AGG es un sistema de transformación de grafos que dada una gramática de grafos permite visualizar paso a paso la secuencia de transformaciones de un grafo inicial dado. La gramática desarrollada tiene dos componentes: la gramática de construcción, que es un editor dirigido por sintaxis-semántica de tipos, y la gramática de ejecución que actúa como un intérprete. Dado que un programa NiMo puede ejecutarse antes de ser completado, ambas componentes se utilizan en la transformación del grafo.

El sistema desarrollado incluye tratamiento de orden superior completo, aplicación parcial, inferencia de tipos con polimorfismo, y evaluación perezosa,

lo que implicó modelar en términos de grafos los elementos y la operatoria de un lenguaje funcional perezoso. El prototipo demostró la factibilidad del enfoque y su potencialidad como banco de transformación y prueba de programas. Asimismo, dio como subproducto una especificación (como gramática de grafos) del lenguaje en su totalidad, es decir una semántica operacional completa, ejecutable en el sistema de transformación de grafos.

Sin embargo, la visualización de la secuencia de ejecución que produce NiMoAGG es muy deficiente pues la transformación no respeta el diseño gráfico inicial de la red. Esto se debe a que no se pueden establecer restricciones sobre la ubicación relativa de los nodos ni la disposición de los arcos. Incluso en la ejecución paso a paso las facilidades para reordenarlo son muy elementales, lo que lo hace poco útil como herramienta de experimentación.

Una opción habría sido mejorar en la medida de lo posible algunos aspectos de visualización interviniendo la interfase gráfica de AGG. Y en cuanto a la eficiencia, modificar el algoritmo de morfismo de correspondencia para explotar las propiedades de localidad de las transformaciones que tiene la gramática de ejecución. Sin embargo el resultado habría seguido siendo insatisfactorio como producto final. Los sistemas de transformación de grafos no están pensados para programar sistemas muy complejos y con una dinámica de transformación tan alta, se emplean sobre todo para modelización.

La otra opción es desarrollar el sistema completo en otro marco, ya sin el soporte del entorno y el motor de transformación de grafos, y centrando el foco del diseño en la visualización e interfase con el usuario.

#### **4. El IDE requerido**

El entorno integrado de desarrollo ha de mantener la funcionalidad completa de NiMoAGG, es decir: proveer un editor de grafos con comprobación e inferencia de tipos, un intérprete al estilo de los lenguajes funcionales perezosos pero gráfico, que produzca una visualización paso a paso de cada transformación, y un entorno en que se puedan alternar la edición y ejecución sin perder el estado. Pero además debe conservar lo mejor posible la estructura del grafo que va siendo transformado en ejecución.

El grafo del programa NiMo debe ser dibujado en un espacio plano, manteniendo cerca las entidades que tienen mayor relación y evitando los cruces complejos de arcos. Este es un problema bien conocido en el mundo de los grafos que no tiene fácil solución, al que se añade que los arcos NiMo representan parámetros de diferente índole, y por lo tanto tienen un orden y convenciones de horizontalidad o verticalidad y ortogonalidad en los quiebres.

El editor ha de asistir en la construcción de grafos con esas restricciones sin que deban acomodarse a mano nodos y arcos como en NiMoAGG, pero admitiendo que se haga. Debe facilitar el acceso a procesos básicos y redes predefinidas, evitar conexiones incorrectas, e inferir incrementalmente el tipo sin que el usuario deba declararlo, pero comprobándolo en caso de que lo haga.

En ejecución hay que conseguir que las sucesivas transformaciones deformen lo menos posible la disposición inicial de la red. En el caso de redes estáticas, como la de Fibonacci, no es demasiado complicado lograrlo, pero en las dinámicas con una expansión fuerte de procesos el crecimiento del grafo es muy rápido. Para poder manejarlo, además de las facilidades de visualización de ventanas (*scaling, zooming, scrolling*, múltiples vistas) imprescindibles para tratar con gráficos que crecen mucho [2], se deben proporcionar otras herramientas de control sobre la visualización de la red, como subredes que puedan compactarse o no expandirse en ejecución, ocultamiento opcional de nodos de tipo, plegado de parámetros, o visores de canal.

Por otro lado, en la puesta a punto es fundamental poder deshacer los pasos de ejecución para corregir la red y retomar la ejecución directamente. En AGG la operación de deshacer transiciones sólo permite volver al inicio o al paso anterior.

La funcionalidad de banco de prueba debería cerrarse también con la traducción automática a un lenguaje textual, Haskell en particular. Y el traductor inverso permitiría usar a NiMo como trazador en línea y depurador de programas Haskell. Además, un programa NiMo en desarrollo podría importar procesos ya programados en Haskell evitando su reedición gráfica, y también la de funciones sobre tipos escalares, cuya visualización expandida como subred no tiene ningún interés y sería más cómodo definirlas textualmente.

Por otra parte, con mecanismos de conversión entre ambas versiones NiMo se podrían usar los programas ya escritos para NiMoAGG, y contrastar estados intermedios de las transformaciones en la nueva versión con los que produciría NiMoAGG. La conversión entre ambas implementaciones no supone sortear una distancia excesiva, en cambio la conversión entre redes NiMo y programas Haskell sí plantea una problemática de traducción importante, al tratarse de un lenguaje textual funcional puro y otro gráfico funcional-dataflow como se discute en la sección 6.

Finalmente, para la escalabilidad de las aplicaciones el entorno debe proveer facilidades para la organización de librerías y proyectos.

La complejidad del sistema a desarrollar requiere una construcción incremental. El diseño del núcleo debe contemplar las extensiones futuras necesarias para lograr la calidad de producto final. NiMoToons es el núcleo del IDE cuyo diseño se presenta a continuación.

## 5. Diseño de NiMoToons

El diseño de NiMoToons [10] se centra en los aspectos de interfaz con el usuario y visualización, los que en NiMoAGG ya venían totalmente determinados por el entorno AGG. El paradigma más usado para implementar entornos visuales es indudablemente el de OO. Existen entornos de desarrollo muy extendidos como Eclipse o Visual Studio que proveen una gran variedad de herramientas.

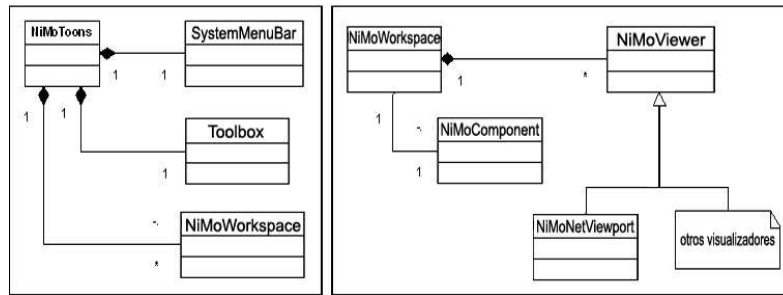


Figura 2. Diagrama de clases de NiMoToons y NiMoWorkspace

Por otro lado resulta interesante contrastar el modelado gráfico de los componentes y la operatoria de un lenguaje funcional-dataflow en paradigmas tan diferentes como lo son el de OO y el de la programación basada en reglas. En NiMoAGG todo se modeló con grafos, incluso su propio código es un conjunto de reglas que son pares de grafos. El modelo de referencia en NiMoToons es la visión de un programa en construcción o en ejecución, no como un grafo que se transforma, sino como un conjunto de objetos gráficos que interactúan.

NiMoToons sigue un diseño OO basado en el uso de patrones de diseño [6] y especialmente en el de composición (Composite), que permite naturalmente la construcción incremental de redes. Los elementos constitutivos de las redes son unidades con puertos de conexión tipados. El modelo de ejecución basado en la capacidad de actuar de cada unidad, y el modelado de cada acción dentro del sistema como un objeto permite deshacer cualquier operación, tanto de edición como de ejecución.

La funcionalidad de NiMoToons debe cubrir cuatro aspectos. Describimos a continuación los elementos del diseño más directamente involucrados en cada uno de ellos.

**Interfaz con el usuario:** La visión más externa del sistema se refleja en el diagrama de clases de la figura 2. La barra de menú permite habilitar distintos espacios de trabajo (*workspace*), importar redes, almacenarlas, acceder a la ayuda y salir del sistema. El *toolbox* se usa para editar redes importando a un espacio de trabajo procesos predefinidos, o *templates* para construir valores, elementos de canal, duplicadores, o procesos nuevos. En el diseño del *toolbox* se utilizó el patrón Prototype.

Los *workspaces* siguen un patrón MVC (Model-view-controller). Cada uno de ellos está asociado a una red (componente de primer nivel), y a una cantidad variable de *netviewports* para visualizarla en una o más ventanas. Los restantes visualizadores exhiben propiedades o información relativa a la componente. El usuario interactúa con la red en algún *netviewport*. Si la acción afecta a la red (y no sólo a su visualización local en el *netviewport* como por ejemplo un *zooming*), se refleja en todos los visualizadores de ese espacio de trabajo. La figura 3 muestra una pantalla con dos *workspaces* (Component1 y Component2), el primero visualizable en dos *netviewports* (Component1-1

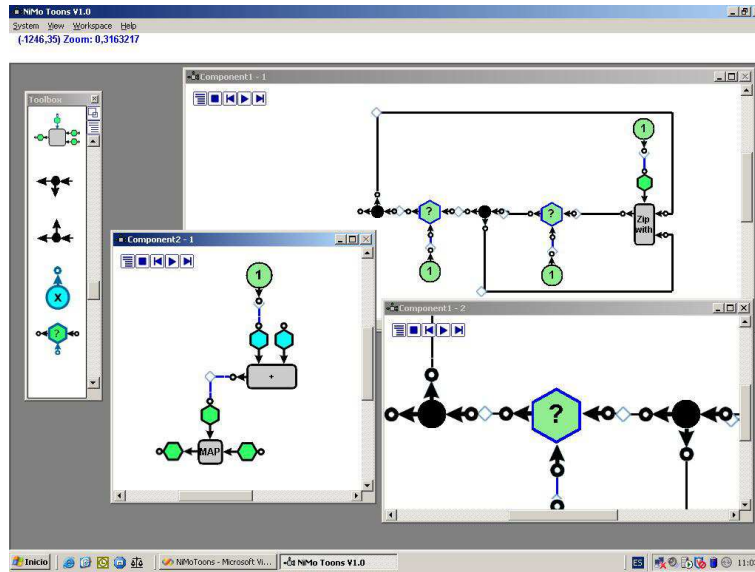


Figura 3. Pantalla de NiMoToons en su primera versión

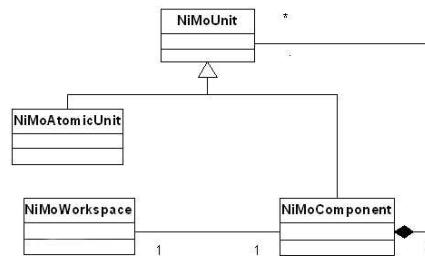


Figura 4. Diagrama simplificado de NiMoUnits

y Component1-2).

**Construcción y modificación de redes:** Una red NiMo se modela con las clases *unit* y *edge*, que representan respectivamente unidades con puertos de conexión tipados y conexiones con estado entre unidades. Las subclases de *unit* implementan entidades que son partes indivisibles de una red NiMo tales como interfaces de procesos, valores, elementos de canal, etc. *component* es también una subclase de *unit* que corresponde a los procesos que no son básicos, es decir una red NiMo encapsulada que se usa como una unidad. Existe una correspondencia entre las entradas-salidas de la red y las de la interfase que la representa.

La red que construye o ejecuta el usuario en su espacio de trabajo es a su vez una componente. Dado que las componentes son y contienen unidades se ha usado el patrón Composite. Una versión simplificada del correspondiente diagrama de clases se muestra en la figura 4.

Para construir una red se importan las unidades del *toolbox* al espacio de trabajo, y se las conecta mediante clics de ratón en los puertos de conexión de ambos nodos. Si la conexión es válida según el tipo se genera el arco con el diamante de estado que corresponda. Para implementar la comprobación e

inferencia de tipos se modela la estructura de tipos basándose también en el patrón de composición (un tipo puede contener tipos).

**Visualización:** Para organizar la disposición espacial de la red se usa la noción de *boundingBox* (BB), que representa el espacio rectangular que ocupa un elemento o conjunto de elementos. Cada unidad, componente y arco tiene un BB. La disposición en la red debe evitar que los BB se intersequen, a excepción de los arcos que sí podrían cruzarse, por lo que en el momento de expandir un proceso-red el BB de la componente determina los desplazamientos necesarios para evitar intersecciones de la nueva subred con los elementos vecinos. Inversamente el BB de la interfase del proceso red puede utilizarse para la visualización compactada de subredes. El algoritmo de reubicación gráfica aun está en consideración. El diseño contempla su encapsulado en una clase propia con el objetivo de poder seleccionarlo en tiempo de ejecución.

**Ejecución y puesta a punto:** Las unidades son entidades independientes capaces de actuar cuando sea posible. La ejecución “simultánea” del conjunto de acciones que puedan realizar todos los procesos en condiciones de actuar dará como resultado un paso de ejecución.

En NiMo la construcción y ejecución es un continuo ya que la intervención del usuario permite cambiar de uno a otro modo sin perder el estado. Los modos de ejecución se establecen a través del menú del *workspace* de la componente (iconos de: paso a paso, detener, lanzar, deshacer). Cada transformación que sufre la red se puede deshacer. Desde crear una unidad, borrarla, moverla hasta cambiar algún color en edición, o ejecutar todos los procesos que pueden actuar en un paso. El sistema apilará objetos que representan las acciones realizadas. La operación de deshacer un paso de ejecución o una operación de edición cancelará estas acciones, representadas con objetos Command que guardan el estado de los objetos afectados usando el patrón Memento.

## 6. Conversión entre NiMoToons y NiMoAGG y traducción a Haskell

En la implementación de los conversores y traductores [7] se ha utilizado una representación intermedia (RI) como elemento central. Ésta es una representación de grafos que sirve como origen o destino intermedio de todas las traducciones realizadas, y simplificaría el añadido de futuras traducciones (versiones sucesivas de NiMoToons y traducciones a otros lenguajes).

AGG exporta sus grafos en GXL, que es un formato de intercambio de grafos basado en XML. Dado que XML es un estándar para el que existen muchas herramientas se lo utilizó también como formato de importación/exportación entre la RI y sus restantes orígenes/destinos.

**Conversión entre versiones NiMo:** el paso de una a otra versión requiere algunas transformaciones de sub grafos en nodos o en arcos con estado y viceversa ya que algunas unidades NiMoToons son subgrafos en NiMoAGG.

Al pasar de NiMoAGG a **RI** se utiliza SAX [11] para generar una primera traducción a partir del fichero GXL. Ésta es luego transformada a la **RI**, haciendo básicamente una compresión de ternas arco-diamante-arco en arcos con estado. La conversión de **RI** a AGG se genera de forma casi automática, utilizando DOM [5] para la traducción a GXL, previa conversión inversa de los arcos con estado.

La conversión de **RI** a NiMoToons es casi directa también, sólo supone alguna transformación de subgrafos en nodos (interfase de procesos) al realizar el vuelco de **RI** a XML. Este grafo descrito en XML deberá ser importado por NiMoToons. La operación inversa es similar, pero desagregando subgrafos.

**Traducción entre redes NiMo y programas Haskell:** Aquí las diferencias entre lenguajes son mucho más significativas. En NiMo no se requieren identificadores para los arcos. Para traducir a Haskell se deben generar identificadores para los parámetros de procesos y definiciones locales para los arcos duplicados. Por otro lado, NiMo tiene procesos básicos con múltiples salidas, la traducción deberá usar tuplas o definiciones auxiliares de tantas funciones como salidas. Por ejemplo el proceso que separa un canal en dos subcanales de acuerdo a una condición, se traducirá usando dos funciones **filter**.

En cuanto al camino inverso, por el momento se han fijado algunas restricciones sobre los programas Haskell, limitándose a los que no requieren una transformación previa (por ejemplo no pueden traducirse directamente los que emplean tipos definidos por el usuario o patrones sobre naturales).

Para la traducción de Haskell a **RI** se utilizó el código abierto de GHC [8]. A partir del árbol sintáctico abstracto (AST) generado por el parser se intervino la función de pretty-printing para producir ficheros XML. El resultado es una primera aproximación a la traducción. Con el uso de DOM se producen las transformaciones necesarias para llevarla a la **RI**.

Para traducir de **RI** a Haskell, como una red NiMo es un conjunto de árboles con back edges, su traducción al AST es relativamente directa. El etiquetado de los arcos del grafo en **RI** permite la generación de los identificadores de variables necesarios para el código Haskell. La aplicación está programada en Java.

## 7. Estado actual

En la actualidad se dispone de una primera versión (implementada usando tecnología .NET) para la parte de construcción de redes de NiMoToons con la estructura descrita en la figura 2. El usuario puede construir varios programas NiMo usando espacios de trabajo diferentes en la misma pantalla, tener visiones diferentes desde los distintos *netviewports* y deshacer las operaciones correspondientes al *workingspace*. Están implementadas las operaciones que permiten traer unidades desde el *toolbar*, conectarlas con una primera comprobación de tipos y reordenar manualmente la red. Cuando un nodo se desplaza los arcos respectivos mantienen su disposición si ello es posible, o se producen

los quiebres ortogonales mínimos para mantener las convenciones de horizontalidad y verticalidad, evitando intersecciones con los nodos. Las traducciones entre ambas versiones NiMo y de éstas a Haskell están acabadas, y también la inversa con las limitaciones mencionadas.

Actualmente se está trabajando en la incorporación a NiMoToons de los mecanismos de traducción desarrollados, en la experimentación de la ergonomía del editor de redes, y en los algoritmos de visualización para reordenar automáticamente la red en edición y durante las transformaciones de ejecución. Simultáneamente se están desarrollando las operaciones de ejecución de redes y su inclusión entre las acciones que pueden deshacerse. El otro aspecto actualmente en desarrollo es la deducción completa de tipos.

El trabajo futuro estará enfocado a consolidar el IDE en lo que respecta a la escalabilidad implementando las opciones de visualización compactada, a desarrollar librerías de redes útiles, ampliar el conjunto de programas Haskell traducibles y, en general, conseguir que el IDE alcance el estado de producto final.

## Referencias

- [1] AGG home page, AGG Home page: <http://tfs.cs.tu-berlin.de/agg/>
- [2] Burnett M., Baker M.J., Bohus C., Carlson P., Yang S., y Van Zee P. *Scaling up visual Programming languages*. *Computer*, 28:45–54, 1995.
- [3] Clerici S., Zoltan C. *A Graphic Functional-Dataflow Language*. Trends in Functional Programming, Volume 5. Editor Loidl H-W. Intellect 2006.
- [4] Clerici, S., Zoltan C. *Página del Proyecto NiMo*. <http://www.lsi.upc.edu/~nimo/Project/>
- [5] DOM home page <http://www.w3.org/DOM/>
- [6] Gamma E, Helm R, Johnson R, Vlissides J, Design Patterns Addison -Wessley Professional Computing Series, 1995
- [7] García Sanjulián, J. *Conversión de NiMo a Haskell y viceversa*. Proyecto Final de carrera FIB-UPC. 2006.
- [8] Glasgow Haskell Compiler <http://www.haskell.org/ghc/documentation.html>
- [9] Haskell home page <http://www.haskell.org/>
- [10] Pestigiacomio, G., *NiMoToons, el núcleo de un sistema integral para NiMo*, Tesis de licenciatura en Cs de la computación UBA, convenio FIB-UPC-UBA. 2006.
- [11] SAX home page: URL: <http://www.saxproject.org/>