Visual Programming

VISHNU Specifications

Jean-Sébastien Mouret

May 30, 2002

[Episode One]

An open-minded eye-candy visual programming environment.

In need of a simple but efficient environment to combine Olena's image processing algorithms, we present a new visual programming framework based on graph grammar capable of representing and executing data-flow diagrams in an interactive manner.

After a survey of visual languages evolution over the past ten years, we describe the complete specification of our framework, comparing its features to other related works, and giving concrete case studies of common problems.

Laboratoire de Recherche et Développement de l'Epita
14-16, rue Voltaire – F-94276 Le Kremlin-Bicêtre cedex – France
Tél. +33 1 44 08 01 01 – Fax. +33 1 44 08 01 99
lrde@epita.fr – http://www.lrde.epita.fr

Contents

Chapter 1

State of the art

The purpose of this survey is twofold. First, it is to feel our software needs and how a visual representation could help doing the job. Second, it is a question of forming the eye of the reader for a better understanding of the various notation systems.

1.1 Applications

As the notion of visual programming has a broad scope, examples of visual programming can be seen in numerous applications. Here follows a bench of common applications to give a feeling of where the real needs are, and what we expect from this kind of software.

1.1.1 Reason

Reason music system by Propellerhead Software [15] is made of rackable virtual instruments emulating machines such as analog synthetiser, sampler, drum machine, mixer, effects, pattern sequencer, and so on. When the user creates a new instrument, the program connects it intuitively to the other instruments. But if this setup is not satisfying, one can flip the instruments' panel and *rewire* the music system. This ability is extremely enjoyable for the user, and is one of our needs. Note that if the user wires an instrument incorrectly, this can induce side effects on its outputs. This behavioral similarity with a real machine is quite droll.

Figure 1.1: Reason's front and back panels. The user can use on-screen patch cords to set up complex routings and cross-device modulation patches.

1.1.2 Virtual Waves

Virtual Waves by Synoptic [16] is a sound synthetiser. One can create any sound by plugging signal filters together. Every possible effect is represented by a box with plugs. Each box can be configured with a nice floating panel.

The data-flow graph is made of three kinds of nodes : generators that produces signal, filters that do treatments, and effectors that output the signal on various devices.

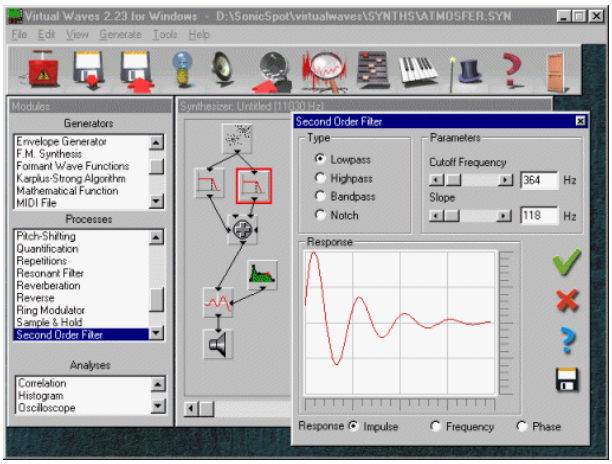

Figure 1.2: Virtual Waves by Synoptic, introducing configurable treatments and graph grammar.

1.1.3 Reaktor

Reaktor by Native Instruments [10] is a complete music synthetiser. Like Virtual waves, Reaktor describes its treatments with a graph, but adds the notion of composite nodes. A treatment can encapsulate a complete treatment graph defining its behavior. Such a treatment is called composite, and the subgraph can also be made of composite treatment. Thus the user can navigate inside the whole synthetiser system by going more or less deep in the underlying treatment hierarchy.

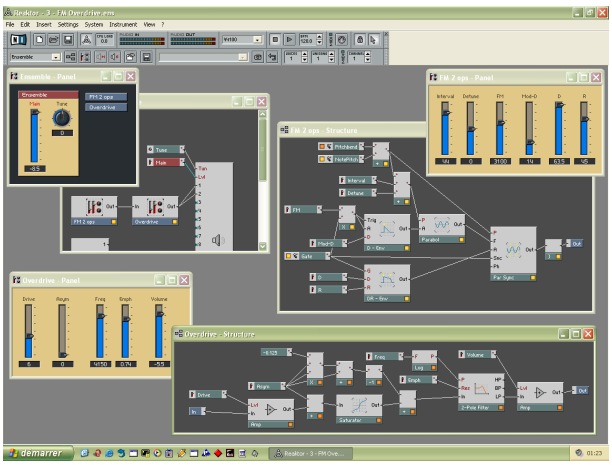

Figure 1.3: Reaktor by Native Instruments, introducing composite treatments.

1.1.4 GraphEdit

GraphEdit is contained in the Microsoft DirectX 8 SDK [3]. It is made to combine audio/video codecs compressor/decompressor and/or converters. This is just to illustrate the power of graph grammar when used to represent composition of a great diversity of components.

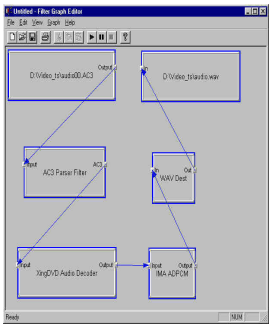

Figure 1.4: GraphEdit by Microsoft, codecs mixer.

1.1.5 LabView

LabView by National Instruments [9] is measurement and automation software. It represents acquisition devices as virtual instruments that can be linked to software processes. Like Reaktor, Labview can encapsulate several treatments in a single box.

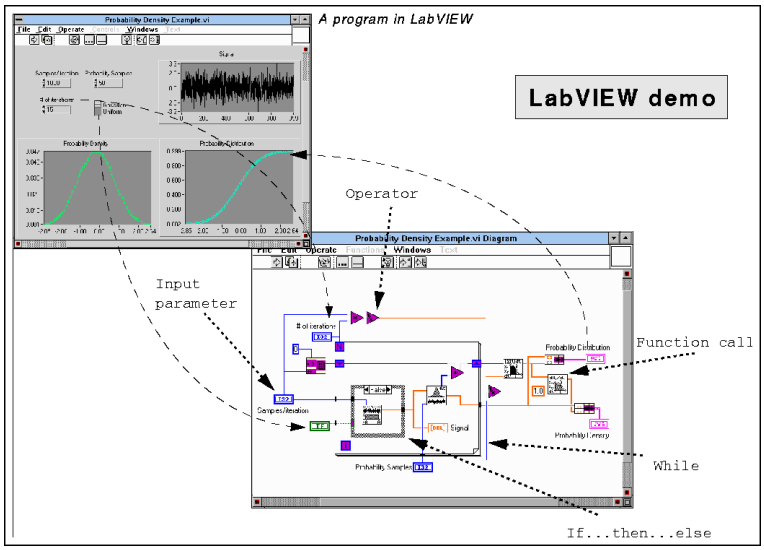

Figure 1.5: LabView by Native Instruments, a complete development system.

In addition to the fact that LabView has huge collection of treatments in multiple fields, multiple data controllers and generators, LabView appends particular nodes dedicated to symbolize control structures such as if, for or while expression, and sequences [1]. This is incredibly useful and makes LabView a complete programming language. Advanced use of these features tends to have strong cognitive effect on visual programming representations and their programmers [20].

1.2 Visual Languages

This is a quick, far from exhaustive, survey of visual languages over the past two decades. Most of the presented languages are discussed in [5] which is a good starting point to understand this field.

1.2.1 Definitions

To better understand visual languages, here are a few preliminary definitions.

Visual Language

- *icon (generalized icon)*
  An object with the dual representation of a logical part (the meaning) and a physical part (the image).
- *iconic system*
  A structured set of related icons.
- *iconic sentence (visual sentence)*
  A spatial arrangement of icons from iconic system.
- *visual language*
  A set of iconic sentences constructed with given syntax and semantics.
- *syntactic analysis (spatial parsing)*
  An analysis of an iconic sentence to determine the underlying structure.
- *semantic analysis (spatial interpretation)*
  An analysis of an iconic sentence to determine the underlying meaning.

Visual Programming Paradigms

Here follows a description of four visual programming paradigms extracted from `http://cbl.leeds.ac.uk/nikos/tex2html/examples/concepts/node35.html`.

- *Data-flow Paradigm*

  Here a program is composed of functional modules, with connecting paths between inputs and outputs. In textual languages, data-flow diagrams are drawn as part of the program design process and then translated into text. Visual languages omit the translation step.

- *Constraint-based paradigms*

  A constraint may affect many variables, which in turn may affect more. This kind of interaction can benefit from a diagrammatic representation. ThingLab is an example of such a language where a set of constraints (rules) describe the invariant properties and relationships of all objects in a problem space. The solutions is the set of values that satisfy all the constraints simultaneously. This approach resembles logic programming.

- *Programming-by-demonstration*

  Programming is done by graphically manipulating the data on the screen, demonstrating to the computer what the program should do. [...] Examples of this programming style are ThinkPad, Rehearsal World [Later influencing AUDITION], and PT (Pictorial Transformation). The latter is a procedural language using a film-making metaphor. [See also Marquise in the Garnet Toolkit.] [See also Figure 1.6]

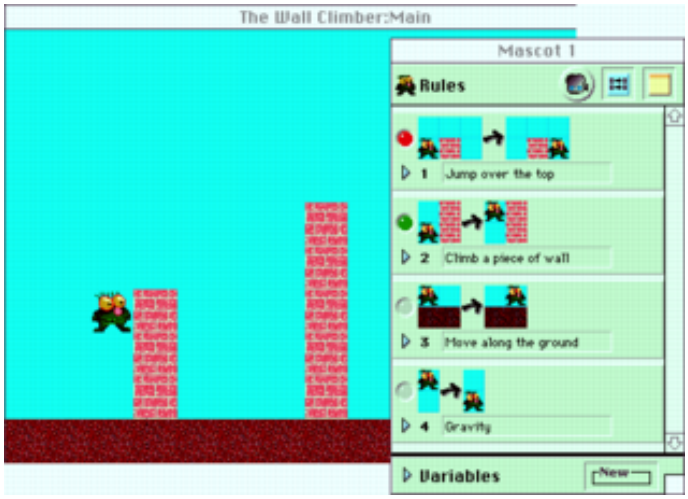

Figure 1.6: Project Cocoa [2], a programming by demonstration VPL for kids.

- *Form-based paradigms*

  This programming style can be thought of as a generalization of spreadsheet programming. [...] The visual representation of a cell matrix allows the omission of the concepts of variables, declarations, and output formatting. [...] Forms extends the spreadsheet paradigm. The basic 'sheet' is a form on which the user can place cells called objects. A cell expression can reference any cell (or cells) in any object within the containing form. Subforms can be used to implement some kind of inheritance.

Liveness

Apart from the fact that data-flow models can be either data driven in which case nodes fire as soon as all inputs become available or demand driven in which case nodes respond only to explicit firing requests; several firing strategies exist to update the runtime visualization of a program. In [17] Tanimoto proposes a distinction between four liveness levels :

- *informative* : just a visual representation, no semantic feedback is provided to the user.
- *significant* : execution on demand, feedback is not provided automatically.
- *responsive* : automatic re-execution on re-edit.
- *live* : continuous execution.

A discussion on the differences between *live* and non-live visualization strategies for program debugging in [7] declares *live* to be the best choice.

1.2.2 ARK

ARK (Alternate Reality Kit) created at Xerox PARC is known to be the *more unique and visionary domain-specific VPLs*. It is an animated environment for creating interactive simulation. A user can manipulate physical objects, like balls and blocks, having their own properties, like mass and velocity. A user can add interactors in the environment to represent physical laws that will interact with these objects.

ARK is made with Smalltalk-80. All of the objects in the underlying Smalltalk environment are available to the ARK programmer. Objects which are not ARK-specific appear as representative objects which can be linked with ARK objects the same way as native objects.

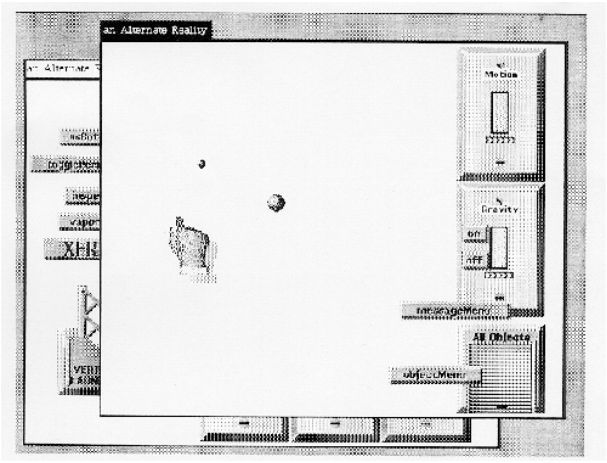

Figure 1.7: ARK, 1986.

1.2.3 Prograph

Prograph is considered to be the most successful of the general-purpose visual languages. Prograph is a visual object-oriented language. It provides simple inheritance, classes with methods, exception handling, persistent objects. Each method is defined with a series of case which contains a dataflow diagram. A method call consist in executing every method's cases, stopping prematurely if special controls like *terminate-on-success* or *failure* are encountered. This permits explicit control over evaluation order.

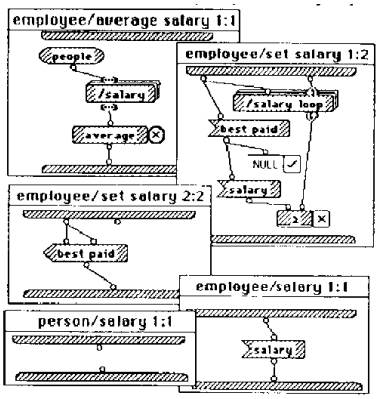

Figure 1.8: Prograph, 1988.

1.2.4 MET++

MET++ [19] is a visual programming framework using data-flow graphs. As recent softwares, it proposed a full node library with gui components and so on. One of Its particularities is to offer bidirectional connections between data units. This implies bidirectional behavior of treatment units. Figure 1.9 shows a simple Fahrenheit to Celsius converter which uses this functionality. MET++ has another particularity. Access to different kinds of containers such as vectors or 2d/3d images is made via a data mapper. Data mapped units provide a *void* port which, when

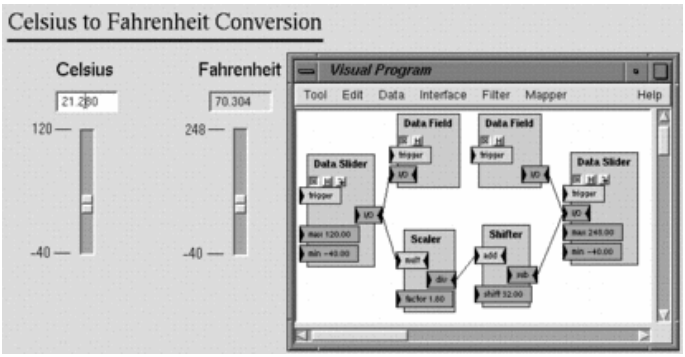

Figure 1.9: MET++, 1996. A Fahrenheit/Celsius converter, using bidirectional connections.

connected to a container unit's index port, dynamically creates an *iterator* port to map the linked data container. This procedure is shown in Figure 1.10.

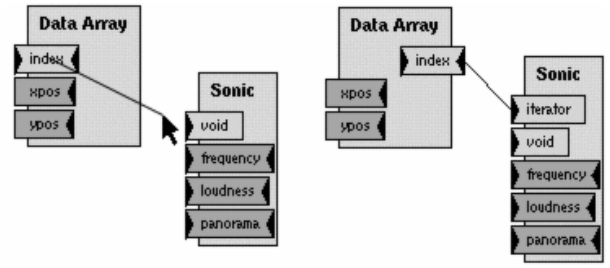

Figure 1.10: Connecting a mapper: dynamic generation of *iterator ports*.

1.2.5 VIPR

VIPR (Visual Imperative PRogramming) represents a unique approach to completely visual general purpose programming. It uses nested series of concentric rings to visualize programs. Figure 1.11 shows how VIPR represents simple function calls sequence from a static and dynamic point of view.

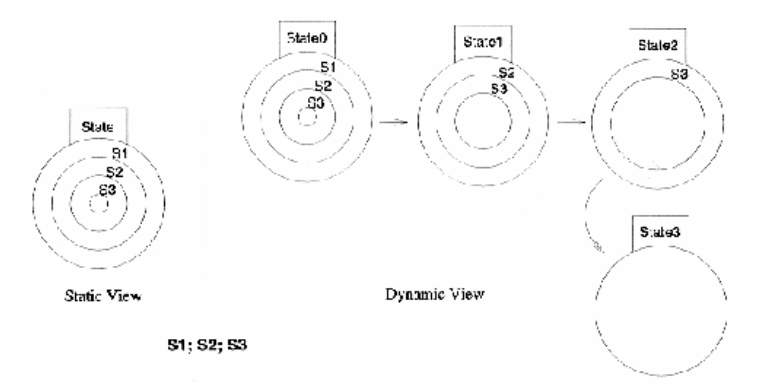

Figure 1.11: VIPR, 1994. Visualization of program execution.

Figure 1.12 shows a function definition and call. Function arguments are symbolized by rings on the upper-right corner, and their return values on the lower-right. VIPR also defines control structures like *if* and *while* with a comparable style.

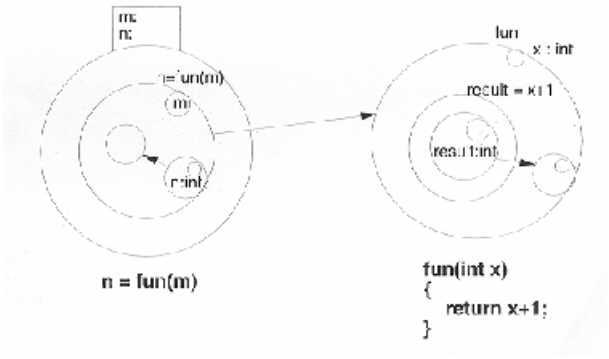

Figure 1.12: Function definition and call in VIPR.

The VIPR group has also developed a visual representation for the lambda calculus which they refer to as VEX for Visual EXpressions.

1.2.6 Cube

Cube [12] is a visual programming system with three-dimensional representation. The first interest of having a three-dimensional representation is the ability to display more information than a traditional two-dimensional representation, and with an easier user interaction. Cube was conceived to evolve in a virtual-reality based programming environment.

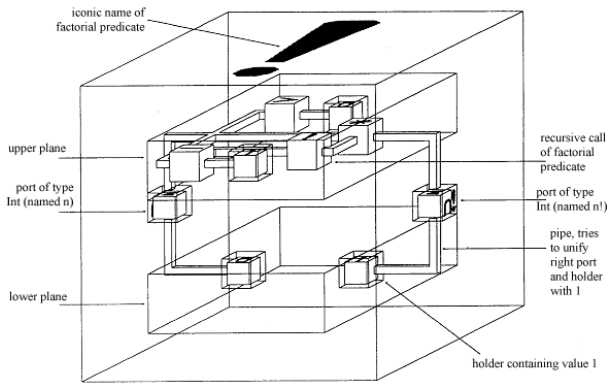

Figure 1.13: Cube, 1991.

1.2.7 3D-PP & PrologSpace

3D-PP is another visual programming system with three-dimensional representation.

> "3D-PP is based on the concurrent logic programming language GHC (Guarded Horn Clauses) which is one the high level declarative languages. A declarative programming language is suitable to be visualized by a visual programming system because visual programming is also declarative. A logic programming language is also

suitable to be visualized because a logic programming language requires comparatively fewer number of programming elements than a procedural language." [13]

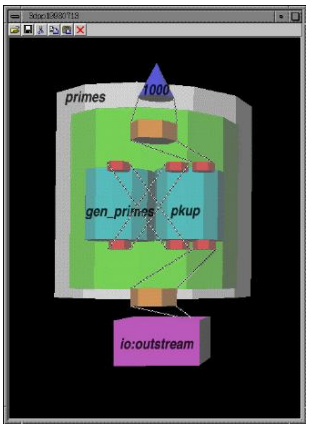

Figure 1.14: 3D-PP, 1999.

The 3D-PP approach is very similar to PrologSpace [21]. PrologSpace adopts three-dimensional program representation based on the logic programming language Prolog. PrologSpace is built on top of VisualProlog which is a version of Prolog that provides support for X windows, widget, three-dimensional graphics, animation and audio.

1.2.8 Visual Haskell

Visual Haskell [14] is an attempt to define a visual language with a syntax as close as possible to the original textual language. To visualize an Haskell program, the source code is first translated to an intermediate form which is then directly translated into visual representation. The goal is make a *two-view* programming system with both visual and textual representations.

```
map          :: (a -> b) -> [a] -> [b]
map f []      = []
map f (x:xs) = f x : map f xs
```

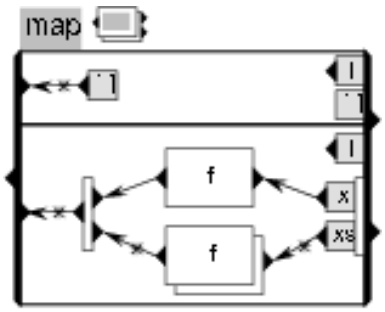

Figure 1.15: Visual Haskell, 1994.

1.2.9 GenGED

GenGED [4] is a generic graphical editor for visual languages based on algebraic graph grammar. Giving a visual alphabet and a grammar, GenGED generates a graphical editor for this language.

GenGED serves as the basis for a more formal research in visual languages. For example, in [8] Ermel & Bardohl use it to visually design a software's architecture and its evolution by defining graph transformation visual rules to formalize the system evolution.

Chapter 2

Welcome to VISHNU

Vishnu, preliminary acronym for Visual Interface for Simple Human Newbie User, is a visual programming framework made to design and run programs at the same time. Vishnu is a mixture of features seen in the previously cited softwares and languages. Here is a first description of this environment.

2.1 Overview

As a starter, figure 2.1 shows an hello world application to comply with language presentation standards. This section explains Vishnu's notation and bridges over traditional programming idioms to check the symbols' powerfulness.

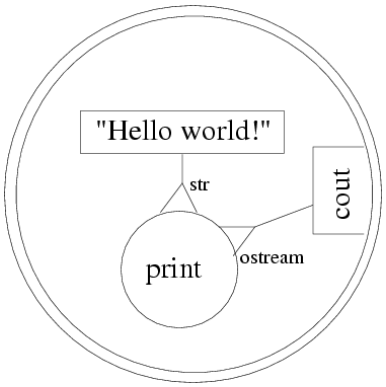

Figure 2.1: The mythical hello world.

2.1.1 Nodes

Vishnu programs are made of nodes. A node encapsulates a treatment unit and is symbolized by a circle (Fig 2.2). Nodes can have outputs (Fig 2.3) and/or inputs (Fig 2.4).

Figure 2.2: A node...

Figure 2.3: ... with outputs ...

Figure 2.4: ... and inputs.

2.1.2 Inputs & Outputs

An output represents a data owned by a node, and so has a type and a name (Fig 2.5). An input represents a possible connection to an output. Like outputs, it also has its own name and type. An output is linked to zero or many inputs, an input is connected to zero or one output (Fig 2.6).

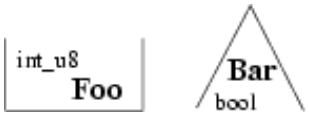

Figure 2.5: Typed output and input.

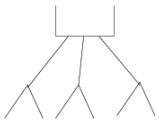

Figure 2.6: Interconnections.

2.1.3 Constants

A constant node represents a constant value of a type. Its runtime behavior is to produce once the associated constant value at the first turn after instantiation, and then keep quiet. Constants somehow have the effect of *booting* a graph by introducing a value in the data flow.

Because of their simpleness, their visual representation is condensed into a single square.

Figure 2.7: Constant node representation.

Note that with these few icons, we can already create basic data-flow diagrams.

2.1.4 Streams

We want no cycle in node graphs because it leads to ambiguous scheduling order. However, to refer to the past value of an output, which is what we mean when intuitively linking an node's input to one of its outputs, we introduce a *stream* notation for outputs (Fig 2.8). That way, inputs linked to an output past value can be discarded of the scheduling algorithm.

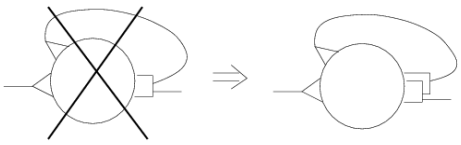

Figure 2.8: Explicit cycles and streamed output notation.

A nice example of stream use is a Fibonacci number generator (Fig 2.9). It also introduces the way to initialize past stream values by linking it to a *constant* node.

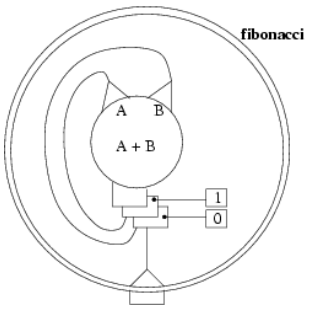

Figure 2.9: Fibonacci numbers generator using streamed output.

2.1.5 Membranes

Membranes are tissue layers that line node's organs. A node can contain several membranes. A membrane holds a nodes graph, and has inputs and outputs. A node owning a membrane programmatically controls the membrane nodes graph execution. It means that when a node is computed, it can decide to run zero or many times its membranes. For example, figure 2.11 shows a node with a single membrane and a boolean output called start/stop. Its behavior is to execute its membrane only if the boolean output is true.

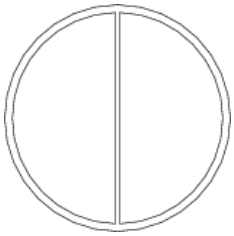

Figure 2.10: A node with two membranes.

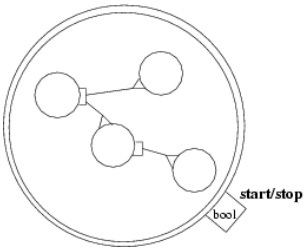

Figure 2.11: Start/Stop node.

Membranes also permit procedural abstraction. The *basic* node has one membrane, and its behavior is simply to execute the membrane's graph. With such a node, a user can encapsulate a composite treatment in a single node.

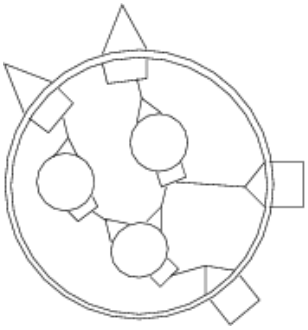

Figure 2.12: Procedural abstraction with a *basic* node.

On figure 2.12, we notice that a node input (resp. output) faces a membrane's output (resp. input). This construction emphasizes the propagation of the data inside the node (resp. outside). Implementation details of such implicit links will be discussed later.

2.1.6 Meta-Membranes

Keep in mind that what we visualize is both program structure and execution. Therefore every nodes shown before are *living* nodes. Even if a membrane's node graph execution is controlled by its parent node, we can consider that nodes in a membrane are somehow persistent. To declare a graph of not yet living nodes, we introduce meta-membranes (Fig 2.13), represented by dotted lines.

Instead of having a graph of node instances, meta-membranes contain a graph of node classes. A node owning a meta-membrane can instantiate it at will, creating regular membranes out of it. These freshly created membranes may be visualized by a stack below the mother meta-membrane.

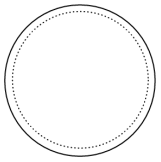

Figure 2.13: Node with a meta-membrane.

With this notation, we can create an *if* node. Depending on the value of a boolean stream formerly named “condition”, the *if* node instantiates the true of false meta-membrane. Figure 2.14 gives an example of such a construct. If we had used regular membranes instead of meta-membranes in this example, the behavior would have been completely different. Nodes contained in regular membrane are created (resp. destroyed) when the parent node is created (resp. destroyed). For meta-membranes, their content is created on demand. Imagine figure 2.14 using a *if* with regular membranes, the execution would have produced (log "bad input") only the first time the condition is false because afterwards the *constant* node will not generate a value, thus not firing the *log* node. With a meta-membrane, the *if* node behavior is to re-instantiate a membrane from the meta-membrane every turn, thus having a fresh *constant* node ready to produce its value.

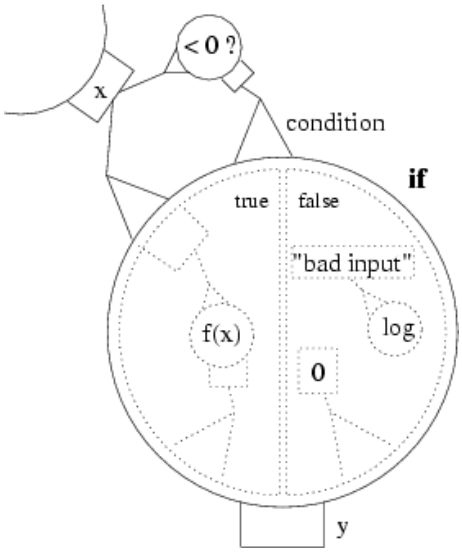

Figure 2.14: y = if x < 0 then (log "bad input"; 0) else f(x)

Using a meta-membrane, we can also create a kind of *for* node. Figure 2.15 shows a *maximum vector value* node that makes use of such a node. Its behavior is to create, at every turn, *n* membranes out of its meta-membrane, and then chain them together as shown in figure 2.16. The dynamic representation of this internal behavior will be as a stack of membranes below the mother meta-membrane.

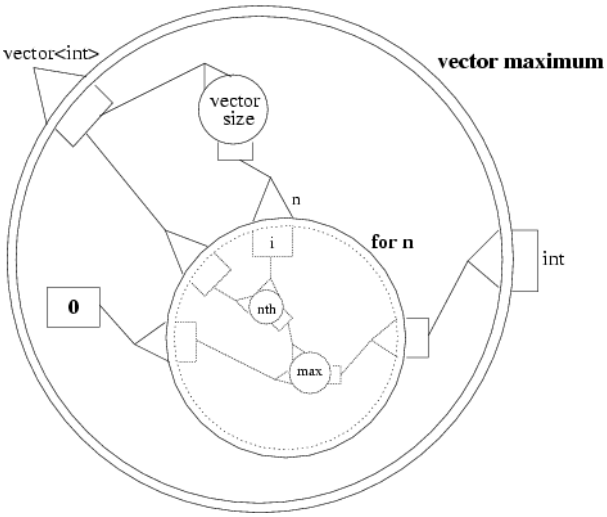

Figure 2.15: Maximum value of an integer's vector.

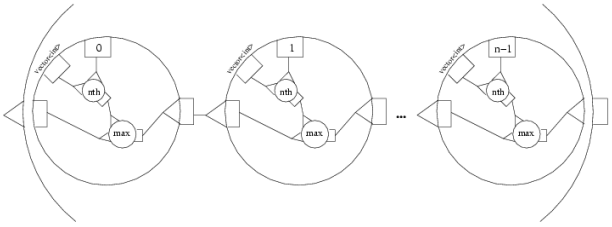

Figure 2.16: Membrane instances created programmatically during execution of the *for n* node in figure 2.15.

2.1.7 Factories

A factory is a type that provides an interface to dynamically create new inputs or outputs. Instead of representing this sort of node output as usual, an editor may symbolize it like a striped input (Fig 2.17) to add a kind of syntactic sugar for node editing. To tell an editor where to place the symbol, a factory can be associated to the outline of a node, or to any of its membranes.

Figure 2.17: Editor's syntactic sugar for factories.

When a user links an output to a factory, the editor will ask the factory to create a corresponding input, and then link it to the from output. This behavior is a bit inspired by MET++ (see 1.2.4).

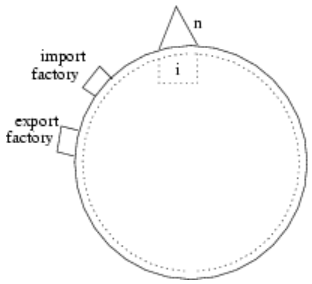

Figure 2.18: Concrete *for n* node ...

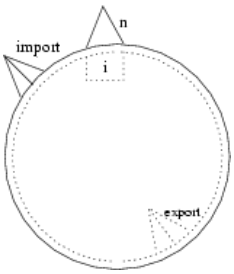

Figure 2.19: ... and its representation in an editor.

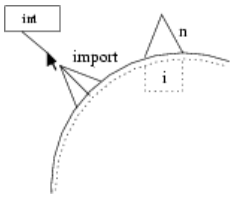

Figure 2.20: Connecting an output to a factory...

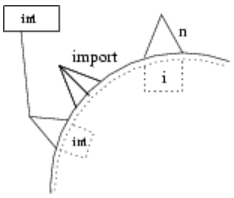

Figure 2.21: ... makes it create a new input.

What a factory creates is controlled programmatically. That way a node can offer several factories to accomplish different tasks. For example, the *for n* node proposes an import factory to simply make an outside value available inside (Fig. 2.20 & 2.21). This can be seen as membrane drilling. Another factory is made to export a value compute inside (Fig. 2.22 & 2.23). It creates all necessary inputs and outputs to initialize and propagate the value through the membranes instantiated during execution.

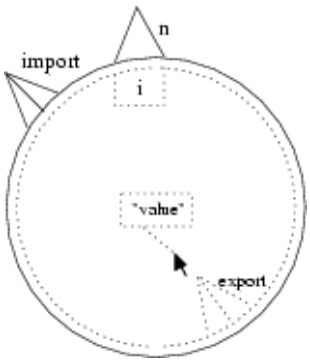

Figure 2.22: Connecting an output to a factory...

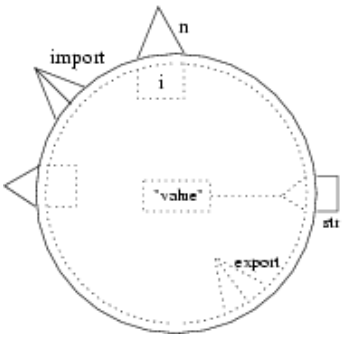

Figure 2.23: ... can create complex arrangements.

2.1.8 Recursiveness

Although this feature is not planned to be implemented in a near future, Vishnu may provide a *recurse* symbol to ease writing of naturally recursive algorithms. The *recurse* symbol is displayed with a black disk linked to a parent node. It can only be defined in meta-membranes to prevent eternal recursion at its sight.

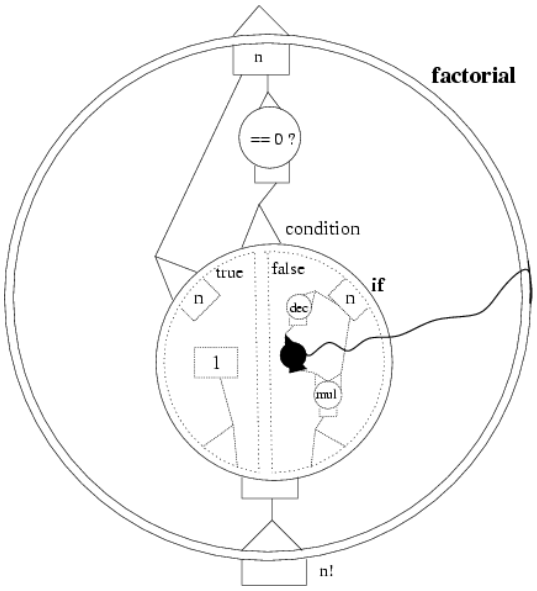

Figure 2.24: Factorial node using a *if* node and the *recurse* symbol.

2.2 How do these things work?

After this first glimpse of the framework, we try to describe a bit deeper how these things work.

2.2.1 Types & Triggers

As we have seen, nodes have typed inputs and outputs. A type definition has the following parts:

- a name,
- a data structure,
- a set of triggers,
- a set of remote triggers.

A trigger is a method of an output that calls a corresponding method in every inputs linked to the emitting output. Triggers will serve as the basis for further optimization because calls to input's methods could be inlined during *crystallization* of a node graph (see 2.3.4).

Likewise, a remote trigger is a method of node input that callbacks a method in its linked output. The difference between regular and remote triggers is that remote triggers don't directly call the corresponding output method, but push the message on the remote output's message queue. Thus a node's execution would look like this:

- 1. receive callbacks triggered by nodes linked to inputs,
- 2. receive output callbacks by processing message queues,
- 3. compute its outputs.

During each step, a node can freely access its members and fire any triggers.

2.3 Standard Framework Components

2.3.1 Multi-threading

To support multi-threading, we only need to add one new node definition. A *thread* node has a single membrane where to place nodes to be executed continuously. Its behavior will be to cache regular and remote callbacks in thread-safe message queues (roughly fig 2.26).

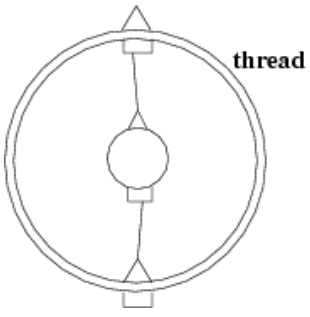

Figure 2.25: A *thread* node enclosing a single treatment.

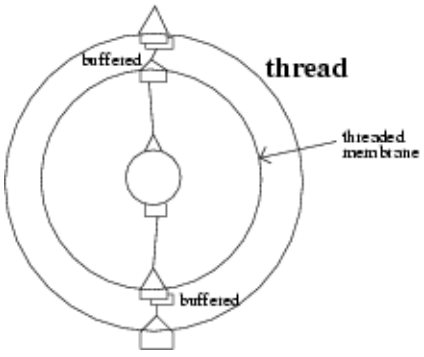

Figure 2.26: Internal thread-safe buffers (this is not a real representation).

2.3.2 Node Library

Definition

A node library contains a list of node definitions and methods to instantiate them. Like a node interface, a node library is also a type. This permits the user to dynamically see who is connected to the library.

Dictionaries

Interesting treatments on a library would be to extract "dictionaries" out of node definitions. A type dictionary could list all known types, giving for each type a list of node definitions having inputs or outputs of this type. An editor could use this kind of dictionary to implement a *right-click contextual popup menu* on a node output to offer the creation of related node. Besides, a node dictionary lists all node definitions, giving for each definition N a list of node definitions having their input types matching N's outputs. Respectively, in an editor, right-clicking on a whole node would open a *contextual popup menu* that propose to create the corresponding nodes and automatically link their inputs to the clicked node's outputs.

2.3.3 Constant makers

Because types tend to be more and more complex, creating constants for them is not a natural task. Therefore every type should offer at least one "constant maker" node capable of creating new nodes representing a constant of this type. For example an simple integer constant maker would be a dialog box with an input field to type the value and a "create" button.

Constant makers may feature two kinds of input type depending on their powerfulness. The first kind of possible input is a membrane interface input, the "create" button creates a living constant node inside the linked membrane. A maker could also have a node library input to offer a "generate" button that would produce source code for a new node definition to be then inserted in the library.

2.3.4 Crystallization

Definition

After having played with for a while with a node's subgraph, the user may choose to crystallize the node. This has the effect of generating specific code for this node. Its membranes become uneditable and its factories are removed.

Crystallizers

This technique permits strong optimizations because the node system offers a large playground to do in-depth analysis of a component graph. Crystallizers are nodes that do this kind of job. They have an node interface input connected to the node to be crystallized, and a library input to put in the generated node definition.

Chapter 3

Conclusion

After a presentation of visual languages, we have presented a new visual programming framework designed to be user friendly and extremely scalable. We have checked its abilities to express common language structures, and started to explain its internal procedure.

Further work

There is still work to do to find an optimal way to unify trigger fashioned and stream wrapped communication styles.

As this framework can support both functional (with regular triggers) and message oriented (using remote and regular triggers) design, there is much work to sketch a standard library of common algorithms, widget set, rendering engine, and so on, using for each the paradigm that fits best to offer an intuitive framework.

Bibliography

[1] Labview tutorial. www.upscale.utoronto.ca/GeneralInterest/Drummond/LabVIEW/.

[2] Project cocoa, 1999-2000. homepage.mac.com/redbird/cocoa/.

[3] Graphedit in microsoft directx 8 sdk, 2000-2001.

[4] Roswitha Bardohl. A generic graphical editor for visual languages based on algebraic graph grammars, 1998. citeseer.nj.nec.com/438230.html.

[5] Marat Boshernitsan and Michael Downes. Visual programming languages: A survey. Technical report, Berkeley, CA 94720, dec 1997. www.cs.berkeley.edu/ maratb/cs263/.

[6] Doug A. Bowman and Larry F. Hodges. Formalizing the design, evaluation, and application of interaction techniques for immersive virtual environments. *Journal of Visual Languages & Computing*, 10-1:37–53, 1999. www.idealibrary.com/links/doi/10.1006/jvlc.1998.0111.

[7] C. Cook, M. Burnett, and D. Boom. A bug's eye view of immediate visual feedback in directmanipulation programming systems, 1997. citeseer.nj.nec.com/cook97bugs.html.

[8] C. Ermel, R. Bardohl, and J. Padberg. Visual design of software architecture and evolution based on graph transformation, 2001. citeseer.nj.nec.com/ermel01visual.html.

[9] National Instruments. Labview. www.ni.com.

[10] Native Instruments. Reaktor. www.nativeinstruments.de.

[11] E. Lee and T. Parks. Dataflow process networks, 1995. citeseer.nj.nec.com/455847.html.

[12] Marc A. Najork and Simon M. Kaplan. A prototype implementation of the cube language. In *Proceedings of the 1992 IEEE Symposium on Visual Languages (VL '92)*, pages 270–272, Seattle, WA, 1992. www.research.compaq.com/SRC/personal/najork/vl92/.

[13] Takashi Oshiba and Jiro Tanaka. '3d-pp': Visual programming system with three-dimensional representation. In *International Symposium on Future Software Technology (ISFST '99)*, pages 61–66, October 1999. www.softlab.is.tsukuba.ac.jp/ ohshiba/main-e.html.

[14] H. John Reekie. Visual Haskell: A first attempt. Technical Report 94.5, PO BOX 123, Broadway, NSW 2007, Australia, August 1994. citeseer.nj.nec.com/reekie94visual.html.

[15] Propellerhead Software. Reason, 1997. www.propellerheads.se.

[16] Synoptic. Virtual waves. www.synoptic.net.

[17] S. Tanimoto. Viva: A visual language for image processing. *Journal of Visual Languages & Computing*, 2-2:127–139, June 1990.

[18] Enrico Vicario. Engineering the usability of a visual formalism for real-time temporal logic. *Journal of Visual Languages & Computing*, 12-6:573–599, 2001. www.idealibrary.com/links/doi/10.1006/jvlc.2001.0214.

[19] B. Wagner. Black-box reuse within frameworks based on visual programming, 1996. citeseer.nj.nec.com/wagner96blackbox.html.

[20] K. N. Whitley and Alan F. Blackwell. Visual programming in the wild: A survey of labview programmers. *Journal of Visual Languages & Computing*, 12-4:435–472, 2001. www.idealibrary.com/links/doi/10.1006/jvlc.2000.0198.

[21] Masoud Yazdani and Lindsey Ford. Reducing the cognitive requirements of visual programming. In *Proceedings of the 1996 IEEE Symposium on Visual Languages (VL '96)*, pages 225–262, 1996. www.computer.org/proceedings/vl/7508/75080255abs.htm.