

VIPER: A Visual Programming Environment for GLU

Dhanraj Rajender and Tony Faustini
Department of Computer Science
Arizona State University
Tempe - AZ 85287, USA
email: raj@lu.eas.asu.edu; voice: (602) 965-3983

Abstract: This paper discusses the implementation of Visual GLU and its integration into a Visual Programming Environment (VIPER). The ultimate goal of VIPER is to allow non-programmers to write applications programs without having to know too much about GLU or the details of different parallel and distributed implementations. Using simple direct mouse manipulation a casual programmer can put together useful applications. VIPER will ensure that the constructions are syntactically correct GLU programs. This paper describes the VIPER system and gives an example program - Laplace's Equations for Heat flow.

1.0 Motivations and Goals of VIPER

The goal of VIPER is to provide a Graphical tool to harness the power and capability of Granular Lucid (GLU) and make it available to a wider audience. In environments where users have limited programming background or other constraints such as time VIPER is a most effective tool for the rapid prototyping and generation of application programs that can take advantage of parallel or distributed hardware. There are many scientific and engineering applications that either take advantage of the multitude of workstations interconnected by a communications network or advantage of newer parallel machines. In practice engineers and scientists do not, except in special cases, have the time or expertise to use newer languages and machines. VIPER is intended to facilitate the use of newer languages (through the use of components written in C) and a visual interconnection language which is easy to use.

GLU obviously provides the framework for generating programs which can execute in a distributed environment. The ideal environment would have the capability of generating GLU code from a well designed Graphical User Interface. This tool should be comprehensive and robust enough to enable the user to build, compile, execute and view programs

without having to leave its environment. The concept of a Visual Programming Environment seemed capable of meeting these goals.

2.0 What is Visual GLU?

Given the goals of the VIPER, a visual representation of GLU was required. A graphical, user friendly mechanism that could enable the layperson to generate useful programs without having to learn GLU and all its intricacies. The VIPER provides a subset of GLU primitives called Visual GLU (VGLU). VGLU provides a visual representation of GLU primitives - GLU operators like `asa`, `fby`,... and arithmetic operators like `+`, `-`,.... The basic idea is to enable the specification of programs using these primitives. Once defined, the VIPER generates code which can then be compiled and executed from within this environment. In effect, VGLU is a specification mechanism which enables the VIPER to generate GLU code.

The VIPER is designed to allow multiple users to create fully functional modules using visual clues. Different people can create programs at different levels of detail. for e.g.: A GLU programmer can use this tool to create a very intricate functional unit - a User Defined Function (UDF) - and submit this unit to the library (a component of the VIPER system). This UDF module then becomes available for others to combine with appropriate UDFs or primitives and create more sophisticated applications. With a good collection of UDFs, this tool can be used at higher levels of abstraction.

Various levels of access capabilities, like the different levels of expertise in Hypercard can be built into it. The VIPER is a high level definition interface to GLU. The goal is to support parallelism while abstracting the details from the user. In order to do this, the VIPER generates GLU code using a demand driven paradigm.

3.0 How it operates

The current version of the VIPER runs on a Sun Sparc on a Unix platform. The GUI is handled through a XWindow interface. The VIPER consists of two main components:

- A Palette which serves as a Library of Primitive Operators and User Defined Functions

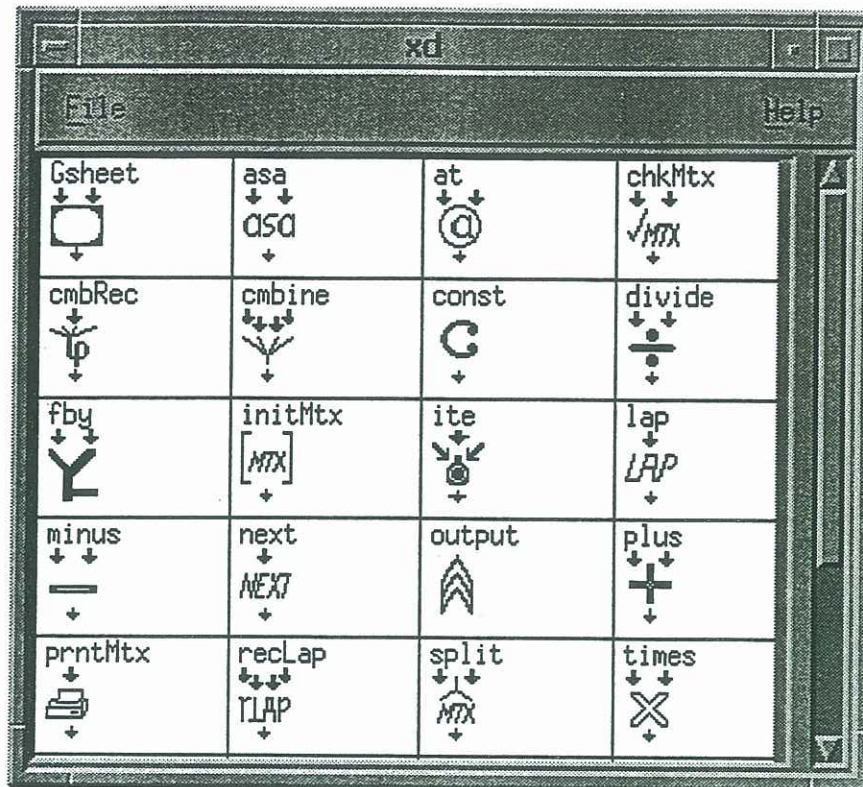


Fig. 1

- One or more user workspaces

3.1 Basic Operation

In order to create a new application, the user begins with a new workspace. Workspaces are generated from the Palette. Using the pointer, suitable components from the palette can be chosen and instantiated on the workspace. These components can be primitive operators or other previously defined UDFs. Each of these components interacts with other components through well defined interfaces. Visually these interfaces appear as input or output pins on the edges of the components. Interactions between the components can be specified by creating wire like connections among the different modules on the workspace. To create a connection between the output pin of component A and input pin 2 of component B, the user clicks on the output pin of component A, and then again on the input pin 2 of component B (the specification could just as easily go from input pin 2 of component B to the output pin of component A).

The VIPER enables the user to specify modules and the relationships between the components. The direct manipulation of components to assert relationships enables the VIPER to check for syntax errors during the development process. Thus errors like connecting two output pins or two input pins can be avoided. Some rudimentary type checking is also provided.

In cases where a new UDF is desired, a new "Container" is created in the current workspace. The "container" is the iconic representation of a UDF. The look and feel of the container is consistent with that of the primitives i.e. interfaces, connection mechanisms and responses to user inputs. In addition, the "container" creates a workspace much like the parent workspace. The implementation details of the function can be defined here. The function can be saved and collapsed back into its icon. UDFs thus created can be submitted to the library Palette. Once accepted, the UDF will become available as independent modules and can be used just like the primitive components in this or other workspaces.

The other way to create a UDFs is to select a part of the net on the main or nested workspace and selecting the *Create UDF* option. The VIPER automatically analyses the number of inputs to this subnet and creates a container icon with the appropriate number of input pins. Since GLU functions have just one output, a major issue to consider is how to handle subnets involving more than one output? For subnets with n outputs, the VIPER generates n functions each having the same input parameters and function body but, with different outputs.

In order to take advantage of the functional programming aspect of GLU, UDFs can also be specified in C. To create a C function, a C-sheet is generated. This sheet supports basic editor commands and enables the user to define C functions. When the function defined in this sheet is submitted to the library palette, it is analyzed for new data types. The VIPER automatically handles otherwise tedious chores like extracting the function declarations, and keeping track of new data types. The interface between the C-functions and the LUCID driver is still rudimentary with great scope for improvement.

3.2 User Interface features

One of the main goals of the VIPER system is to make it a system that people would WANT to use. In trying to meet this goal, we have kept the

look and feel of the windows consistent through the different workspaces. A simple and elegant Edit menu provides the capability to select modules, delete the selected modules from the current workspace, copy the selection onto other workspaces. The user can view current application in either text or icon mode. If the text mode is selected, the VIPER generates code for that particular workspace and displays it in a non-editable mode. Being able to alter the application specification while in text mode involves the VIPER being able to recognize the change and alter the visual specifications , icons, connections and all. This might be a feature in a later version.

The VIPER is expected to be capable of handling real world programs. And programs in the real world frequently get large and complex. Given the visual nature of VGLU, a mechanism is required to focus on small parts of a crowded workspace. Scrollbars have been implemented to help better manage deskpace and clutter. The VIPER also supports a Zoom feature which enables the user to focus in on a part of the subnet.

4.0 Laplace - An Example GLU program

Here is an illustration of a simple GLU program using the VIPER. I fig:1 is the simple textual specification of a Laplace GLU program.

```

• print_mtx(mtx) asa.time done
• where
•   mtx = init_mtx() fby.time
•       rec_lap.time(split(mtx,1),split(mtx,2),split(mtx,3),split(mtx,4));
•
•   mtx_size(_mtx) = get_sz(_mtx);
•   rec_lap.time(M1,M2,M3,M4) = if ((mtx_size(M1)<grain_size) || (mtx_size(M1) % 2==0))
then
•       combine(lap(M1),lap(M2),lap(M3),lap(M4))
•       else
•       combine(nst_rec_lap(M1), nst_rec_lap(M2),
•               nst_rec_lap(M3), nst_rec_lap(M4));
•   nst_rec_lap(_mtx) = rec_lap (split(_mtx,1), split(_mtx,2), split(_mtx,3), split(_mtx,4));
•       fi;
•
•   grain_size = 50;
•   done = check_mtxs (mtx, next.time mtx);
• end

```

Fig. 2

In order to create an equivalent program in the VIPER using VGLU, we start off by creating a new Workspace from the palette. The components are created on this workspace using appropriate point and click sequences. This done, the next task is to specify the relationships between the different components. The following figures illustrate the modular design of the Laplace program. Fig. 3 shows the structure of the *main* program. The main program contains modules *split_mtrx*, *rec_lap*, *init_mtrx*, *check_mtrx*, and *print_mtrx*. The subsequent figures of some of these modules are designed to show that the VIPER system accomodates both GLU and C modules.

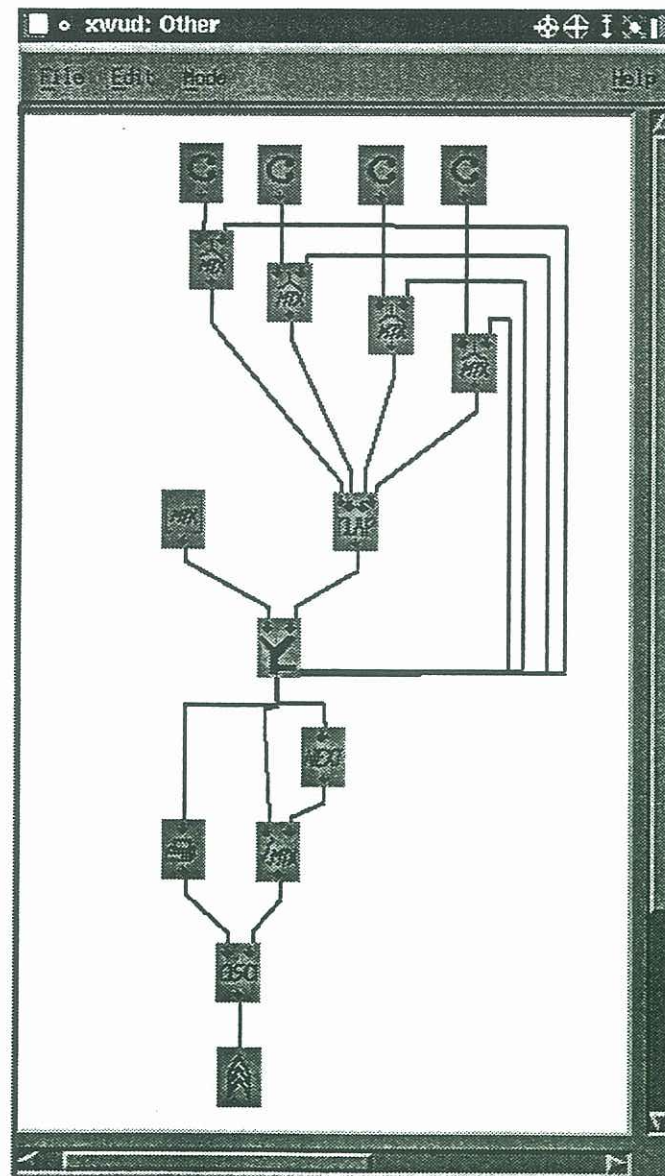


Fig. 3

Fig.4 is the specification of the module *rec_lap*. This figure helps demonstrate the issue of formal parameters to the function. *rec_lap* is

invoked from the main program with four actual parameters. The arrows that exist along the top border of this workspace correspond with the actual parameters in the calling module. All modules have one and only one output.

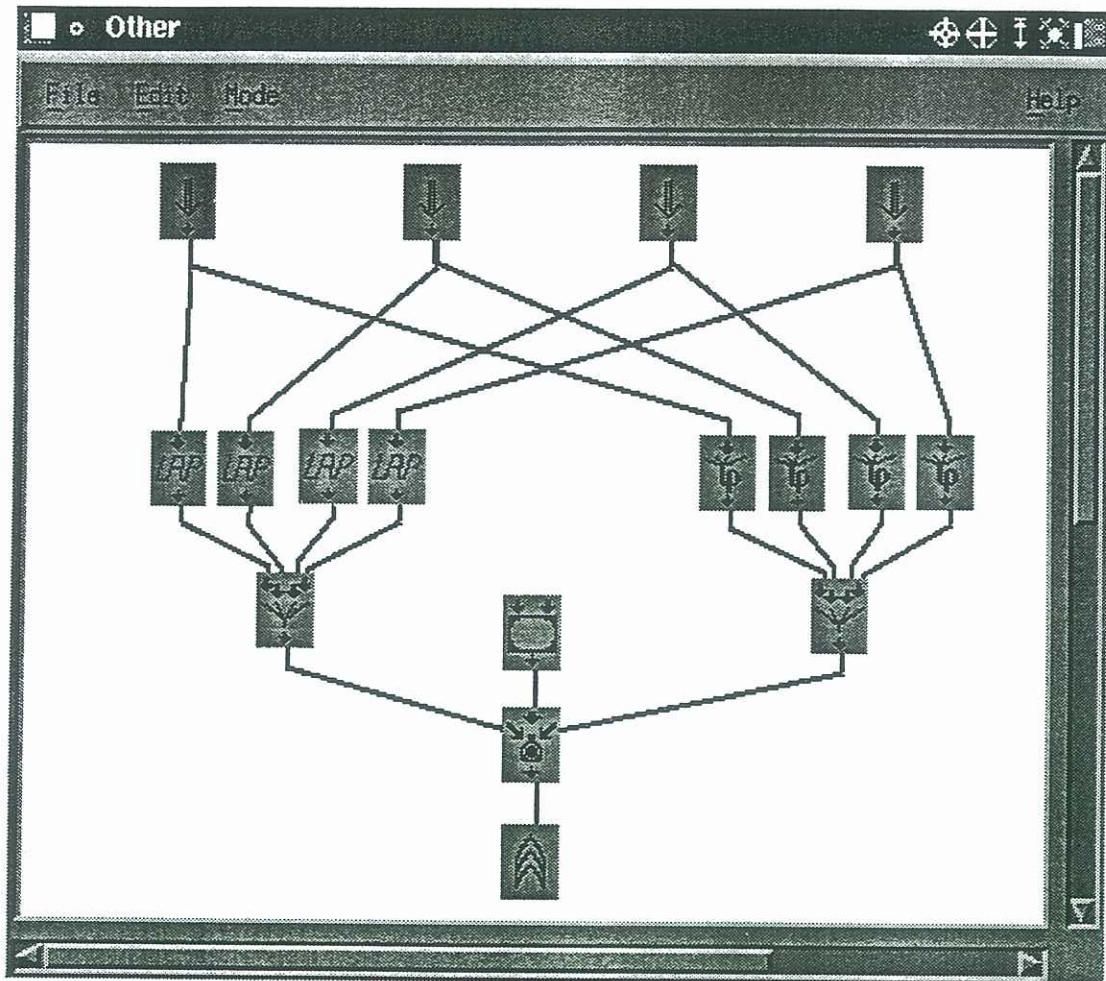


Fig. 4

In order to complete its task, the *rec_lap* module invokes another GLU module called *nst_rec_lap*. The *nst_rec_lap* module needs to be defined before being able to submit the *rec_lap* module to the library. C modules like *combine* must also be defined. In fig.4, the user has instantiated a iconic representation of a C-function (*print_mtrx*). In order to define the module, the VIPER allows the user to invoke an editor by double clicking on button. Fig. 5 gives us an idea of this feature.

```

emacs: Emacs @ vinalu
void print_mtx (node_ptr the_node)
{
    int i = 0;
    int row, col;
    int mtx_sz;
    elem_type *the_mtx = the_node->mtx;
    int size = the_node->size;

    printf("In print_mtx with rows [%d] and cols [%d]\n",size,size);

    mtx_sz = size*size;
    for (i=0; i<mtx_sz; i++)
        printf("%s %3.2f", ((i%size)==0) ? "\n":"\t",the_mtx[i]);

    printf("\n");
    exit(1);
}

Emacs: print_mtx.c      (C)---H11

```

Fig. 5

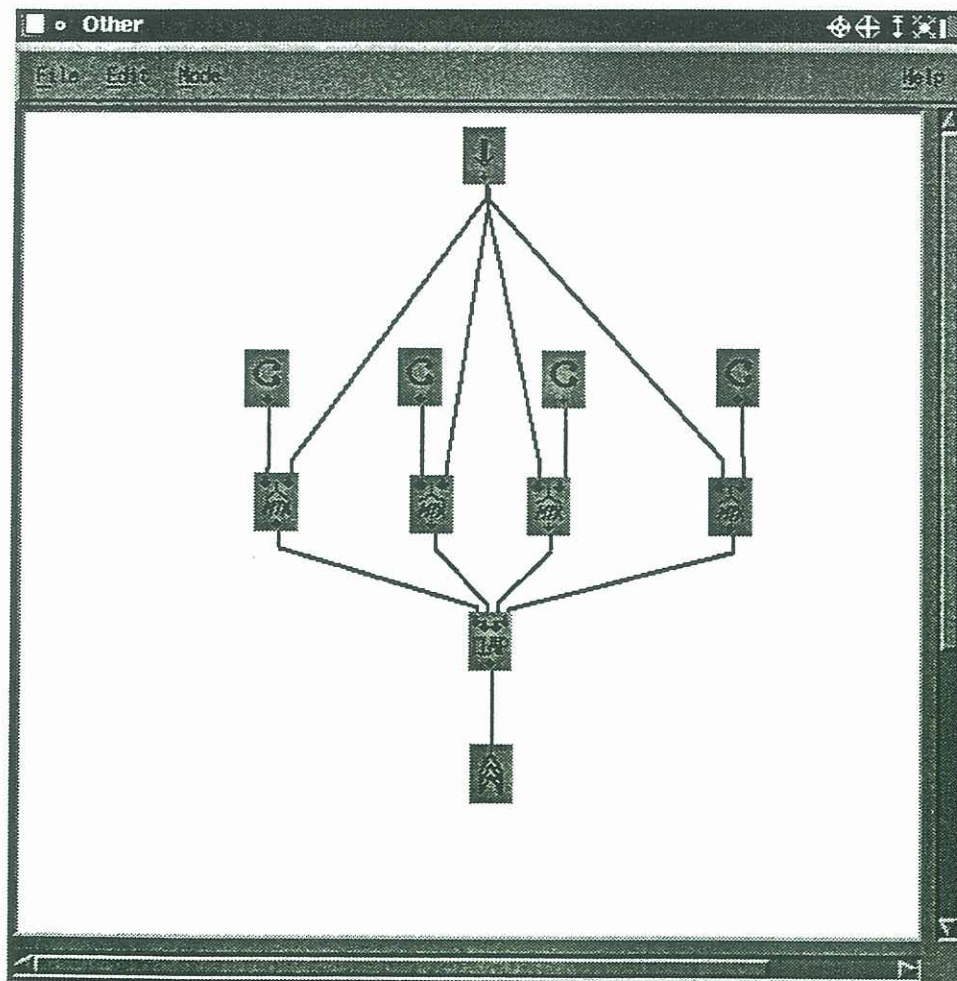


Fig. 6

Once the VGLU specification for the Laplace program is complete, the VIPER system has sufficient information to generate code. The process is

begins by choosing the compile option. The code is generated using a demand driven dataflow approach.

5.0 Interaction with other Systems

Some of the other Visual Programming environments built are the VLucid [3], and Fabrik [2].

The VLucid system and the VIPER system share some of the same goals. i.e. "To provide a high-level, highly interactive visual dataflow programming environment in which even inexperienced users may quickly and easily construct powerful distributed applications." [1] Both systems try to accomplish this goal by providing a Graphical User Interface to the underlying language. While they share a few features in common, what is more interesting is the differences between the two systems.

Conceptually, the most significant difference is that the VLucid system is a comprehensive dynamic environment. It provides runtime support by making each of its operators a lightweight thread with its own stack and set of program registers [3]. On the other hand, the VIPER is an environment that provides the tools necessary to develop a GLU application. It provides support for compilation and execution through system calls and lets the underlying operating system handle the rest. This enables the VIPER to execute in any X- environment.

In comparison with the Fabrik system [2], the VIPER is a static environment. The Fabrik system is a dynamic "alive" system. Functional specifications in Fabrik are evaluated during specification and the results are displayed. While the VIPER can be made dynamic using system libraries to help it interact with the operating system (to compile, execute and display results) it is not intended to be as "alive" as the Fabrik system.

A significant advantage of the VIPER system is that its' VGLU component is not tied down to a particular Operating System or architecture. It merely requires the use of the standard GLU compiler. The VIPER provides all this without bogging the user down with the details of GLU.

6.0 Static -vs- Dynamic issues

The basic design for the VIPER provides support for static application development using visual icons as well as text. From this specification, it generates GLU code. It then interfaces with the standard GLU compiler to compile this code. A run option is provided for each workspace. When this option is selected, the VIPER executes the compiled code by making a simple Unix system call. This minimizes the interface between the VIPER and the underlying operating system and hence affords, greater portability, and independence from operating systems issues. However, the VIPER is being developed with hooks that could change it to a dynamic system.

Now, due to the static nature of the VIPER, we can use it to generate applications that can execute in any environment that can run ordinary (non-visual) GLU programs.

7.0 Future issues to addressed

The latest versions of GLU and Lucid a multi-dimensional and there are a number of approaches that can be taken augment VIPER so that it deals with the multi-dimensional nature of the GLU and Lucid. The simplest solution which works is to augment all the indexical operators like "fby", "next", "@", etc. so that they can be focused on particular dimensions. This is easy to achieve all we have to do is to permit the user to double-click the associated icon when it is placed in the worksheet. For example, the user might drag a "fby" node onto the worksheet. Once on the worksheet the user might double-click the "fby" node. This would produce a dialog box which would contain all the possible dimensions the user could choose to focus on. By default "fby" focuses on "time" and to be in strict conformity to Book Lucid 94 we should say "fby.time" or "fby.t". We have seen how we can easily associate dimensions with an operator but how do we introduce new dimension names ? In Book Lucid 94 new dimension names could be introduced by either introducing the names at the head of a where-clause using the keyword "dimension" or through the use of formal dimension names in function declarations. An example of the first of these is:-

```
x where
  dimension a,b;      // new dimension names
  x = 1 fby.a x+1;
```



```

    end;
or
f.n,m(a,b) = 1 fby.n 1+#.m;    // formal dimension names n and m

```

How do we introduce names visually ? One solution might be to associate a pull-down list with each where-clause or worksheet and for each new dimension name we desired to associate with the worksheet we might be required to select a menu item allowing us to introduce the new name. The names of the dimensions would always be available in the pull-down list in the worksheet. What about the formal dimension names ? Without going into details another scheme similar to the last could be used. This would give us the equivalent syntactic power as Lucid or GLU. Although this approach is workable it is not very visual the extra dimensions are treated as they are in textual Lucid. The visual version might have a slight advantage over its textual counterpart in that it could ensure that only syntactically correct GLU programs could be constructed.

There are other ways of visually dealing with multi-dimensionality that need to be explored. For example, a spreadsheet is a useful way of specifying two or three dimensions. Could this be used ? Clearly it could easily be used to specify multi-dimensional input constants or for visualizing the values associated with a multi-dimensional expression in Lucid. Could we also use this to specify multi-dimensional expressions ? The answer to this question is less clear and needs to be thought about further.

8.0 References

- [1] Ashcroft et al. 1994. Multi-dimensional Declarative Programming - Oxford University Press, 1994.
- [2] Ingalls, D., Wallace, S., Chow, Y. Y., Ludolph, F. and Doyle, K. 1988. Fabrik, A Visual Programming Environment. - *OOPSLA '88 Conference Proceedings*. San Diego CA (September): 176-190
- [3] Mitchell, W. H., Faustini, A. A., The vLucid Distributed Visual Dataflow Environment.