The State of the Art of Visual Languages for Visualization

C. Williams, J. Rasure
Dept. of Electrical and Computer Engineering
University of New Mexico
Albuquerque, NM 87131
{carla,rasure}@chama.ece.unm.edu

C. Hansen
Advanced Computing Laboratory
Los Alamos National Laboratory
Los Alamos, NM 87545
hansen@acl.lanl.gov

Abstract

Data flow visual language systems are now being used to provide sophisticated environments for the visualization of scientific data. These systems are evolving rapidly and are beginning to encompass related technologies such as distributed computing and user interface development systems. This paper presents a hierarchical classification of the components and issues involved, giving the reader an understanding of the design decisions and trade-offs that the developers of these systems are making. The component categories can be used as a framework for discussing where interoperability of competing visual programming environments might occur and what the future holds for these systems.

1. Introduction

The newest generation of sophisticated environments for the visualization of scientific data typically include a data flow based visual language which provides a graphical interface to predefined operators to import, manipulate and display data [1]. Applications are constructed by using a mouse to link together icons representing the operators to create an executable block diagram. This facilitates the rapid prototyping of solutions to complex data analysis/visualization problems.

The purpose of this paper is to describe the components that characterize a data flow based visualization environment and to identify the issues that must be addressed. This provides a framework within which to understand and discuss the design decisions and trade-offs made during the development of these systems. It also helps to identify the level at which interoperability between current visual programming languages might occur and directions for future work. The purpose of the paper is not to make side-by-side comparisons of competing systems. However, the following discussion is based upon the implementation of current systems including Khoros from the University of New Mexico[2], AVS from AVS Inc.[3], Explorer from Silicon Graphics[4], apE from Tara Visual[5], and Visualization Data Explorer from IBM[6].

<table>
<thead>
<tr>
<th>Categories</th>
<th>Components</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Visual Language Programming Paradigm</strong></td>
<td>Language specification</td>
</tr>
<tr>
<td></td>
<td>Programmer's interface</td>
</tr>
<tr>
<td><strong>Visual Language Program Process</strong></td>
<td>Operator definition</td>
</tr>
<tr>
<td></td>
<td>Execution model</td>
</tr>
<tr>
<td></td>
<td>Data model</td>
</tr>
<tr>
<td></td>
<td>Program monitoring</td>
</tr>
<tr>
<td><strong>Development Environment</strong></td>
<td>User interface development system</td>
</tr>
<tr>
<td></td>
<td>Operator programming interface</td>
</tr>
<tr>
<td></td>
<td>Configuration management</td>
</tr>
<tr>
<td><strong>Visualization Requirements</strong></td>
<td>Data import/export</td>
</tr>
<tr>
<td></td>
<td>Data manipulations</td>
</tr>
<tr>
<td></td>
<td>Data display</td>
</tr>
<tr>
<td><strong>Considerations for Future Work</strong></td>
<td>Collaborative programming</td>
</tr>
<tr>
<td></td>
<td>Interoperability</td>
</tr>
</tbody>
</table>

Table 1: Hierarchical classification categories

Table 1 summarizes the categories of components used for the classification. The choice of data flow as the Visual Language Programming Paradigm is the underlying concept upon which these systems are based. This representation is relatively consistent across current visual language environments, but the implementation of the actual mechanisms that support the execution of the Visual Language Program Process vary dramatically. A Development Environment is not a requirement of a visual language system, but is typically included because: 1) the original developers of the visual language needed one to efficiently build the system and 2) visual languages can function as application builders if a development environment is available to extend the system. Visualization Requirements detail the issues unique to the application of visual language envi-

0-8186-2897-9/92 $0.00 © 1992 IEEE
Environments to the analysis and display of scientific data. Finally, several areas are identified as Considerations for Future Work. These components are discussed in detail in the following sections.

2. Visual Language Programming Paradigm

The visual language specification component defines the graphical elements that comprise the programming language and the meaning of those elements within the context of the visual programming paradigm includes syntax and semantics. The programmer's interface to a visual language provides for access to the graphical elements of the language paradigm and may also provide for textual entry of additional program information.

2.1 Language Specification

The syntax of a visual language may be thought of as the graphical elements that define its outward appearance. For a data flow language these are the operators (nodes) and connections that make up the flow graph network. Sample visual programming workspaces are shown in the color plates: Figure 6 (AVS), Figure 7 (SGI Explorer), and Figure 8 (Khoros).

Semantics refers to the meaning ascribed to valid constructs in the language. Operators can be distinguished based on their function. These include input, output, transfer, control, and extractor nodes. To solve complex problems using visual languages, control nodes are needed to provide run-time decisions to determine the execution order for operators. Control operators may include for-loops, while-loops, if-then-else, merge, and trigger. Data dependent control operators extract information and then set values which change operator parameters to direct the flow of program execution. Figure 1 identifies several of these different types of operators.

Figure 1: Various Khoros operator (node) types.

The expression of a visual hierarchy varies a great deal among different systems. Some systems implement hierarchical operators as a border around a grouping of operators, while others replace the grouping with a single (higher level) operator. For example, the Khoros workspace of Figure 8 contains two instances of a user-defined subprocedure node in the branches following the if-then-else operator. The first (identified by the double border) is closed, while the second is open to show the subprocedure's internal network. Other systems implement a "graphical hierarchy" that merely creates a pictorial hierarchy (like a group command in a draw package) but without any true programming abstraction. For a visual language to support recursion each instance of an operator must have its own local data per instance of execution. None of the current visual language systems support operators that allow recursion.

2.2 Programmer's Interface

The programmer's interface to the visual language provides, as a minimum, a graphical editor for constructing the program flow graph. However, a language based exclusively on data flow cannot provide the best interface for every interaction with the programmer. Consequently, most data flow visual languages are augmented with additional modes of user input. These are typically used to facilitate the setting of operator parameters via knobs, sliders, buttons, and keyboard entry. Figure 2 shows such a widget for controlling the isosurface value.

Figure 2: SGI Explorer control panel for isosurface operator containing dial widget and toggle buttons.

The keyboard can also be used to enter textual expressions for operator parameters such as the control variables for the count-loop in Figure 3.

Figure 3: Khoros count-loop and variables pane.
Systems that combine several user interface modes are referred to as multiparadigm systems. Graphical editors may also be syntax sensitive to prevent the programmer from creating an illegal program network. Figure 4 demonstrates how colors can be used to denote data types.

Figure 4: AVS network with different colored connections representing different data types. Active operator is highlighted.

An efficient means of accessing operators and selecting one from those available is essential to the productivity of the visual programmer. Some systems advocate a palette of icons representing the nodes, others display menus listing the textual names of operators. Figure 5 shows both a display menu (at the top) and a palette (at the bottom) for accessing operators. Either of these methods becomes cumbersome when there are a significant number of operators (say, greater than 50) available. A hierarchical arrangement of operators grouped into appropriate categories is necessary, the number of levels determined by the size of the system. While this does provide for an orderly search, it can increase the time required to access operators.

And finally, aside from showing the general flow of data through the network, a visual program is not self documenting. The programming interface should support the addition of documentation that can be associated with the various graphical components (operators and connections) of the visual program.

3. Visual Language Program Process

A visual program process is analogous to an operating system process—it is a running program together with its associated state and data. Issues that must be addressed include the functionality embodied within an operator definition, the execution model employed, and the characteristics of the data model that an operator works on.

3.1 Operator Definition

Operators for traditional textual languages, such as Fortran and C, are relatively simple. For example, an "add" operator uses two scalar values as its input to produce a single scalar output. The actual machine instruction executed is a function of the data type of the scalar values, which is determined at compile time. An analogous, but more complex, description can be given for an operator in a visual language, beginning with an understanding of the abstraction level of the operator. An "add" operation typically available in a visualization environment may be many levels of abstraction above the analogous machine instruction. For example, adding pixel values of two images together. Similarly, an operation to perform some type of data rendering is an abstract operator because its actual functionality is dependent on the data model (type) to be acted on by the operator. In other words, the level of abstraction is derived from the data model used and the function being performed. Visual languages derive much
of their power by having abstract operators which hide the complexity of the data model and data manipulation from the user.

Current visual language systems utilize a variety of techniques to specify the interface to operator modules (which may be differentiated by their binding method, as described in the following paragraphs):
- program - an operator is a program executable by the operating system (bound as a UNIX process),
- function call - an operator is a textual language function call interface (dynamically linked or compiled),
- parser - an operator is a string passed to an expression parser (visual language is translated as input to the parser, which then interprets the string), or
- code - an operator is a set of textual language instructions (compiled or interpreted at run-time).

Each interface specification technique has its own inherent advantages and disadvantages and also impacts the implementation of the data and execution models as discussed below.

There are a number of different approaches to binding an operator to a specific instance for execution within a visual program. These are also influenced by the method selected for the operator interface. In general, the later the binding time the better, but at the cost of increased overhead in the execution model. For example, an operator with a program interface may be initiated as a process at the time it is accessed from a menu and placed in the execution model. When an operator with a function call interface, may be executed when it is dynamically linked by the dispatcher (late binding), or it may be executed as part of a compiled visual program (early binding).

Operators are polymorphic if the output of any operator can serve as the input to any other operator. This can be achieved if the operators do not have any side affects with respect to the rest of the visual program and if they can fully interpret the data model with respect to the operation being performed. Using this definition, it is not likely that a very comprehensive system will ever be fully polymorphic—it would not be practical. For example, the “add” operation may be able to operate on any specific instance of the data model, but it is not reasonable to expect a “volume renderer” to function correctly for all possible applications of the data model (for example a 2 D field).

The implementation of the operator functionality may be coded in a traditional textual language such as C or Fortran or a parallel programming language like C* or Fortran 90. If the programming paradigm supports hierarchy, a higher level operator may be built up as a visual program consisting of a flow graph of lower level operators.

By definition, data flow operators consume and produce blocks of data. “Data granularity” refers to the size of the block of data processed by an operator. Consider the data moving through a visual process to be a stream of blocks. If the block size is the same as an instance of the data model (the norm for current visualization environments), then the data granularity is large. Hence the name “large-grain data flow.” If the block size is the same as an atomic element of the stream transport, then the data is fine-grain. “Fine-grain data flow” operators are needed for real-time process control or real-time signal processing.

### 3.2 Execution Model

To become a visual process a visual program must be executed. The program, described as a flowgraph of operators, is first translated, then the operators are scheduled (which may be inherent in the translation) and dispatched.

Most visual languages are translated via interpretation to provide the user with a higher level of interactivity with the visual process and to allow for partial program execution. But, interpretation does add to the overhead of executing the visual process. It is very difficult to design a compiler for a visual language because of the attributes of the data flow paradigm and the attributes of the operators. Systems that have been implemented with a compiler embed a “quasi-static” scheduler to manage the visual process execution.

The scheduler of a visual program determines which operator should be active depending on data availability and demand for data. Most systems have a global coordinating scheduler that keeps track of such information, but it is also desirable to have the operator itself determine when it should be active. In the case of operators that have a program interface, the scheduler may also have knowledge of a network of other machines on which it may distribute processes at the operating system level.

The dispatcher is responsible for binding the operator to the visual process. For an operator with a function call interface, the dispatcher can dynamically link in the object code. A dispatcher for program operators may execute an operating system process or it may send new execution information to an operator that is already running. When visual processes are being distributed across a network of machines, each machine may have its own local dispatcher which communicates with the scheduler.

### 3.3 Data Model

Large-grain data flow systems may not save intermediate data (no permanence) because storing of all the data generated within the visual process is prohibitive. Instead, intermediate data results must be recomputed in order to produce new output. These systems do not save state and
therefore program execution cannot be interrupted nor can partially executed programs be recovered.

The transport of data between operators can vary from using allocated memory of a single process (when operators with function call interfaces are all linked to a single process) to using a combination of UNIX shared memory and sockets. In the first case, the data can easily be global to the entire visual program which greatly simplifies data transport and storage. In the second case, there are typically many processes communicating with each other. If these are on different machines, efficient use of data transport (permanent and non permanent) dramatically effects the performance of the system.

A visual language for a specific application domain will typically have its own internal data structure for representing the data model accepted by its operators. As polymorphism of operators is attempted across several application domains this representation necessarily becomes very complex.

3.4 Program Monitoring

The monitoring of a visual process is useful for determining its proper performance and in aiding the programmer in debugging the program. Interpreted systems typically highlight the operator(s) as they are running to give the programmer an execution view of the visual process state. This is shown in Figure 4 where the colorizer operator is active (highlighted). Most systems also allow the user to probe a connection so that intermediate data can be viewed or rendered. The notion of visual process monitoring is sometimes referred to as "program visualization" [7]. For example, Khoros operators can be individually executed as a network is incrementally built up and single stepped for debugging purposes.

4. Software Development Environment

The software system has yet to be created which can meet everyone’s needs “off-the-shelf.” Since the field of scientific visualization is in its infancy, with new techniques and applications appearing almost daily, visualization environments must be designed so that they can be modified and extended. The components of a visual programming environment that provide the support needed to add new operators to the system include a user interface development system (UIDS) [8] for specification of program interface(s) and a method of describing data structures so that a common set of data transport utilities can be used. If a high-level specification is used for the operator programming interface and if the data structure can be described abstractly, much of the code needed to implement a new operator can be generated automatically. Once a new operator is created, it must be installed, documented, and maintained via system configuration management tools.

4.1 User Interface Development System

Building a visual programming system is a complex software task that integrates user interface-software, data processing software, data rendering software, and the host operating system and windowing system. All of the current visual language systems are layered on either a custom user interface (UI) software toolkit or an existing standard (MOTIF, Windows, Athena, OLIT, etc.). For the display of data, developers of visual programming systems choose to use either a custom toolkit and/or a combination of standards (PHIGS, PEX, GL, Xlib, etc.).

To make the development of the visual language programming system more efficient, developers may either rely on using a commercial user interface builder, or build all of the user interface by hand, or develop their own user interface builder that works well with the chosen UI toolkit and visualization toolkit. The user interface builder often has a code generator as a back end. The generated code acts as an application framework for a new operator. Figure 9 shows the Khoros interactive User Interface Specification (UIS) file Editor which describes the graphical interface of a new operator to the code generators.

Providing the programmer (who wishes to create a new operator) with a development environment allows the basic system to be extended to better solve specific applications. In addition, the visual language interface might only be used in the early stages of prototyping a solution, with the end product not involving a visual program at all. Instead, a simplified, custom interactive interface may be created for easier access by end users.

4.2 Operator Programming Interface

Most current visual programming systems do not include a complete UIDS to aid the end user with creating a custom interactive application, but most do provide for a great deal of flexibility in adding new operators to the visual language. For example, the operator programmer may be given a library of programs for accessing a specific data structure or an abstract data type that can encompass many possible data structures. This library would include functions that would be used to make the operator compatible with other operators within the visual language.

4.3 Configuration Management

Finally, an operator needs to be installed, maintained, and made accessible to the visual programmers. If this is to be done in a productive manner, i.e. not in a haphazard
5. Visualization Requirements

The requirements of a software environment with respect to the visualization of scientific data are somewhat de-coupled from the components of the visual language itself. While it is true that the visual language is the mechanism used for generating an image, the same can be done with textual based programming language. However, the visual programming paradigm provides the scientist or visualization specialist with a flexible tool for graphically representing the sequence of operations that are required: First the data must be imported into the system; then manipulations are performed to achieve some desired result, which is then displayed using some visualization technique.

5.1 Data Import/Export

Existing visualization systems each use their own internal data structure to represent the data model accepted by their operators. Therefore, users’ data must be translated, usually through a file-read or library call, into the appropriate data format. The data structures used for scientific visualization can be classified into four groups: data on a structured uniform cartesian grid with implicit coordinate information, data on a structured non-uniform cartesian grid with explicit coordinate information (i.e. curvilinear grids), data on a non-structured grid and geometry data. The requirements are application specific but if the data structure for a particular visualization system is not robust enough to handle a certain class of data, the user must seek a solution elsewhere. Some systems utilize multiple data formats to represent different types of data. This implies that there are different operators for different data classes. Thus a user might not be able to accomplish a desired visualization if an operator is not available for a particular technique within a data class. Other systems use a unified data model where all data is of a general class with specializations which aid in better memory management as well as assisting in the visual programming interface by indicating and allowing only legal connections. Polymorphic operators can then manipulate the data appropriately for each specialization.

Ease of use is another differentiating factor between visualization systems. Data importation either involves making library calls, editing an ASCII file and using that for data translation possibly through a GUI, or creating an operator to translate user data into the system’s internal data structure. Most current systems do not force the user to make library calls for the data translation step. A stand-alone translation application may be functionally powerful but difficult to use.

5.2 Data Manipulation

The data manipulation is performed by operators within the system which have the capability to compute, enrich, enhance, extract or otherwise transform data. Manipulations can be grouped into three distinct classes of transformations: data to image, data to data, and data to geometry. Data to image is used in direct image-based visualization. In this class, data values are mapped directly into a color space and the result is a displayable image. An example is shown in Figure 6 which maps the pressure field for a global ocean model to a color continuum. Data to data transforms permit manipulation of the data to be visualized. Figure 8 demonstrates this manipulation by performing a Marr filter on the image data (based on the data variance). Extracting scalar values from a vector field is another example which is shown in Figure 6. Data to geometry transforms are required since these systems render geometry, typically utilizing Z-Buffer hardware, during the visualization process. This is shown in Figure 7 where an isosurface representing DNA nuclear material is extracted from a confocal microscopy data set of a dying cell.

5.3 Data Display

The techniques used for data display include graphics and rendering: transparent surfaces, volumetric rendering techniques, etc. Additionally, there is a presentation requirement for 2-dimensional visualization or 3-dimensional visualization. Most systems excel on one or the other, both not in both. Coupled with this requirement is the need for annotation of data and images. Since so much of scientific data is based on dynamical models, animation is an important requirement. There are two basic types of animation: image based and geometry based. Image based animation refers to the process of generating a sequence of images and playing them back to form an animation loop. Geometry based animation refers to the process of saving disjoint (over time) geometries and rendering those back as a sequence. This, of course, requires rendering hardware.

6. Conclusions and Directions for Future Work

The hierarchical classification of visual language system components presented in this paper has been developed and refined over the past two years based on the authors’ experiences with creating a visual programming environment, analyzing and using other visualization sys-
tems, and attempting to specify the "ideal" system for the future. This classification will continue to change as visual languages continue to evolve, but the categories of components defined in Table 1 should remain relatively constant, while most of the changes will occur in the specific details as we gain a better understanding of the issues involved.

The visual programming environment of the future will be an elegant blend of all the features discussed above. It will be used as a software development environment for scientific research which supports many levels of interaction within the system. Additional areas which should be considered when looking toward the future include collaborative programming and interoperability between systems.

6.1 Collaborative Programming

Visual programming provide a great potential for collaborative programming. If the user interface of the visual programming language is distributed to a set of users across a network of machines, they could simultaneously write a single program. While the concept of group-ware programming is not new, its application to visual programming will allow for a more natural implementation. In addition, a heterogeneous distributed computing environment must be supported by the scheduler so that the location of data and compute resources is transparent across the network of participants.

6.2 Interoperability

Current visualization environments, while similar in many respects, exhibit a wide disparity in the types of operators which are included with each system. These differences may be attributed to the original application interests of their respective developers. For example, some systems are strong in data display operators using various rendering techniques. Others emphasize data manipulation with many operators for data analysis.

In an ideal world, the visual programmer could combine the results from several systems, capitalizing on the strengths of each. This interoperability could take place a several different levels, from the sharing of data, to the sharing of operators across different environments. To achieve operator compatibility, software development tools will be required within visual programming environments that allow a user to easily wrap a non-standard function call interface with the data token conversion, data transport, etc. as needed. This conversion process will lead to a better understanding of how various systems are implemented and may motivate developers to provide a common set of language support mechanisms and interface definitions that will facilitate interoperability.

Acknowledgments

This work was partially sponsored by Los Alamos National Laboratory (contract number 9-XF99-E6828-1). It was also performed under the auspices of the Department of Energy and was partially funded by Department of Energy High Performance Computing Grant KC0701 and NSF STC-CPRC CCR-912008.

References

Figure 6: AVS workspace demonstrating image-based visualization. Data is from ocean circulation model computed at LANL's ACL on the CM-2. Pressure field at 80 meters is mapped by a linear colormap showing regions of high and low pressure.

Figure 7: SGI Explorer workspace demonstrating geometry-based visualization. Dataset was sampled with a confocal microscope in the Life Sciences Division at LANL. Isosurfaces represent DNA nuclear regions of a dying cell.

Figure 8: Khoros workspace with two instances of a user-defined Marr filter subprocedure that contains a loop to pass through the data twice. The variance of the image is extracted and used to determine which version of the Marr filter should be executed.

Figure 9: Khoros Interactive Graphical User Interface Editor as used to define the UIS file for a new operator. Code generators then create all the interface code to integrate the operator into the visual language.

(See color plates, p. CP-23.)