Visual Programming Using Structured Data Flow

by Jeffrey Kodosky, Jack MacCrisken, and Gary Rymar
Nihon National Instrument K.K.

Abstract

In the quest for a visual programming language that can be used effectively by a broad range of people with different programming skill levels, a system has been developed based on a dataflow model extended with graphical control-flow structures. This system eliminates many of the problems associated with using dataflow graphs as a programming language, making programs easier to construct while still preserving the natural understandability of dataflow diagrams. Subroutine-like modules are encapsulated using an icon-based notation that facilitates the construction of large programs through hierarchical composition. Finally, a compiler has been constructed that generates machine code that is comparable to code generated by compilers of conventional languages such as Pascal and C. The visual language, called G, is embedded in LabVIEW®, a scientific software system for laboratory automation and simulation.

Introduction

The term visual programming has been used to describe a wide range of activities, including image processing, animation, presentation of data in graphical form, and construction of specifications, documentation, or programs in pictorial rather than textual form. At National Instruments, our research has focused on the development of a practical system for constructing programs using a diagrammatic method. The method employed in LabVIEW combines two previously unrelated programming methodologies—structured programming and dataflow programming—and embeds them in a graphical editing and execution system.

Our work is motivated by the laboratory automation area. Interactive use of benchtop instruments is straightforward and efficient for the laboratory scientist, technician, or engineer, but automation is not. Introducing a computer to the benchtop and writing programs to automate the tests or experiments introduces a tremendous amount of artificial complexity and results in a large loss of productivity.

For example, the frequency response of an audio component could be tested with a signal generator connected to the input and an AC voltmeter connected to the output. The test operator selects a frequency on the signal generator, logs the voltage reading, repeats the process for multiple frequencies from 20 Hz to 20 kHz, and finally plots the result.

With computer-controlled instruments, the entire procedure can be done automatically. In a production environment, where many units will be tested with the same procedure, the cost of developing the instrumentation program using traditional methods is justified. In a laboratory environment, however, the testing and experimentation is continually changing, so the programming cost and time become major concerns.

To address this problem, we adopted the virtual instrument metaphor, whereby software for laboratory automation is viewed as a hierarchy of instrument-like modules containing interactive front panels and programs in the form of block diagrams. Within the block diagram, interconnected icons reference lower level modules. The hierarchical composition of execution-time user interfaces (front panels) with design-time visual programming (block diagrams) is applicable to many different areas of science and engineering. The LabVIEW software system contains editors for front panels and block diagrams, extensive built-in functions and libraries of analysis functions, and the embedded G block diagram language.

Designing a useful and practical visual programming language is a challenging task. A two-dimensional graphical notation improves over textual notation only if the graphical notation is expressive enough for a large class of algorithms and is in some measure more efficient than a traditional, textual programming language. For instance, a flow chart conveys little more information than does the indentation of the text of a structured program but takes considerably more space, making it a poor candidate for visual programming. Conversely, state diagrams convey information about state relationships more efficiently than text representations, but state diagrams are excessively cumbersome for all but a small class of programs.
A number of control-flow-based graphical notations have been used as the basis for visual programming systems, including flow charts, state diagrams, Petri nets, and Nassi-Shneiderman diagrams. Because dataflow diagrams are superior in many respects, such as in the depiction of parallelism, we based the visual programming language G on the dataflow model.

**Dataflow Diagrams**

A simple pure, data-driven dataflow diagram is illustrated in Figure 1. It is a directed, acyclic graph consisting of nodes, arcs, terminals, and data tokens. Terminals are the connections to the external world, and act as the sources or sinks of data tokens. Arcs are the directed paths over which data tokens move, and nodes are the locations in which computations are performed. The fan-out of an arc implies copying the data token; the fan-in of an arc is disallowed. A node consumes tokens on its input arc and produces new tokens on its output arcs. What makes the diagram data-driven is the firing rule, which states that a node cannot execute until all of its input arcs have a data token available, at which time the node consumes one token from each input arc, performs the computation, and produces one token for each output arc. In Figure 1, node J has already executed, K and L are eligible to execute, and M is still ineligible because it needs a token on its second input. What makes the diagram pure is the requirement that the node computations have no side effects.

![Figure 1. A Simple Dataflow Diagram](image)

In contrast to the control-flow model, the dataflow model has no concept of locus-of-control, no program counter, and no globally accessible memory. A data token exists only from its production by a node or input terminal to its consumption by another node or output terminal. All nodes that are eligible to execute can do so in any order or even in parallel; the results of the diagram will be the same in all cases. Dataflow diagrams specify the data dependency between computations, but do not specifically force any particular sequence of independent computations.

The previously described dataflow model is simple and easy to analyze. It is also far too simplistic as a programming language. In particular, the dataflow model lacks the provision for conditional or iterative computations. To remedy this situation, the firing rule is typically relaxed, cycles are allowed in the graph, and the notion of priming an arc is introduced. Figure 2b illustrates this more general dataflow model with a program that computes a root of an equation using Newton's method (figure 2a). Newton's method is an interactive procedure that refines an estimate of the location of a zero crossing of a function by computing the value of the function and its derivative, and calculating the zero crossing of the tangent line. This calculation is then used as the new estimate, and the procedure is repeated until successive estimates agree to the desired accuracy.

The diagram in Figure 2b contains two nodes with generalized firing rules. The selector node fires with only two of its three inputs containing tokens, for example, a token on the T input and a True token on its Boolean S input. When the distributor node fires, it produces a token on only one of its two outputs; that is, based on the value of its D input, the distributor node produces a copy of its first input on either the T or F output. Prior to executing the diagram, the selector Boolean S input must be primed with a True token.

![Figure 2a. Newton's Method](image)

![Figure 2b. An Implementation of Newton's Method Using an Extended Dataflow Model](image)

---


By extending pure dataflow as just described, the language has become more expressive, but at a terrible cost in clarity. It is no longer obvious when or if an arc will carry a data token or a node will execute. It is much more difficult to understand the computation of a diagram that contains cycles, primed arcs, and nodes with relaxed firing rules, and even harder still to construct a correct diagram.

Structured Dataflow Diagrams

Structured programming is an established methodology and its advantages are well known. Program looping structures are well understood. In practical terms, the dataflow extension of the previous section forced loops to be built using goto statements—a giant step backwards in programming methodology. The preferred solution would be an extension to pure dataflow programming that preserves its clarity by preserving its firing rules and acyclic structure, yet incorporates the proven benefits of the structured programming methodology. Programmers know how to reason about loops and conditionals, so those structures should be built in as fundamental components of the diagramming language rather than be constructed of more primitive pieces.

The principal property of a program structure that enhances clarity and makes analysis and verification tractable is the boundary between the body of the structure and the rest of the program. Access to the code in the body of the structure is restricted—it can be entered only from the top. The natural diagrammatic analog is a box. Representing a program structure as a box syntactically divides the dataflow diagram into two parts—the inside or body of the structure and the rest of the diagram. Requiring the box to behave like a node as far as the rest of the diagram is concerned, and the body to behave like an isolated diagram, means that the only modification to the pure dataflow methodology is the addition of boxes with program structure semantics such as loop behavior or conditional behavior.

Figure 3 shows the four control structures of G. There are special terminals associated with a structure that are used to control its behavior or indicate its state. These terminals are fixed in number and produce or consume data of a particular type. The junction of an arc and a structure border is a special terminal called a tunnel. Tunnels can be any data type and can occur in any number. All the structures behave like nodes as far as the rest of the diagram is concerned, meaning that data tokens must be available at all input terminals before the structure can execute, and data tokens are produced for all output terminals when the structure completes.

The while loop structure has two special terminals inside it—the count, $\text{count}$, is a source of an integer value, and the continuation flag, $\text{flag}$, is a sink for a Boolean value. The while loop executes its subdiagram, the body of the loop, which produces a token on the arc connected to $\text{flag}$. When the subdiagram completes and the value of $\text{flag}$ is False, the loop terminates and the tokens at the output tunnels propagate along the output arcs. If the value of $\text{flag}$ is True, the while loop re-executes the subdiagram using the same tokens on the input tunnels and overwriting those on the output tunnels. The value produced by $\text{count}$ during the first iteration is 0, and it is incremented for each successive iteration.

The for loop is similar to the while loop in that it re-executes its subdiagram; however, the for loop does so a predetermined number of times, specified by the token at the special integer terminal $\text{count}$. As viewed from the rest of the diagram, $\text{count}$ is a sink, but as viewed from the body of the for loop, $\text{count}$ is a source. Tunnels on both the for loop and while loop structures can be designated as indexing. An input indexing tunnel is a sink for an array data token, but inside the body of the loop the input indexing tunnel is a source for successive elements of the array on successive iterations. Conversely, an output indexing tunnel accumulates the values from successive iterations of the body into an array data token.

The case structure has multiple subdiagrams that overlay the same screen area. The only subdiagram that is executed is the one selected by the value of the integer token (or Boolean token in the case of two subdiagrams) supplied to the selector terminal, $\text{selector}$. Each case structure subdiagram must supply values for each of the output tunnels. The case structure can be thought of as a table of functions of the same arity, that is, functions that take the same number and type of input and output parameters. The selector terminal indexes the table and applies the selected function to the input tokens to produce the output tokens.

The sequence structure is similar to the case structure in that multiple subdiagrams exist; however, the sequence always executes each subdiagram in succession. Its main use is as a means for specifying the order of execution when there are no explicit data dependencies between functions. The sequence structure is typically used when dealing with
side effects such as input from and output to peripheral devices.

Figure 4 shows another construction, called a shift register, that is available on each of the loop structures. A shift register consists of a left part and a right part related by having the same vertical location on the left and right sides of the structure. The left part is a sink to the outside but a source to the loop body; the right part is the opposite. A data token supplied to the right side by the subdiagram is moved to the left side prior to the start of the next iteration. The shift register can be thought of as a program variable that is updated inside the loop. A token supplied to the left side prior to executing the loop initializes the variable, and the token emitted from the right side after the loop completes has the value of the variable after the last iteration.

![Figure 4. Shift Registers on Loop Structures](image)

Figure 5 shows the Newton's method example as programmed in G. The terminal at the left supplies the shift register with the initial estimate of the root. Each iteration produces a better estimate, which replaces the previous one in the shift register. The loop continues as long as the absolute value of the difference between successive estimates is larger than the desired limit. After the loop terminates, the last estimate in the shift register is propagated to the terminal at the right.

![Figure 5. Newton's Method in G](image)

Hierarchical Structured Dataflow Diagrams

To handle large programs, a practical visual programming language must have an abstraction mechanism analogous to the subroutine in a conventional text language. In G, this is done by associating an icon with a diagram. Regions of the icon are designated to correspond with the source and sink terminals of the diagram. The resulting module, called a virtual instrument (VI), is referenced from another diagram by means of the icon. Figure 6 shows a diagram, Untitled 2, that normalizes an array. The diagram references a subVI, Magnitude, that computes the maximum absolute value in an array. Notice that the input indexing tunnel automatically sets the value for \(N\) on each for loop. The wiring tool, a cursor that looks like a little spool of wire, is used to make connections on the diagram. While the wiring tool is idling over the subVI icon, the help window appears showing where the connections are.

![Figure 6. A Diagram Reference to a subVI in G](image)

Multiple instances of a VI icon within a diagram can either execute multiple instances of the VI in parallel or be enqueued to run serially. The programmer designates the desired behavior after taking into account any side effects of the VI. All the examples thus far have been purely functional, or without side effects. Real-world programs, on the other hand, have side effects, the most obvious being file input and output. Side effects are abundant in laboratory automation programs in the form of analog-to-digital converter inputs and digital-to-analog converter outputs, so a system must have built-in functions and VIS with side effects. The G sequence structure can order the execution of side effect VIS when there is no explicit data dependency between them. In general, it is impossible to tell whether a VI with side effects will conflict with another instance of itself running in parallel, so the programmer must designate serial or parallel execution of multiple calls for each VI.

Implementation of a Visual Language System

LabVIEW contains three interrelated editors, one for each of the three parts of a VI—the block diagram, the icon, and the front panel. A diagram is constructed by selecting built-in functions, structures, and previously constructed VIS from graphical palettes and arranging them in the block diagram window. The arcs are drawn using a wiring tool for the mouse cursor. As each edit transaction is performed, the syntax checker detects and flags any cycles introduced into the dataflow graph, propagates data attributes to all the
terminals, computes the data type for each instance of a polymorphic built-in function, and redraws any arcs whose attributes changed. Arcs are drawn with a distinctive pattern, width, and color to indicate the data type, array dimensionality, and numeric representation.

The main source and sink terminals are not directly installed in the diagram, but rather appear automatically as a result of installing controls and indicators on the front panel. The front panel can be thought of as a collection of stylized data declarations for the diagram input and output parameters. Because a VI has a front panel in addition to an icon, every VI within the program hierarchy can be run programmatically via its icon or interactively via its front panel user interface. This is a valuable aid for debugging and troubleshooting. Because the front panel user interface is separate from the diagram, the programmer can optimize both the panel and the diagram, each for its own particular purpose—the front panel for operational simplicity and logical grouping of data, and the block diagram for program clarity. Figure 7 shows a complete VI for measuring frequency response. The front panel and icon of the VI are shown in a LabVIEW window. The block diagram of the VI, shown below the front panel and icon, calls a function generator VI, FGen, which generates the stimulus, and a voltmeter VI, DVM, which measures the response from the device being tested. The dotted line between FGen and DVM is a dummy Boolean data flow introduced as an artificial data dependency so that the DVM will not execute until FGen completes. A sequence structure would accomplish the same thing.

![Figure 7. Frequency Response VI Panel and Diagram](image)

### Performance

The original version of LabVIEW incorporated an interpreter and had performance comparable to interpreted BASIC. LabVIEW 2 incorporates a compiler that generates code with performance comparable to that produced by a C or Pascal compiler. We have achieved this performance after several re-implementations of the execution system. The LabVIEW prototype, a particularly literal implementation of the dataflow model, actually allocated, propagated, and deallocated data tokens. The next implementation statically allocated tokens where possible but still copied data values excessively. Subsequent implementations have used ever more sophisticated heuristics to eliminate data copies by analyzing the relationship of a node's outputs to its inputs in the presence of arc fan-out. For example, in most situations the data token for the output of the negate built-in function can be statically assigned the same memory location as the input token. A sufficient but unnecessary condition for this in-place relationship is to hold no fan-out of the input arc to the negate node. This in-place relationship is not too significant by itself, but propagating such relationships hierarchically up through loop structures can result in substantial savings in memory space and processing time.

### Future Directions

Several areas have been identified in which G is more cumbersome than we would like. For example, consider a programmable thermostat where the program is a loop that runs each hour to establish a new temperature setpoint, and another control loop runs every 5 sec comparing the actual temperature to the setpoint, turning the heater on or off as required. An attempt to implement this in G as two parallel loops with the setpoint data flowing between them will encounter two difficulties. First, because both loops run concurrently, there can be no explicit data dependency between them, or one loop would not start until the other completed. Second, even if there were an explicit data flow, it would have to be nonconservative in the sense that a token would be consumed many times for each time it was produced.

There are many ways around the problem, but most either compromise the modularity of the two loops (keeping them separate permits substitution of one without affecting the other) or obscure the relationship between them, namely the shared setpoint value. Currently, our favorite solution employs a common subVI that contains persistent data. One loop calls the subVI to write the data, the other calls it to read the data and the data persists beyond the completion of the subVI. In a sense, the subVI is simulating a global variable. The best way to deal with persistent data in a dataflow system still eludes us.

For real-time applications, more explicit control of scheduling is desired. There is currently no method for specifying that a particular data path or VI is to be given priority over other parallel activities. There is also no way to specify or enforce deadlines.
Because dataflow diagrams are inherently parallel, it should be straightforward in principle to distribute a diagram over multiple processors. In practice, we have just begun experimenting. Our approach is to provide explicit control over the distribution before any attempt is made to automatically distribute the diagram. The absence of side effects, including shift registers, within the body of a for loop means that multiple iterations could proceed in parallel, but there is currently no attempt to detect or exploit this.

As mentioned before, the front panel can be thought of as a collection of data declarations for the inputs and outputs of a VI. For each data type, there are many styles for displaying and editing the value(s). The simple data types have styles that are easily recognized as controls and indicators similar to those of typical instruments and appliances. More complicated data types can have much more elaborate display styles, such as contour plots, colored surfaces, and so on. In fact, the front panel is an excellent way to integrate interactive data visualization with programmatic data acquisition and analysis.

The case and sequence structures have stacked, multiple diagrams to preserve screen space. An alternative depiction simultaneously showing all the subdiagrams side-by-side is probably needed for additional clarity, especially for the common situation of two case subdiagrams corresponding to the if-then-else programming construction. Finally, we feel that it is desirable to define the formal semantics for G so that it can be effectively compared with other programming technologies and enhanced in a consistent way.

Conclusion

LabVIEW has been in use since October, 1986. The system has been successfully used to program instrumentation systems in areas such as process control, medical monitoring, production testing, and simulation. Users report that they are much more productive in the ability to construct programs, and are able to adapt programs to changes in requirements in a small fraction of the time used with other programming systems. However, there is also a noticeable learning curve associated with developing a dataflow programming perspective. Experienced control-flow programmers seem to have slightly more difficulty adjusting to dataflow programming than nonprogrammers, although it is not clear why.

Appendix: Summary of G

G is a strongly typed, structured dataflow programming language. The elementary data types are numeric (integer and floating point), string, and Boolean, and there are two aggregators—array and cluster (similar to a Pascal record or C struct). A module in G is called a Virtual Instrument (VI). A VI consists of an icon, a panel, and a diagram.

A panel contains controls and indicators that define the data types of the inputs and outputs of the VI; for example, a slide control represents a numeric input, and a plot represents an output that is an array of points, where a point is a cluster of two numerics.

An icon contains terminals (non-overlapping subregions) that are in one-to-one correspondence with a subset of the panel controls and indicators.

A diagram is a directed, acyclic graph containing nodes, interconnecting signals, and source and sink terminals which correspond to the panel controls and indicators, respectively. Nodes have zero or more input terminals (sinks) and zero or more output terminals (sources). A signal connects a single source terminal to one or more sink terminals.

Nodes are built-in primitive functions, icons of other VIs representing calls to the VI (recursion is not supported), or structures. Structures contain one or more subdiagrams; sink terminals on the structure are source terminals in the subdiagram and vice versa.

Execution of a diagram or subdiagram begins by placing tokens on all the isolated source terminals, which correspond to panel controls or sink terminals of a structure. The tokens propagate along the signals, duplicating themselves if the signal splits.

A node executes when each wired input has a token. A default token is generated for each unwired input. A node consumes all the input tokens and produces a token for each output, which then propagates along the signal.

A diagram finishes execution when all its nodes have completed and all tokens have propagated to the isolated sink terminals, which correspond to panel indicators or source terminals of a structure. When a diagram is complete, every node has executed exactly once and every signal has carried a single token (apart from fan-out duplications).

A node that is a VI icon executes by placing its input token values on the VI panel controls and executing the VI, (by executing the VI diagram). When the VI completes, the indicator values are placed in the output tokens of the node and the node completes. A VI is typically serially reusable, although it may be parallel, in which case the VI replicates its data space for each call.

A node that is a structure executes according to the semantics of the structure. A for loop structure executes its single subdiagram zero or more times where the count is set explicitly by wiring a value to the input, or implicitly from the size of the smallest input indexing array. A while loop executes its single subdiagram one or more times. A case structure executes exactly one of its two or more subdiagrams based on the value wired to the selector input. A sequence structure executes each of its subdiagrams in order.