LabVIEW

JEFFREY KODOSKY*, National Instruments, USA

Shepherd: Crista Lopes, University of California, Irvine, USA

LabVIEW™ is unusual among programming languages in that we did not intend to create a new language but rather to develop a tool for non-programmer scientists and engineers to assist them in automating their test and measurement systems.

Prior experience creating software for controlling instruments led us to the perspective that the software ought to be modeled as a hierarchy of "virtual instruments". The lowest level virtual instruments were simply reflections of the individual physical instruments they controlled. Higher level virtual instruments combined lower level ones to deliver more complex measurements. A frequency response virtual instrument could be implemented using a voltmeter and a sine-wave generator inside a loop that stepped through a frequency range. This was mostly an abstract concept at the time because it was hard to imagine how an existing language or tool could provide the rich yet intuitive experience of using a real instrument.

Inspired by the first Macintosh computer, we quickly realized the graphical user interface would be a natural way to interact with a virtual instrument, but it also sparked our imaginations about using graphics for creating software at a higher level of abstraction.

The February 1982 issue of IEEE Computer was devoted to data-flow models of computation, and it convinced us that graphical data-flow diagrams needed to be part of the solution. The major difficulty we saw, however, was the need to use cycles in the data-flow diagram to represent loops. Cycles increased complexity and made diagrams hard to understand and even harder to create.

This concern led to a major innovation in creating LabVIEW: merging structured programming concepts with data-flow. We represented control-flow structures as boxes in a data-flow diagram. We knew how to reason about loops, so we could introduce them as first class elements of the graphical representation rather than being constructed from lower-level elements. A box could encapsulate the semantics of the iterative behavior; it could clearly separate the body of the loop (the diagram inside the box) from the code before and after the loop (the diagram outside the box); and, its boundary could hold iteration state information.

Those fundamental concepts of "graphical", "structured" and "data-flow" enabled us to propose a software product. We staffed up a small skunkworks team to implement it. We called it LabVIEW. It was to be an engineer's tool for automating measurement systems. At first, we were reluctant to admit that we had created a graphical programming language. When we finally did, we nicknamed it G, for Graphical language, so we could talk about the language as distinct from the integrated development environment (IDE), LabVIEW. In practice, almost everyone refers to both the language and the IDE as LabVIEW.

Without intending to do so, we created a programming language radically different from those that came before, pioneering techniques of graphically creating and viewing code, eliminating manual memory management without adding garbage collection overhead, and anticipating the massively parallel systems of the modern era. LabVIEW continues to evolve and thrive after more than 30 years.

*Co-founder and Fellow, National Instruments

Author's address: Jeffrey Kodosky, R&D, National Instruments, 11500 N Mopac, Austin, Texas, 78759, USA, jeff.kodosky@ni.com.

This work is licensed under a Creative Commons Attribution 4.0 International License.
© 2020 Copyright held by the owner/author(s).
2475-1421/2020/6-ART78
https://doi.org/10.1145/3386328

CCS Concepts: • **Software and its engineering** → **Visual languages**; **Integrated and visual development environments**; **Data flow languages**; • **Applied computing** → *Physical sciences and engineering*; • **Human-centered computing** → *Graphical user interfaces.*

Additional Key Words and Phrases: LabVIEW, data flow language, graphical programming

ACM Reference Format:
Jeffrey Kodosky. 2020. LabVIEW. *Proc. ACM Program. Lang.* 4, HOPL, Article 78 (June 2020), 54 pages. https://doi.org/10.1145/3386328

Contents

1 INTRODUCTION

The history of LabVIEW™ [1] is unusual for programming languages because the goal of the project was not to create a new language. Our goal was to develop a tool to assist non-programmer scientists, engineers, and technicians to automate their test and measurement systems. The state-of-the-art had progressed to the point where most instruments, e.g., voltmeters, function generators, etc., were easily connected to computers using standard interfaces, but developing the software for controlling the instruments in the measurement system remained challenging.

Our slogan became "to do for test and measurement what the spreadsheet did for financial analysis." Spreadsheet programs allowed financial analysts to quickly and easily create custom programs for modeling various "what-if" scenarios without training to be professional programmers. In similar fashion, our tool would enable scientists and engineers to quickly and easily create custom programs for their test and measurement systems, without needing to enlist the aid of programmers.

In the first section we describe the influences and ideas which ultimately lead to the creation of LabVIEW. When version 1.0 was released as a product, we billed it as a "non-programming" way to create test and measurement software, even though internally we knew it consisted of a rigorous

[1] LabVIEW™ is a trademark of National Instruments

and well-defined programming language. Internally, we distinguished the language, "G", from the editor and integrated development environment (IDE), LabVIEW, but for a long time we did not make the distinction externally because we were more interested in promoting its usability as a tool, rather than promoting the computer science behind the language.

When we do specifically talk about the language we refer to it as a "graphical, structured data-flow" language, and this is the description we use in this paper.

- Graphical means that it is represented as a 2-dimensional diagram consisting of nodes and interconnecting wires, rather than as text.
- Structured refers to the use of boxes, called structure nodes, to represent various control structures, e.g., loops and conditional execution. Structure nodes have a diagram inside the box and they control whether and when to execute this inner diagram. The inner diagram executes with the same semantics as a diagram outside the box. Structure nodes can be nested just as control structures in text-based languages can be.
- Data-flow indicates the semantics of the language. The G diagram represents a data-flow model where nodes perform computations based on their input values. When they complete they propagate their output values along the wires to downstream nodes. A node executes when all of its inputs are available. The data-flow model is particularly simple: the diagram is an acyclic graph where each wire carries a single value. There is no queuing or buffering. In recent years, the model has expanded to support higher-level abstractions (described in later sections). From the outside, structure nodes behave as any other node behaves: all inputs musts be present before the node executes, and all outputs are produced when the node completes. On the inside, a structure node contains one or more (sub-)diagrams. The structure node chooses when to execute a sub-diagram, but the sub-diagram operates with the same data-flow semantics as everywhere else. By using structures, the G language can represent iteration while keeping the data-flow graph acyclic and the node firing rule simple and uniform. We believe that the resulting diagrams are thus much easier to construct and analyze while retaining or exceeding the expressive power of other languages.

LabVIEW continues to evolve since its introduction in 1986. Many improvements are basically completing the capabilities implied by the original language and environment, but there are also important advances in the language itself. Part of the success of LabVIEW is attributable to its focus on test and measurement applications, and the concomitant close integration with I/O hardware and precision timing. Unifying the representation of functional behavior with I/O configuration and timing and distributed execution across CPUs and FPGAs is a strength of LabVIEW as well as a fertile area for additional research.

Organization: This paper is organized into five major sections in a mostly chronological order. For a description of just the semantics of the language itself, read sections 3 and 5. Section 2 covers the background and motivation for starting the project, the original goals, and the inspirations leading to the conception of LabVIEW. It also covers the skunk-works development and implementation of version 1, as well as the reaction from the early users, and an assessment of the abilities and limitations of both the language and the environment at that time. Section 3 is a detailed description of the operational semantics of LabVIEW/G as it existed in version 1. Section 4 continues the history by describing the motivations and goals for version 2, the development efforts and delays, and the response from users. The section contains a brief history of the major milestones of subsequent versions as development continued over the years. Section 5 is a description of the semantics of the language features introduced in version 2 and later versions. Section 6 discusses ongoing

research and possible future directions for LabVIEW. The paper ends with a retrospective section and summary section.

Sources: For the most part, I have relied on personal recollection to recount the history of LabVIEW. Ever since LabVIEW was introduced, I have been asked many times about how I came up with the idea for it, so I have had a lot of practice telling the story to individuals, groups, in interviews and at internal conferences, in more or less detail, depending on the stamina of the listeners. I did keep a notebook which shows the early journey to the concept as well as a few more years of questions and ideas, after which I kept notes and diagrams in documents on my computer. Many of these are in formats no longer readable with modern programs, but can still be read on working older machines. Two developers from the start, Rob and Steve, still work with me so they were able to corroborate my memories and provide additional details. The most comprehensive account of the history is in Gary Johnson's book [Johnson and Jennings 2001]. It derives from interviews he conducted with me and other developers in the early 90s. Over the years, we explained, re-visited, and defended design decisions repeatedly, and this is reflected in various internal design documents. As a result, the history of LabVIEW has remained vivid in my memory.

The Austin History Center interviewed me in 1997 about the origin of National Instruments and LabVIEW. I have a rough transcript of the interview but not the edited one.

2 HISTORY - PART 1: ORIGIN OF LABVIEW

2.1 Background

National Instruments was founded in 1976 by James Truchard, William Nowlin and Jeffrey Kodosky, the author of this paper. Jim took on the role of CEO. At the time of founding, we were all employees of Applied Research Laboratories (ARL) of the University of Texas at Austin (UT) working on various underwater acoustics projects for the US Navy. That work culminated with our implementation of a large automated test system, FQM-12 [Truchard and Ashby 1978], designed to test any underwater transducer. Tests involved measuring short high-powered pulses (the pings you hear in submarine movies), beam-patterns showing directional sensitivity, response versus frequency, and so on. It was necessary to build custom interfaces between the computer (PDP-11), various instruments (waveform generator, attenuator, power amplifier, oscilloscope, A/D converters), a relay panel, and a motor for rotating a transducer or transducer array. My responsibility was to design and implement the software to control the devices, run the tests, and produce the reports. The work was completed in three years with the help of a small team.

I had begun working part-time at ARL in 1973 while a graduate student in physics. I had had experience programming in Fortran and Basic, though I didn't have a particular interest in computer science. My first job at ARL involved writing a variety of programs for a PDP-8, and the hands-on experience with that machine piqued my interest. Later, our group at ARL acquired a PDP-11 and I became very proficient programming it in assembly and then using C when we acquired a copy of Unix. I studied the operating system using John Lions' source code and commentary books. I became intrigued enough to take a graduate course in operating systems. Shortly afterwards, I was exposed to LISP and became so fascinated with it that I switched my degree program from physics to computer science. And just for fun, I wrote a LISP interpreter for the PDP-11 and submitted it to DECUS, the Digital Equipment Corporation Users' Society.

In the early 1970's the Hewlett-Packard (HP) company invented the General Purpose Interface Bus (GPIB) and subsequently submitted it to the IEEE, where it became the IEEE-488 standard [IEEE 1988]. This bus on a cable enabled HP to connect as many as 14 instruments to one of their "programmable calculators" facilitating the building of automated test systems. By the mid 1970's more

instrument manufacturers were embracing the GPIB standard. Jim, Bill, and I saw an opportunity to start a company building a GPIB interface for the PDP-11. We continued working full-time at ARL until 1980 to complete the FQM-12 while "moonlighting" to build National Instruments.

2.2 Motivation

Manufacturers must test their products before shipping them to ensure they are working properly, so there is typically a test station at the end of an assembly line where each product is subject to various tests and measurements. The extra expense of this activity is a strong incentive to automate it, and the introduction of the GPIB made than much more practical.

By the early 1980's National Instruments was a successful company building GPIB interfaces, driver software, and related products for a variety of mini and microcomputers. In order to continue the growth that we were enjoying, we realized we needed to expand the scope of our business. We considered many options, from building custom instruments, to creating software for generating engineering graphs, to building single-board computers.

In the end, we decided it would be best to focus on the next level challenge in constructing automated test and measurement systems, namely writing the software, because that was most synergistic with our existing business. The GPIB enabled our customers to easily connect their instruments to their computer, but it still proved challenging to create and maintain the software that controlled the instruments and acquired the necessary measurements.

Most of the test and measurement programs of the time were custom programs written in BASIC or C by the engineers or technicians in the manufacturers' test departments. Many test engineers used an IBM PC with a few bench-top instruments and a program written in BASIC. A simple system for testing an amplifier might have a programmable function generator to provide a signal to the amplifier, an oscilloscope to view the output waveform, and a digital multimeter to read the output amplitude. A program to automate the testing would start out simple enough: output GPIB commands to set the function generator to produce a sine wave of a particular frequency and amplitude, output GPIB commands to set the range of the multimeter and then read the amplifier's output amplitude; repeat these operations for successive frequencies or amplitudes in order to characterize the amplifier response or linearity.

Over time, the test program would grow in complexity and obscurity as more options were added, and more error conditions were detected and handled, and various interactive user inputs were added in order to pause and resume, set configuration parameters, and so on. When a new test engineer eventually took over, the typical process would be to toss the old unmaintainable program and start from scratch writing a new test program. More sophisticated test scenarios involved a rack of instruments and a rack mounted minicomputer programmed in C. A test engineer often needed the support of a team of computer programmers to implement test programs. But the requirements from the test engineer could conflict with the design principles of the programmers. Software modularity could often be at odds with the performance needs of the test engineer.

We started National Instruments by building GPIB interfaces to more easily connect instruments with mini and micro computers. But by helping GPIB become more widely adopted we were also helping to exacerbate the software problem we now sought to address.

Jim decided that I should move to a small office near UT for easy access to the libraries and away from the corporate office, so that I could remove myself from the daily activities and focus on this new initiative. While it wasn't clear where to start, we did have a couple of examples that could serve as comparison benchmarks.

The first was the Asyst DOS-based software product. Asyst combined the FORTH language with libraries and tools targeted to the scientist and engineer. While the combination integrated multiple

components for the benefit of the user, it didn't seem to address the fundamental programming issue, as we understood it.

The second, and more inspiring, example for us was the electronic spreadsheet. Paper spreadsheets had been used forever in financial reporting—the concept was natural and ingrained even if preparation could be tedious and error prone. Electronic spreadsheets eliminated the disadvantages. Calculation was automatic and fast—changing a value would propagate throughout the spreadsheet and show the result instantly. A financial analyst could quickly do "what-if" calculations to see how results depended on interest rate, for instance, without having to enlist a team of programmers to write a custom program. The computation and analysis were completely under the control of the domain expert.

We defined our goal: create a domain-specific tool usable by scientists and engineers which would dramatically improve their ability to build automated test and measurement systems. We captured that in the phrase "to do for scientists and engineers what the spreadsheet did for financial analysts". It was motivating, and it defined the scale of impact we sought, but it wasn't a roadmap for what to do.

2.3 Conception

To help brainstorm on the topic, I asked a friend from college days to join me as a consultant for NI: John "Jack" MacCrisken. Jack and I graduated from Rensselaer Polytechnic Institute (RPI) in 1970 with BS degrees in Physics. I continued with graduate study in Physics and Computer Science at the University of Texas at Austin without completing a degree, while Jack continued by getting an MS in Computer Science and Electrical Engineering at Stanford. He had a lot more experience with large software applications, so I hoped his insights would prove helpful.

One of the first things Jack and I did was to visit half-a-dozen customers to learn more specifics about their test software development and maintenance processes. What we learned basically validated our understanding of the issues. When pressed for thoughts on how to improve their situation, they invariably suggested incremental improvements to their libraries and tools. There were no breakthrough ideas for dramatic improvements.

Jack and I continued to meet for several days every few weeks. Jim would join us for part of the time to vet our latest ideas and offer suggestions and encouragement. For completeness we considered creating measurement-specific libraries, and even using a spreadsheet itself as a means for describing a test program. While simple scenarios were manageable, more complex test and measurement scenarios were not. Existing programming approaches sufficed for simple scenarios and there was nothing compelling about solving the simple applications in yet another way. We wanted our tool to handle the complex scenarios.

Many of the interesting test scenarios were stimulus-response measurements: a stimulus signal is applied to a device and the output signal is measured. The measurement is repeated with a varying stimulus to generate plots of gain versus frequency, or output versus input amplitude, or amplitude versus direction, and so on. From these and other examples, we concluded that representing looping/iterative behavior was a key aspect of any solution we would propose.

I recalled an idea from my experience with the FQM-12 software. Both operator-level shipyard mechanics as well as designer-level acoustics engineers and acoustics researchers used that system. At the highest level, the system was highly constrained, allowing only designated measurements to be made based on the model number of the transducer being tested. But acoustics engineers and researchers could access the system at lower levels to run more varied and complex tests using more detailed control of parameters and instruments. These levels of access formed a hierarchy of software interfaces to the instrumentation, and therefore I thought of the FQM-12 software as a hierarchy of "virtual" instruments.

There was discussion in the academic literature about virtual instruments, but the focus seemed to be on object models with inheritance, where an instrument class could be specialized to a more limited subclass. At the time we felt that using object-oriented programming techniques would require more expertise of our customers rather than less. It was hard to imagine how existing languages could provide as rich and intuitive experience as using a physical instrument and even harder to imagine a reasonable experience for constructing a higher-level virtual instrument by composing simpler ones.

Then came the summer of 1984. My brother-in-law sat me down in front of his Macintosh computer, showed me how to use the mouse and opened MacPaint. It was a revelation. I immediately went out and bought a Mac (prior to that time I had not been interested in personal computers because, as a Unix brat, I wanted larger memory and disk space, and more horsepower to run larger programs). I was convinced the Mac represented the future of human-computer interfacing.

The Mac's graphics provided an obvious way for software instruments to reflect the physical instruments we worked with. The virtual controls on the computer screen could be just like the controls on an instrument, except more flexible. Now it was clear that a virtual instrument should have a graphical interface on the computer screen. When I played with MacPaint I never opened the manual—I discovered how to use it by trial and error. It turns out most engineers learn how to use physical instruments the same way—skip the manual and try making measurements until you get it right. With such a rich interface using mouse gestures and a menu bar, the possibilities for learning by experiment were enormous and should even be fun.

My enthusiasm for the Mac graphical user interface was tempered by the fact that writing a program that had the capability and responsiveness to support such an interface was vastly more difficult than writing a test and measurement program. Still, a graphical user interface (GUI) was too good to ignore so we started to brainstorm how we might make implementing test and measurement programs with GUIs much simpler.

After a couple months of study that wasn't very promising, I had the naiveté to suggest that maybe I could come up with a way to use the graphics to construct a general program easily—in other words, to program graphically. Jack endorsed the idea. Jim was dubious but agreed to let us take a few months to investigate. We all knew there were graphical notations for programs (the GPIB is defined in terms of state diagrams), but whether we could come up with one that would be easier to use than just writing C code was the big unknown.

Jack and I looked at a variety of diagram types and at Jim's suggestion even worked out in some detail what a flow-charting approach might look like. We also evaluated programming using only state diagrams. Neither approach helped. We had known about data-flow diagrams, but it wasn't until we came across the February issue of IEEE Computer that we took an in-depth look. The entire issue was devoted to data-flow programming, including data-flow machines [Gajski et al. 1982].

Data-flow diagrams were very appealing. They were easy to understand and construct—until the program needed a loop. Programs that needed iteration had to resort to cycles in the data-flow graph, and that's when things got much more complicated. As if to emphasize that point, one article [Davis and Keller 1982] featured a full-page, relatively simple diagram with a cycle implementing Newton's method for finding a zero of a function. It had a bug in it! Entering and exiting the cycle was remarkably easy to mess up. A program requiring nested loops would be a big challenge to understand, let alone construct correctly.

While we investigated multiple other types of diagrams, I kept coming back to data-flow because of its elegance and simplicity. But I was frustrated with the need for cycles to represent iteration, as iteration is an important part of most test and measurement programs. I was stuck, but I didn't want to give up. I resorted to sketching random ways to represent the concept of a loop—two

kinds of loops, one for sweeping a stimulus through a fixed range, and another for tuning until a parameter was within some limit. A favorite sketch was a box with a control panel to govern the iterating behavior, though I didn't know what to make of it.

Somewhere along the line, it dawned on me that a box in a data-flow diagram could represent a loop. Inside the box is the body of the loop—a standard data-flow diagram which would execute repeatedly. From the outside the box would appear to be just like any other node. Iteration would be represented without the need for cycles. There would be no way to confuse the body of the loop with any other part of the diagram (unlike a diagram with cycles). In computer science we know very well how to reason about loops, so why shouldn't a loop be a first-class data-flow language element?

A cyclic data-flow diagram is hard to read, first because wires run in all directions, and second because it requires special nodes that can execute without having data on all their inputs and that can complete without providing data on all their outputs. Using a box to represent a loop means that all nodes would have a simple and uniform firing rule: all inputs must be present before executing, and all outputs are produced upon completion. And wires could all flow in one direction: from the start of the program to the end.

The box representing a loop could have a special terminal on the outside for specifying the number of iterations. Alternatively, it could have a special terminal on the inside for indicating the iteration just completed is the last one. After all iterations complete, then the loop itself completes.

One other item is needed to complete the picture. A cycle in data-flow not only effects the transfer of control to re-execute the nodes in the cycle, it also transfers a datum from the end of one iteration to the beginning of the next. This can be accomplished in the box notation by creating a pair of terminals on the left and right edges of the box. We called the pair of terminals a shift register. When one iteration provides a value to the right-side terminal, that value will appear on the left side terminal at the start of the next iteration. Such a construction is effectively an implicit cycle, but it is bound to the box and highly constrained. It is impossible to mess up creating such an implicit cycle because there is no place to connect to it except at the shift register terminals.

It was a small step to see that other control structures we are familiar with from traditional programming languages, e.g., if statements and switch statements, can also be represented as boxes in a data-flow diagram. In these cases, however, the structure contains multiple distinct internal diagrams; the structure selects one to execute based on the value supplied to a special terminal on the outside of the box.

Data-flow diagrams are inherently parallel. The parallelism was intriguing to us but not important for the measurement programs we wanted to write. However, it is necessary to properly order both commands to instruments and other I/O operations, so we needed one final control structure to force serialization when needed. The sequence structure contained multiple sub-diagrams; each was executed when the previous one completed. We thought such a structure would be more convenient than creating dummy data-flow connections to control execution order.

Once Jack and I had the basic elements of our structured data-flow representation I decided to try it out on some of our engineers. I created a diagram to represent our prototypical stimulus-response test and asked if they understood what was going on. Did they think they could construct such a diagram using something like a paint program? I was disappointed and disheartened seeing them struggle to understand. I thought my boxes would make it so much easier!

After stewing about it for a week, I thought maybe my boxes were too nondescript, so I created more stylized graphics: a pad of paper for a `For` Loop, a rounded rectangle with an arrow head for a `(Do-)While` Loop, and, at Jim's suggestion, a film strip for the Sequence structure. The only one that remained a nondescript patterned box was the Case Select structure because I couldn't think of anything more distinct (and I still can't today).

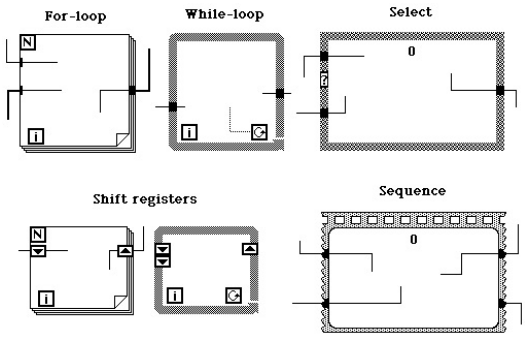

Fig. 1. Original data-flow control structures
The small rectangle on the border of a structure where a Signal crosses is called a Tunnel. A Tunnel holds onto a set of Terminals, one that connects to the outside, and one per sub-diagram which connects on the inside. A thin line represents a scalar value while a thicker wire represents an array. A solid line represents a numeric type while a dotted line represents a Boolean type. A For-Loop will optionally index an array at an input Tunnel and accumulate an array at an output Tunnel. At the start of the next loop iteration, the value on the right side of a shift register is shifted to the top left and the left values are shifted down one.

The result astonished me. A simple change in the graphics (which was still only in black and white) had a dramatic effect on the engineers: they were much more interested and excited by the diagrams and could easily imagine constructing similar ones.

Figure 1 was created using LabVIEW 1 and shows the elements of the structured data-flow representation as it stood then.

We were excited about the representation. Although we knew we were creating a new programming language, we never talked about it as such for fear of provoking adverse reactions within the rest of the R&D organization: why were we creating yet another language when we were supposed to be creating a tool analogous to the spreadsheet? Our rationalization was that the representation was simple enough that even non-programmers would be able to use it effectively.

A fair question to ask is why we thought the flexibility of a programming language was needed, as opposed to a forms-based configuration environment with common measurement patterns predefined. One reason came from my previous programming experience—when automating measurement hardware, I always found some complication requiring adjustments to timing or data formatting. Providing sufficient capability in a configuration environment to satisfy all possible needs results in so much fine-grained configuration that it effectively creates a language anyway. Another reason came from observing the futility of trying to standardize instruments—core functions could be standardized, but there were so many useful variations and optimizations that a complete standard was infeasible. The final reason came from basic engineering—it is possible to build higher-level fixed patterns on top of a flexible base, so we should build the flexible base first.

With the programming question resolved, we proceeded to flesh out the remainder of the virtual instrument (VI) concept. The GUI controls and indicators on the "front panel" of the VI had corresponding terminals on the "block diagram" that were the sources and sinks for data, respectively. The front panel of a VI corresponded to the front panel of a real instrument while the block diagram of the VI corresponded to the internal circuitry of the instrument. From a computer science perspective, we viewed the panel as a procedure declaration while the diagram was the procedure implementation.

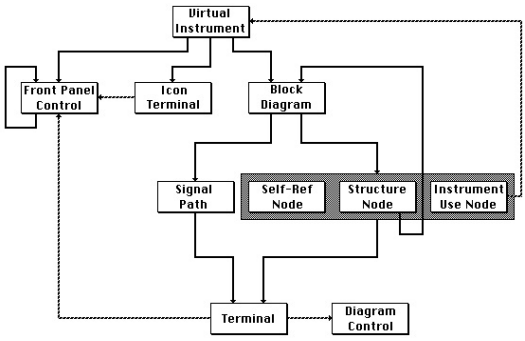

Fig. 2. Data structure diagram
The Self-Ref Node is an invisible node created to hold on to the source and destination terminals of a diagram. A Node owns terminals while a Signal has a list of references to Node Terminals, where the first in the list is the source terminal. The graphical path of a Signal is called a Wire although Signal and Wire are often used interchangeably. (The solid line represents a 1-to-n owning relationship and the dashed line represents a 1-to-1 reference relationship).

We needed a way to compose VIs so that we could build an application as a hierarchy of VIs. We introduced the concept of an icon to graphically represent a VI. Every VI had an associated icon and "connector pane", the latter being a collection of input and output terminals on the perimeter of the icon corresponding to the controls and indicators, respectively, on the panel. VI Alpha can call VI Beta by placing Beta's icon in Alpha's diagram and connecting wires—the icon in the diagram represents a "sub-VI call".

At this point we had a concept, and the next step was to turn it into a software architecture. Jack suggested we create a data structure diagram to represent the main elements of the system and how they related to each other. An example of a data structure diagram, drawn in MacPaint, is shown in Figure 2.

Each box in the figure represents a data structure and lines represent ownership of other data structures. A solid line meant one or more, a dashed line meant reference-to rather than ownership. We grouped the different kinds of Nodes inside a larger box to show they had related properties, e.g., ownership of a set of terminals. Though we didn't know it at the time, we were effectively representing single inheritance via enclosure.

With this data structure diagram, we then talked through the various transactions we could imagine doing on an instance of a VI hierarchy using an editor, e.g., adding a new node to a diagram, moving a node, connecting two terminals with a wire, and so on. This process helped us identify individual data items that were needed in the various data structures, e.g., bounding rectangles for nodes, paths for wires, etc.

We also talked through how execution would work. We imagined a brute-force token-passing mechanism, the interesting parts being the behavior of the structures and calling a subVI. When a shared subVI is called concurrently from two parallel branches, it is necessary to serialize access to it.

2.4 Implementation

Once we had convinced ourselves we had a reasonable data structure architecture and plausible sketches for algorithms that operated on the data structures then we were ready to build a product.

The initial plan was to develop a prototype for the Macintosh to prove the concept, and then develop the shipping version on the IBM PC where we expected the majority of the market was.

We decided we would develop the software on the Macintosh so we could become intimately familiar with the machine.

In the summer of 1985, in order to not disturb the day-to-day operations of the company, we established the development effort in a one-room, 850-square-foot, windowless office that we found near the UT Austin campus. It was the perfect size and location for our skunkworks project. We purchased several Macintosh machines and immediately voided the warranties by having Hyperdrive hard disks installed—it would have been impossible to do any development on a Mac without one.

Over the next year nine people joined me on the development team:

- Jeff Parker, a new UT CS graduate was the first person to join me;
- Rob Dye, a UT CS masters student who started as a summer intern;
- Cathie Wier, already at NI working on GPIB drivers, joined the team and continued to pursue a UT CS graduate degree;
- Lynda Pape, a UT CS undergraduate student who started as a summer intern;
- Eduardo (Lalo) Perez, a UT EE graduate who eventually used LabVIEW in his PhD dissertation on image processing;
- Paul Ponville, a MS CS graduate from University of Louisiana, Lafayette, and the only person who actually studied the data-flow model of computation as a student;
- Steve Rogers, a recent UT CS graduate already at NI working on real-time GPIB drivers, joined the team;
- Ken Rozendal, a UT CS graduate student working part time; and
- Rosy Mehrotra, a new UT CS graduate.

We had a rudimentary network to share a Corvus disk which contained the master copy of the source code. There was no source code control, so we had a physical semaphore: a Tupperware bowl, the possession of which designated the developer who had exclusive access to the master copy for merging changes.

We were designing a highly interactive application while at the same time discovering how to program the Macintosh with its object-oriented toolbox. We were naïve enough to believe we would be successful, and passionate enough to vigorously debate the intricacies of the user interface and user experience. We endorsed the WYSIWYG (what-you-see-is-what-you-get) perspective (meaning in our context that a diagram computed what it appeared to compute), and we focused on ease-of-use because we believed that our users' time was more valuable than our own.

Screen real estate was very limited, and we struggled to make diagrams understandable without using space on lots of text labelling. Consistent use of recognizable glyphs was one technique, and more dynamic use of screen area was another. For example, overlaying the frames of a sequence structure and the cases of a case structure saved a lot of screen space at the cost of not being able to see all the code at once. The disadvantage of not being able to see two cases of a case structure simultaneously was at least partially offset by being able to see each case in the context of the surrounding diagram.

Aesthetics played an important role in our debates. Early on, we decided that we would implement Manhattan-style wiring even though it was more difficult than point-to-point (diagonal) connections. Manhattan-style appeared neater because it didn't have the pixelation of slanted lines, it worked better when a wire split to two or more destinations, and it fit tighter when routing a path in a compact diagram. We wanted to minimize extraneous "noise" in diagrams so we decided not to have any arrows at wire destinations but rely on the natural tendency to layout structured data-flow

from left to right. We opted to have a simple gap where wires crossed, and no gap where they connected (years later we added the option to draw dots at the junctions to further distinguish them from crossings; it is widely used, though some users still consider the dots to be too much noise).

We were building an interpretive system, so we never made a clear distinction between the language and the editing and execution environment. As the user interacts with the editor, LabVIEW creates an internal parse tree that it then renders on the screen. That same parse tree data structure is used by the interpreter to execute the diagram (today it is input to the compiler).

We knew users would want printed diagrams for documentation purposes, so we implemented printing capability. However, we didn't think paper was a good way to view diagrams. We believed the editing environment provided a much better and more interactive way to navigate and explore diagrams. In modern terms, we were recommending a browser-like experience as opposed to a book-like experience.

In any event, we never considered parsing a printed diagram, or a diagram created with a paint program, to be a worthwhile effort. A G diagram exists internally as a parse tree by virtue of its interactive construction, so there is no need for any parsing.

We thought about trying to ensure a diagram was always syntactically correct as it was being edited, much like syntax-directed editors of the time, but we quickly realized it would not be convenient for a user. So, we decided to provide immediate feedback whenever an error was detected and prevent the VI from executing until the errors were corrected. After every edit operation, such as adding or deleting a node or wire, a fast type-propagation algorithm is run to determine the direction and type of all signals and to verify there are no cycles (which would deadlock execution).

The summary indication of an error was a "broken run arrow" preventing the VI from being executed. In addition, many errors were also indicated by a "broken wire". If data of the wrong type was wired to a node, that wire would be broken, i.e., drawn with a distinctive dashed pattern. Creating a cycle in the diagram would break all the wires in the cycle. Deleting a node would leave previously connected wires dangling, so-called "loose ends," which were also considered to be errors and drawn broken.

One day we invited Rob's thesis advisor, Dr. JC Browne, to see an early prototype of part of the system. We had been discussing how we should think about text strings versus arrays of characters, so we asked his opinion. He suggested that strings could be a fundamental data type, not an array of characters. It was immediately obvious that was the right answer. We could have explicit conversion between string and array if we wanted to, but we wouldn't confuse the user by defining strings as arrays. That discussion also reinforced our commitment to strong typing.

Each developer on the team specialized in one aspect of the project but typically collaborated on all of it. Our excitement level grew as the project began to take shape. We felt like we were on a mission and we pushed ourselves to the limit. We worked long hours, sometimes all night, to reach each milestone.

By late October 1985 enough core capabilities were operational that we decided to set a goal of being able to demonstrate to the rest of the company at least rudimentary functionality in all areas, from editing to execution to instrument I/O. Up until this time, the rest of the company had only the vaguest notion of what our skunkworks group was doing. On the afternoon of October 31st, we had our first demonstration of the fledgling product. It was a big success, even though we had to reboot our machines multiple times during the demos, and no one machine had all the pieces integrated. We began a tradition that day, and for many years afterwards we demonstrated new features to the rest of the company on Halloween.

As we became more proficient using the graphics toolbox on the Mac, and got more of the editing subsystem working, it became clear that it was not going to be possible to build a comparable program on the IBM PC until it also had a sufficiently rich windowing and graphics environment, and it wasn't clear if or when that would happen. Steve Jobs had been fired from Apple, making the future of the Mac uncertain. After a long discussion, Jim and I decided we should continue with the project and release a product on the Mac and see what happened. This was not enthusiastically received by our sales department, who felt it would be impossible to sell a Mac-based product in our market, but Jim convinced them it would be worth a try.

In the early stages of development, Jim learned that software had recently been deemed to be patentable by the US courts. He felt we ought to protect our investment by applying for a patent [Kodosky et al. 1994]. To improve the chances of getting it approved, Jim sketched some circuit diagrams that showed how hardware could implement the behavior of our structured data-flow diagrams. I thought it was a nice touch but perhaps superfluous. However, a little over a decade later those circuit diagrams would be the inspiration for another development effort: LabVIEW for FPGA (field programmable gate array).

In early 1986, we learned the April issue of Electronic Design would dedicate its front cover to announcing our product. It was still unnamed, so we needed to come up with a name by the magazine's deadline. Ron Wolfe, the marketing leader for our project, called a crash meeting where we listed as many possibilities as we could think of, and eventually came up with LabVIEW. It is an acronym for Laboratory Virtual Instrumentation Engineering Workbench.

We thought we were close to finishing, but we were way too optimistic. The more testing that we did, the more bugs that we found. We started beta testing way too early. Getting more people to help test really swamped us with support and more debugging. There was a lot of wasted effort since everyone was reporting the same issues. We ended up extending the beta test period several times throughout the summer of 1986.

We faced other issues during development as well. As our code continued to grow in size, we hit limitations in the tools we were using, specifically, the MegaMax C compiler. Our code overflowed symbol tables and linker buffers. We visited the company several times so that they could make special versions tuned for compiling our code. Just as our program was getting large enough to hit the memory limits of the Mac itself, the Mac Plus was introduced sporting a full megabyte of memory. We were saved.

We had other timely fortuities too. We were avid readers of MacTech and MacTutor and almost every issue had example code showing how to implement a cool new feature, such as floating windows and popup menus. Popup menus violated the Mac UI guidelines, but they were just what we needed to keep our pulldown menus from becoming too unwieldy.

Our performance target was to be about as fast as interpreted Basic, and with a couple of iterations on the implementation of the execution subsystem, we approached that on most computations and I/O. However, on some operations, particularly array operations, LabVIEW was much slower. One benchmark, a "Sieve of Eratosthenes" example, took a few seconds in Basic but did not complete after executing 8 hours in LabVIEW. Clearly, we had to do something about this.

The execution subsystem implemented a straightforward, but literal, interpretation of the data-flow diagram: allocating, copying, and deallocating a data token for each wire. Consider the Array Element Replace node: it had inputs for array, element, and index; it produced an output array the same as the input array but with the value at index replaced by the input element. The node allocated an output array, copied the input array over and replaced the value at index. Then it de-allocated the input array. If this node were inside a loop passing the updated array to the next iteration via a shift register, the array would again get copied, resulting in two copies every iteration.

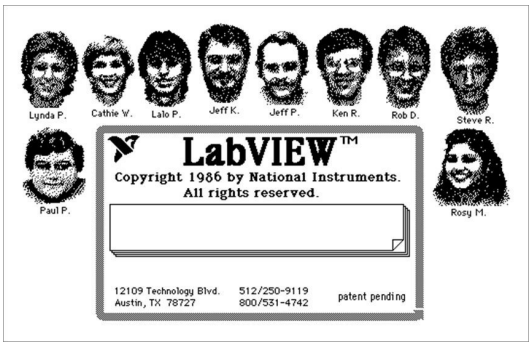

Fig. 3. About LabVIEW Screen
Bitmap images of the LabVIEW 1 developers included in the About LabVIEW screen.

We did what we could to minimize this overhead, but the sieve benchmark was still far off the mark. It would not be until we invented the "in-place" algorithm in version 1.2 (see section 4.1) that we would finally conquer this benchmark.

We finally shipped LabVIEW 1.0 at the end of October 1986, a full year after the first demo version. We were so excited about releasing it, we actually made the first batch of disks ourselves. We stayed late to box up the disks with the documentation and ship the boxes since the shipping department had gone home already.

Figure 3 shows the "About LabVIEW" screen image containing bitmap pictures of the original LabVIEW developers. We thought that would be an appropriately graphical way to sign our work.

2.5 Customer Reaction

LabVIEW 1 had a painful lack of editing features, only minimal I/O and computational capability, and it was slow. But it really was accessible to engineers and scientists who weren't skilled programmers. It inspired them to do projects they otherwise would not have attempted. They saw LabVIEW as a tool they could use to get a project done by themselves, without having to get a team of programmers to help them.

The problem with this situation was that the people we were targeting, those writing Basic programs to control a few bench-top instruments, ignored LabVIEW. They were content to continue their current practice. In their place we had a wide variety of people attempting a broad range of applications from specialized measurements, to laboratory automation and control, to simulation, to clinical research, and on and on. And they all needed much more performance and more memory.

Customers enjoyed interacting with the GUI. The "direct manipulation" of objects on the screen was like a game, and there was a rush of immediate gratification when a button click executed the diagram. On the other hand, there was no undo capability in LabVIEW—an accidental delete meant completely reconstructing what was deleted. Nor was there any ability to "stretch" wires. Once a node was wired, users could not move it. To make space on the diagram to insert a forgotten node, users had to unwire all downstream nodes, move them, and then rewire them. This made editing diagrams tedious and error prone.

In spite of the limitations, customers were persistent and largely successful using LabVIEW. Our early users were Macintosh enthusiasts, so they were tolerant of our efforts on the Mac and they were inclined to persevere in order to highlight the value of the Mac.

One customer reported that it took 20 minutes to load his application, but it wasn't a problem because he would start loading it when he got to work and then go get a cup of coffee and visit

with colleagues. When he returned to his desk, it was ready to go. He also reported that each edit he made took several minutes to respond with the "immediate feedback" updating the diagram, but again, he was able to work around that by multiplexing with other activity. We surmised his application was at the edge of feasibility, but we were still embarrassed and dismayed by the slow performance. Sometime later we were able to see the VIs he created. We were further dismayed by how convoluted his diagrams were. Yet he was still happy because, he said, he was able to conduct his research and publish his results in half the time it took previously. His experience motivated and inspired us—with more effort we could have an even larger impact on researchers' productivity.

Early adopters of LabVIEW included researchers and engineers at Lawrence Livermore Labs, Hughes Aircraft, Dupont, NASA, Lockheed, National Magnet Lab, and others. The flexibility of LabVIEW, the integration of the measurement I/O, and the autonomy the researchers had in evolving their experiments, were the oft-cited reasons for their adoption of LabVIEW.

Of course, there were other customers who were not successful using LabVIEW. The typical reasons were insufficient performance and insufficient memory. While we were able to make incremental improvements in a couple minor releases after version 1, it wouldn't be until a larger machine was introduced and we were able to create a compiler that those issues were resolved (see section 4).

There were two language related topics that made an impression on us, one frustrating and the other challenging. The frustrating one was seeing the extra difficulty people with prior programming experience had in learning LabVIEW. They seemed to be so oriented to sequential programming they had a hard time grasping how to design a program using data-flow. The most egregious example we saw was a VI containing a single sequence structure with dozens of frames, each corresponding to a statement in a sequential program. It was disheartening, particularly when accompanied by a comment that LabVIEW was hard to use.

The challenging topic concerned the inherent parallelism of data-flow. It was easy to create a diagram with multiple loops executing concurrently, but we didn't have a feasible mechanism for communicating between the loops. None of the prototypical test and measurement examples we explored while designing LabVIEW needed such a capability, so we didn't think it was necessary for version 1. That is, until we encountered a customer application where the most natural solution was multiple concurrent loops exchanging data asynchronously. We lost that customer, but we did come up with a clever way to solve the problem (see section 3.6). However, an elegant solution eluded us for 20 years until 2016 (see section 5.4).

3 INFORMAL OPERATIONAL SEMANTICS OF G IN LABVIEW 1

There have been multiple attempts to come up with a formal semantics of the language, but nothing seems to be more understandable and useful than the following somewhat casual description. This is what we understood in 1985 and has remained the same ever since (with minor extra flexibility introduced later). Simplified versions were described and presented in many places (e.g., the LabVIEW user manual and [Kodosky et al. 1991]) but a comprehensive detailed document was never produced, not even internally.

For convenience, all the examples in this section are drawn using a modern version of LabVIEW (2018), but they are converted to black and white to approximate the appearance in version 1. They contain only nodes that were available in LabVIEW 1 although the original icons were not as polished as they are now. The one exception is the comparison node. In LabVIEW 1 it had three boolean outputs: >, =, <. In later versions of LabVIEW the comparison was split into three separate nodes, each with a single boolean output, and that is the form comparisons take in these examples.

The core features are described in detail in this section, while section 5 summarizes the additional language features included in subsequent versions of LabVIEW.

3.1 Basic Data Flow

A diagram contains a set of source terminals, a set of destination terminals, a set of nodes, and a set of signals (aka wires) which connect terminals to form an acyclic graph.

A node has a set of input terminals and a set of output terminals. The terminals can be located on the top, left, bottom and right edges. By convention, node inputs are usually on the left edge and outputs on the right edge so data flows from left to right.

A diagram executes by placing a data token on each of its connected source terminals and then allowing the tokens to propagate along the signal to the destination terminal(s) that are either node input terminals or diagram destination terminals. A signal has exactly one source terminal but it may have more than one destination terminal—fan-out is allowed, but not fan-in. If there are multiple destination terminals the data token is replicated as necessary to create one for each destination.

Whenever a node has received a data token on each connected input terminal it can begin to execute (input terminals that are not connected have a default data token created for them). When a node finishes executing it creates a data token for each connected output terminal. Those tokens propagate along the signals as described earlier. Nodes that have no inputs may begin executing as soon as the diagram begins.

A diagram finishes executing when all its nodes have finished and all data tokens have propagated to all connected destination terminals of the diagram. Figure 4 illustrates multiple stages of the execution of a simple diagram.

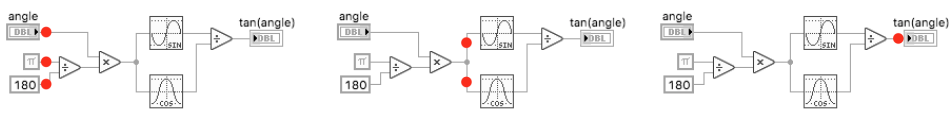

Fig. 4. Data Flow Example
Left: At the start of diagram execution, tokens are placed on each of the diagram's source terminals. The values of the data tokens come from the constant or the associated front panel control named "angle". Center: The second node has executed, propagated its output, and two other nodes are ready to execute (and could execute concurrently). Right: The last node executed and propagated its output to the destination terminal, the front panel indicator, "tan(angle)", so the diagram has completed.

This much is simple traditional data-flow and is easy to understand. Data flows by-value: a node sees only the values on its inputs, nothing about how those values were calculated. When a diagram completes, every node has executed exactly once, and every signal has taken a single data token from its source terminal and propagated it to its destination terminal(s)—no queuing occurs.

Nodes can be functional, where output values depend only on input values, or they can be non-functional and have side-effects, such as file I/O nodes, a random number node, and timing nodes. Multiple nodes may execute concurrently, so if they have side-effects it may be necessary to create an artificial data dependency between them to ensure proper behavior.

While it is obvious from the description so far, it is still worth pointing out that there are no traditional variables in data-flow. A signal is an anonymous variable that is assigned once and cannot be referenced anywhere except by the nodes it is connected to. As a result, there are no naming or scoping issues to deal with.

3.2 Data Types

Data types are represented by controls and indicators of various styles on the front panel of a virtual instrument (VI), described in section 3.5. LabVIEW 1 supported these data types:

- A Boolean, a true-or-false value, was represented as a toggle switch, push button, slide switch, or LED display.
- A number was represented by a text display, a slide, a knob, a meter, or a strip-chart (showing historical values).
- A string was simply a text box.
- A cluster, a collection of data types similar to a C-language "struct", was a box on the panel containing one or more other data type displays.
- An array was a box on the panel containing an element data type display and showing an index display as part of the box. An array of numbers or points could be displayed as a 2D plot on a graph display.

All these types, including clusters and arrays, are passed by value. The array value contains its size as well as the element data.

For readability, each data type uses a different wire pattern when drawing the signal connections on the diagram. Numbers use a solid line, Booleans a dotted line, strings a squiggly line, and clusters a tiny railroad track pattern. An array uses the same pattern as its element data type but increases the thickness of the wire proportional to the dimensionality of the array.

LabVIEW 1 made no distinction between integer and floating-point numbers. All numbers were stored as IEEE-754 extended-precision floating-point which made it possible to store an integer as a denormalized float. A floating-point number was rounded when an integer was required, as for the index input of the Array Index node or for the loop count. An integer operation on a denormalized float did an integer operation, while a floating-point operation automatically interpreted and converted a denormalized float into a normal float as part of its operation.

3.3 Primitive Nodes

The G language contains built-in nodes, called primitive nodes, for fundamental operations on the built-in data types. These include Boolean logic, math, and string manipulation. Primitives exist to bundle data elements together in a cluster or unbundle a cluster into its component data elements. There are also array primitives for building and indexing arrays, replacing an element of an array, etc.

There are no nodes for memory allocation or deallocation operations. That happens automatically with the creation and consumption of data tokens during execution.

Beyond the fundamental data type operations, LabVIEW 1 also included nodes with side-effects, such as a random number generator, tick count, and delay nodes, plus nodes for I/O to files, serial ports, GPIB instruments, and interactive dialog boxes.

3.4 Structures

Structure nodes provide the control-flow that makes the language easy to use. From the outside, each structure node behaves like any other node: all connected input terminals must have data tokens before the structure can begin to execute, and when it finishes executing it produces a data token on each connected output terminal.

A structure's sub-diagram executes like any other diagram. The structure is responsible for transferring the structure's input terminal data tokens to the sub-diagram source terminals, and for transferring the sub-diagram destination terminal data tokens to the structure output terminals. A structure's terminals are often referred to as tunnels since they connect on the outside as well as the inside. If an input is not used inside a structure, i.e., the tunnel is not connected on the inside, the input token is simply discarded (though it still must be propagated to the structure for it to execute).

3.4.1 Case Structure. Execution of the case structure is easiest to describe. The selector is a special input terminal on the left edge of the case structure to which a Boolean or a number may be connected (or a string in later versions of LabVIEW). For a Boolean selector, the case structure has two sub-diagrams, or cases, labeled false and true. For a numeric selector, the structure has two or more sub-diagrams labeled with the values that the sub-diagrams should handle. Figure 5 shows an example of a numeric case structure with two cases.

When the case structure executes, it executes exactly one sub-diagram—the value of the data token supplied to the selector chooses which. Once the sub-diagram is chosen, the tokens on the case structure inputs are transferred to the sub-diagram source terminals and the sub-diagram begins to execute. When the sub-diagram finishes, its destination terminal data tokens are moved to the case structure output terminals and the structure finishes.

All sub-diagrams are essentially functions with the same "arity": all structure inputs are available to every sub-diagram (although not every sub-diagram needs to use every input), and each sub-diagram must produce a data token for each output.

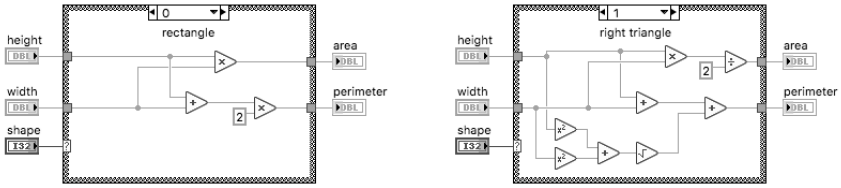

Fig. 5. Case Structure Example
Left: sub-diagram 0 is displayed showing the calculation of the area and perimeter for a rectangle. Right: sub-diagram 1 is displayed showing the calculation for a right triangle. Only one of the sub-diagrams is executed, depending on the value of the selector (special input terminal with [?] glyph).

The case structure has become much more flexible since the original implementation, e.g., introducing a default case and adding ranges, as described in Section 3.4.1.

3.4.2 Sequence Structure. The sequence structure is another multi-sub-diagram structure, with sub-diagrams, or frames, labeled 0 to n. It always executes all of its sub-diagrams in order, beginning with 0 and ending with n, at which time the structure itself completes. Sequence inputs may be used in any sub-diagram, but outputs must be assigned in exactly one sub-diagram.

A sequence structure is often used to control the execution order of nodes that have side-effects but don't have explicit data dependencies. The example in figure 6 shows a typical way of measuring the execution time of some operation.

Use a sequence local to pass a value calculated in one sub-diagram to subsequent sub-diagrams. A sequence local is a small box attached to the inside of the sequence structure. It may be assigned (as a sub-diagram destination terminal) in only one sub-diagram and may be read (as a sub-diagram source terminal) in following sub-diagrams, but not in any prior sub-diagrams (it can't be read before it is assigned).

Even though outputs are computed in earlier sub-diagrams, none are propagated outside the sequence structure until the sequence itself completes, at which time all outputs are propagated.

The sequence structure was intended for ordering operations that had side-effects but no data dependency. And the sequence local was a workable but somewhat awkward means for propagating data from one frame to another. We failed to foresee how badly both constructions would be abused! The sequence structure became a hack for users (typically with conventional programming

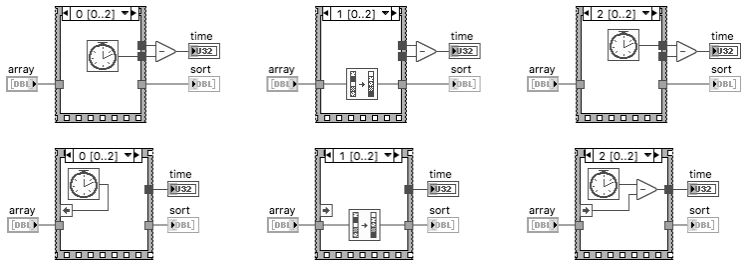

Fig. 6. Sequence Structure Examples Top Left: sub-diagram 0 is displayed showing a call to get the current time. Top Center: sub-diagram 1 is displayed showing the input array being sorted. Top Right: sub-diagram 2 is displayed showing another call to get the current time; after the sequence completes the elapsed time for the sort is computed. Bottom: the same computation is shown rewritten using a sequence local.

experience) who didn't bother to think in terms of data-flow and instead put a "program statement" in each frame to effectively create a sequential language.

3.4.3 `For` *Loop Structure.* A `for` loop structure has a single sub-diagram and a special terminal fixed at the upper left corner called the loop count (a small box with N inside). The `for` loop also has a special terminal called the loop iteration (a small box with i inside) confined to the sub-diagram. The loop iteration is a diagram source terminal of type integer and has the value 0 on the first iteration of the loop, i.e., the first execution of the sub-diagram; it is incremented for each successive iteration. The `for` loop structure completes after the sub-diagram executes N times, so it matches the common construction in C: `for(i=0; i<N; i++){ ... }`.

The outputs of the sub-diagram may be configured in one of two ways: either to accumulate an array of results from every iteration or to output the value from the final iteration.

Similarly, the inputs of the sub-diagram may be configured to index an array or to pass the entire array into each iteration. When an input array is being indexed it is not necessary to wire anything to loop count—it is set implicitly by the size of the array. If multiple input arrays are indexed, the value of loop count is the minimum of the array sizes. Finally, if a value is also directly wired to loop count, it is used along with the array sizes in computing the minimum.

Figure 7 shows examples of "auto-indexing" input and output tunnels. It also shows some equivalent constructions without auto-indexing.

If a two-dimensional array is passed into a `for` loop, the outer dimension will be indexed and successive one-dimensional arrays will be passed into the sub-diagram. Similarly, if a one-dimensional array is passed to an output tunnel, it will be accumulated into a two-dimensional array.

Each iteration, or execution of the sub-diagram, must complete before the next one can begin. This makes it a simple matter to time side-effects as shown in figure 8. The timer runs concurrently with the beep function and delays the start of the next iteration.

Another construction available on a loop structure is called a shift register. A shift register is a collection of tunnels: one on the right edge of the loop and one or more stacked together on the left edge of the loop. The left edge tunnels are initialized at the start of the loop. Thereafter, before the start of each subsequent iteration, the shift register is "shifted": the values on the left side are shifted down to the next lower tunnel and the value on the right side is moved over to the top left tunnel.

Figure 9 shows the simplest shift register. Shifting simply moves the value on the right tunnel to the left tunnel.

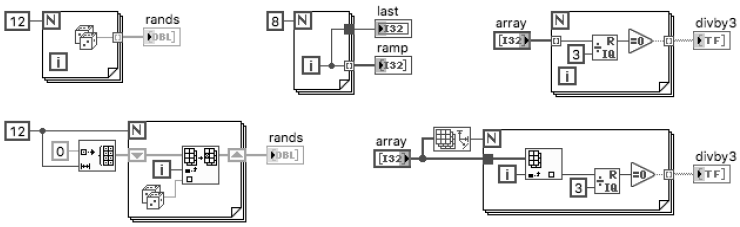

Fig. 7. For Loop Structure Examples
Top Left: The for loop executes its sub-diagram 12 times, and on each iteration a random number is computed. The random numbers are accumulated into an array at the output tunnel. Note the thicker wire outside the for loop indicating an array. Top Center: The for loop executes its sub-diagram 8 times accumulating an output array of the values 0 through 7. The upper output tunnel is configured to just output its last value, 7. Top Right: The for loop executes its sub-diagram once for each element of the input array because the input tunnel is configured to index its array. In this case there is no need to wire a value to loop count. Note the different wire pattern for a boolean value. Bottom Left: Equivalent to the top left example without using output auto-indexing. Bottom Right: Equivalent to the top right example without using input auto-indexing.

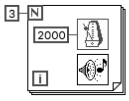

Fig. 8. For Loop Structure Timing Example
The for loop makes 3 beeps two seconds apart (the metronome primitive waits until the next multiple of 2000 ms, i.e., until the next even second on the clock). If the beep itself takes longer than 2 seconds to play, the iteration will not complete until the beep finishes.

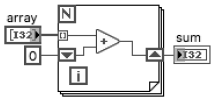

Fig. 9. For Loop Structure Shift Register Example
The for loop sums the elements of an array. The left side is initialized to zero. At the end of the first iteration the right side has the value of the first element of the array. Before the start of the second iteration the right side value is moved to the left side. The second iteration adds the value of the second element of the array so the right side then contains the sum of the first two elements. When the last iteration completes the right side contains the sum of all the array elements. (Note: if the array is empty the sum will be 0).

The left side of a shift register may be expanded to have multiple tunnels. Shifting occurs by moving the values on the left tunnels down one tunnel and then moving the value on the right tunnel to the top left tunnel.

Figure 10 shows a loop that calculates Fibonacci numbers. At the start the left tunnels are initialized to 0 and 1. The result of the add is 1 and is placed in the right tunnel. Before the start of the second iteration the top left value, 0, is moved to the bottom and the right value, 1, is moved to the top left so that the left side now contains 1 and 0. The add again results in the value 1 and is placed in the right tunnel. Before the start of the third iteration the top left value, 1, is moved to the bottom and the right value, 1, is moved to the top left so that the left side now contains 1 and 1. The add results in a 2 for the right tunnel. Before the start of the fourth iteration, the shift causes the left side to contain 2 and 1, and the add produces a 3.

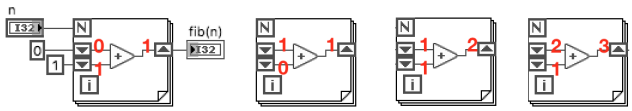

Fig. 10. `For` Loop Structure Shift Register Example
The `for` loop uses a shift register to iteratively calculate the n-th Fibonacci number. The left image shows values from the first iteration and the next 3 images show values on the next 3 iterations. At the start the left tunnels are initialized to 0 and 1. The result of the add is 1 and is placed in the right tunnel. Before the start of the second iteration the top left value, 0, is moved to the bottom and the right value, 1, is moved to the top left so that the left side contains 1 and 0 for the second iteration. The result of the add is again 1 and that is placed in the right tunnel. Before the start of the third iteration the top left value, 1, is moved to the bottom and the right value, 1, is moved to the top so that the left side contains 1 and 1 for the third iteration. The add results in 2 for the right tunnel. Before the start of the fourth iteration, the shift causes the left side to contain 2 and 1, and the add produces a 3. (Note: the behavior of the shift register for $n == 0$ ensures `fib`(0) is 0).

It is possible for a `for` loop to execute zero iterations, e.g., if 0 is wired to loop count or an indexed input array is an empty array. In this case the values for all the output tunnels are the defaults for the data type of the tunnel: empty array, 0 for numeric scalar, false for Boolean, and so on. Shift registers carry their values across—if the loop executes zero iterations, the initial value of the top-left tunnel is copied over to the right tunnel and becomes the output value.

3.4.4 `While` *Loop Structure.* A `while` loop structure has a single sub-diagram. It also has two special terminals confined to the sub-diagram, loop iteration and loop condition. Loop iteration is a diagram source terminal and has the same behavior as in the `for` loop structure. Loop condition is a diagram destination terminal of type Boolean. If `true` is passed to it, then the `while` loop will iterate and execute the sub-diagram again. If `false` is passed to loop condition, then the `while` loop will complete and propagate all its outputs. Figure 11 shows a simple example of a `while` loop.

The same tunnel and shift register options that exist on `for` loops also exist on `while` loops. By default, array input tunnels on a `while` loop are configured to be non-auto-indexing, although they can be changed to auto-indexing. However, the `while` loop is not limited by the array size. If an input array is indexed beyond its size the value used for those iterations is the default for the array element data type, e.g., 0 for a numeric element.

Shift registers work exactly the same way as on a `for` loop.

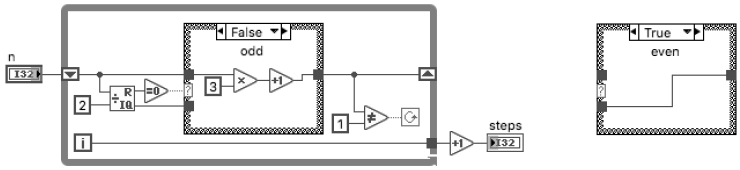

Fig. 11. Figure 11. `While` Loop Structure Example
Compute how many steps are needed to reduce n to 1 using the calculation $n_{i+1} = \{ \frac{n_i}{2}$ if n_i even , $3n_i + 1$ if n_i odd $\}$. It is an open mathematical question whether this loop terminates for all values of $n > 1$.

The `while` loop structure always executes its sub-diagram at least once, so it matches the C construction: `do { ... } while( ... )`.

3.5 Virtual Instruments (VIs)

A software module in LabVIEW/G is called a virtual instrument (VI). It consists of a diagram, a panel, and an icon/connector. The diagram is sometimes called a block diagram to indicate it is the top-level diagram of the VI, as opposed to a (sub-)diagram inside a structure node. The panel is often called the front panel, in analogy to a physical instrument, and as a reminder of the associated interactive capability for unit testing and debugging, for instance.

In traditional programming language terms, the diagram is the implementation of a function while the panel and connector together define the formal parameters in the function prototype, the connector basically giving the order of arguments and the panel defining names and types. The icon serves as a graphical version of the function name. Each VI is usually stored on disk in a separate file, but groups of VIs can be stored in a library file (LLB).

Figure 12 shows the relationships among a VI's panel, diagram, and icon/connector. The inputs and outputs of a VI are the controls and indicators, respectively, which appear on the panel, and which are individually associated with a terminal location on the VI's connector. Furthermore, each control and indicator on the panel has an associated source and destination terminal on the VI's diagram. These associations are not represented graphically, but gestures in the environment make it easy to create, edit and inspect the associations. Controls and indicators usually have unique names and visible labels to make identifying the associated diagram terminal easier.

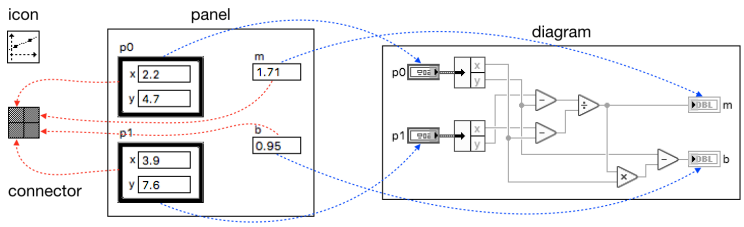

Fig. 12. Virtual Instrument Components
A Virtual Instrument (VI) consists of a panel with controls and indicators, a diagram containing source and destination terminals associated with the controls and indicators, and an icon with an underlying connector pane showing the terminal locations corresponding to the controls and indicators. When an icon is used in another diagram, it represents a call to the VI.

When the icon of a VI appears as a node in a diagram it represents a call to the VI and is sometimes referred to as a subVI node. The VI being called is often referred to as a subVI since it is being called as a subroutine from the calling diagram. The icon and connector are graphically colocated, with the icon on top (although the connector can be shown on top in the editing environment, if desired). Signals on the caller diagram connect to the connector's terminals, thereby passing data into and out of the subVI.

Figure 13 shows one VI calling another as a subVI. It is not necessary to connect all the inputs and outputs of a subVI. At execution time, unconnected inputs are supplied with a default value defined by the subVI. Unconnected outputs simply discard the output data token returned by the subVI. (Later LabVIEW versions introduced the concept of required inputs—it is an error to leave them unwired).

It is not necessary to have every control and indicator associated with a terminal on the connector. Some controls and indicators can exist solely for use when executing a VI interactively, as opposed to executing it as a subVI.

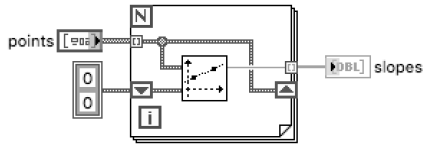

Fig. 13. Calling a SubVI
A VI's Icon/Connector used in a diagram represents a call to the VI, i.e., as a subVI or subroutine of the higher-level diagram.

A subVI may be called from multiple callers, potentially executing concurrently. The subVI's configuration determines what happens. A VI may be configured to be reentrant or serially reusable (non-reentrant) when used as a subVI. In the former case, each caller effectively executes its own temporary copy of the VI, so multiple copies may execute concurrently. In the latter case, callers of the subVI are serialized so that the VI executes on behalf of one caller at a time. This is useful when the VI has side-effects such as file I/O or measurement I/O.

A VI's icon may appear in its own diagram, in which case it represents a recursive call to itself. A recursive VI must be re-entrant, of course. It should be noted that while reentrancy and recursion were part of the original language specification, reentrant subVIs were not implemented until version 2, and recursion was not implemented until much later (2009).

3.6 Executing VIs

Loading a VI file in the LabVIEW environment opens its panel in a new window. All the subVIs are also loaded, although none of their panels is opened. The environment provides multiple ways of navigating to the panels and diagrams of subVIs.

Any VI may be executed interactively from its panel window by clicking the run button (located along with other buttons in the window frame). While it is running, a VI may be terminated early by clicking the abort button. Commonly, a user will use the mouse and keyboard to enter values into the controls on the panel, click the run button, and then observe the updated values on the panel's indicators as the VI executes.

The runtime engine in LabVIEW has always attempted to faithfully exhibit the parallelism in the language. For instance, when multiple loops are executing concurrently, the execution of the nodes on their sub-diagrams will be non-deterministically interleaved. This was originally done via cooperative multitasking and today by using many threads on multicore processors.

An application in LabVIEW typically consists of a hierarchy of VIs, i.e., a top-level VI and a collection of subVIs. The top-level VI is often designed to be controlled interactively as it executes, and as such, it is commonly designed as a continuous loop, relying on the operator to stop it when desired (often by clicking a Boolean button whose terminal is connected to the loop condition terminal). The controls and indicators on the top-level panel are not associated with any terminals on the connector since the top-level VI is not used as a subVI. Furthermore, the source and destination terminals associated with the controls and indicators are located inside the loop. This makes the controls and indicators responsive since they are read and updated on each iteration of the loop.

Figure 14 shows a simple top-level VI modeling a sine wave generator. Each iteration of the loop reads the controls, so they are responsive during execution. When a `false` value is read from the "on" control, the loop will end, completing the execution of the VI. The strip-chart is an example of an indicator whose associated diagram destination terminal accepts a numeric scalar, but the indicator has an internal circular buffer so that it can display the recent history on a graph.

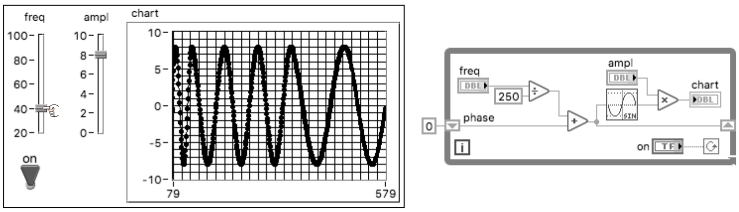

Fig. 14. Top Level VI
A top-level VI is often designed to be interactively operated. It typically contains an infinite loop which reads and updates the panel controls and indicators. Here a sine wave is generated and displayed on a strip-chart Indicator until the user clicks the "on" toggle switch control to its off indication. A slide control adjusts the amplitude, and another adjusts the frequency of the sine wave as it is generated (sliding from high to low frequency in this image).

The fact that every VI has a panel means that regardless of where it is used in a hierarchy every VI can be executed interactively. This facilitates unit testing as well as debugging.

3.7 Uninitialized Shift Registers

The original specification required the left-side of a shift register to be wired on the outside of the loop in order to initialize it. It was a fortuitous accident to discover there was a benefit of leaving it unwired or uninitialized.

When the loop begins to execute the value of the left side of an uninitialized shift register is whatever it was the last time the loop finished executing. An uninitialized shift register behaves like a static variable in C. It is initialized to the default value (zero, false, empty, etc.) when the VI is loaded but thereafter it retains the value last written to it.

Using an uninitialized shift register in a serially reusable subVI allows the subVI to hold onto state data between executions, and so it can be used to transfer data between concurrent loops as they are executing. Figure 15 shows a stateful subVI which models a global variable. One loop writes to it, while another loop reads from it. A slightly more complicated stateful subVI can model a queue or a stack or other data structure.

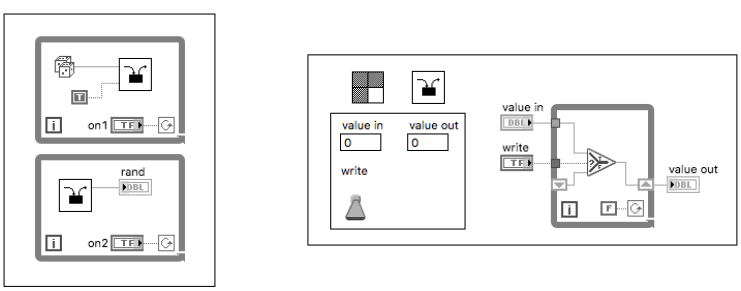

Fig. 15. Stateful SubVI
Left: A VI with two concurrent loops, one writing to a stateful subVI, and the other reading from it. Right: The serially reusable stateful subVI showing its use of an uninitialized shift register to hold on to state information between (serialized) calls from each loop. The subVI loop executes only once per subVI call (because `false` is wired to the loop condition terminal); the only reason the loop is needed is to hold onto the shift register.

A serially reusable stateful subVI is an interesting construction. The subVI is the interface to the state data it contains. The data is protected because the subVI allows access to it by only one caller at a time. In this respect it resembles a monitor in Concurrent Pascal [Hansen 1975].

3.8 Data Logging and Retrieval

After a VI executes, it is possible to take a snapshot of the data on the VI's controls and indicators and store it with the VI. The panel effectively defines a relation, and the data stored is a database recording the executions of the VI. This is not necessarily a useful feature for all VIs but for those that are top-level tests of a device-under-test (DUT), this database provides a simple way to record test results.

Using the File menu, a user can manually take a snapshot, or configure the VI to automatically take a snapshot each time it completes. Also using the File menu, previously recorded data may be viewed on the panel. A special variant of the subVI node can be used on the diagram to programmatically retrieve recorded data.

The original implementation suffered from many shortcomings, not the least of which being an edit to the panel potentially making all the recorded data irretrievable. The data snapshots were stored in the same file as the VI itself which resulted in another set of issues. Some fixes and improvements were made in later versions of LabVIEW but the idea never gained much traction and there was no investment to further develop the idea.

4 HISTORY - PART 2: BEYOND LABVIEW 1

After LabVIEW 1 shipped, for a time, our development efforts continued much the same as during the beta test period: customer support, bug fixes, and some incremental improvements. But we faced a crucial decision.

Our target market was test engineers automating a few bench-top instruments, but most seemed to ignore LabVIEW, either because LabVIEW was too different or because the Mac was considered too much of a toy. On the other hand, scientists, engineers, and researchers in a wide variety of fields were intrigued with LabVIEW, seeing it as a way they could accelerate their work—but they needed LabVIEW to handle larger applications and deliver higher performance.

We had to decide if we should keep our original focus on the test bench or expand the scope of our efforts to embrace the broader market we were seeing. We chose the latter, even though it meant redoubling our development efforts.

LabVIEW was the largest project we had done at NI and represented a huge R&D investment, but now, before there was any return on that investment, we were committing to an even larger investment. We would increase the size of the team and start development on LabVIEW version 2, and it would contain a compiler to provide performance comparable to a C compiler.

After LabVIEW 1 was released, Jack and I attended several conferences of the IEEE Symposium on Visual Languages, but it wasn't until 1991 that we had the opportunity to present our work on LabVIEW [Kodosky et al. 1991].

4.1 LabVIEW 2

The LabVIEW 1 execution system was a literal implementation of data-flow, which was fine for most I/O and computation, but which caused excessive data copying, as discussed earlier.

A compiler would be useless unless we could figure out how to reduce the copying. For example, when a signal forks to multiple destination terminals, instead of copying the data, all destinations could reference the same immutable data at the source terminal. A destination node would have to copy the data only if it needed to read the input and create a modified copy as a new immutable source on its output signal. Placing the copy responsibility on nodes, rather than signals, could

reduce copying. This strategy would be a new implementation of our data-flow, but the semantics of the language would remain unchanged.

In the Array Element Replace (AEltRep) example described in section 2, if it were possible to determine that the input was the sole active reference to the upstream array, then the output could become the active reference to it and alter the array data, rather than making a copy. If the input was not the sole active reference, but the other references were read-only operations and they could be scheduled to complete before the AEltRep node, then at the point the AEltRep node was scheduled, the input would be the sole active reference as before.

We called the generalization of this concept the "in-place" algorithm. Jack and I spent weeks exploring how the idea could work through a hierarchy of structure nodes and subVIs, in the presence of concurrent execution, and what the scheduling implications would be.

We never considered pushing the copy problem back into the user's realm. Users were excited about using LabVIEW because it let them escape the great bane of traditional programming: memory management. We believed we had to do whatever it took to retain that language feature.

When we convinced ourselves we had explored all the cases, Jack and I spent a weekend filling whiteboards with example diagrams, annotated with blue dots and yellow arrows, explaining a progressive local algorithm for determining "inplaceness" and scheduling dependencies.

Monday morning, Steve and Jeff P. began to implement the algorithm, fixing mistakes and refining further as necessary. The first version of the algorithm shipped in LabVIEW 1.2 and gave us the confidence that we could eventually achieve our compiler performance target.

The in-place algorithm was, and still is, one of the most complicated aspects of LabVIEW. One reason is that, rather than a general algorithm with a few special cases, it is more like a large collection of special cases that rhyme in a general way. The algorithm continues to evolve and improve to this day. In 2009, Dr. Ken Kennedy and some of his students at Rice University confirmed for us that our local incremental algorithm compared favorably with a global optimization algorithm. While our algorithm dealt with more complex data types, their analysis provided insights that led to further improvements [Abu-Mahmeed et al. 2009].

During the development of the in-place algorithm and the G compiler, Apple introduced the Mac II, an open machine with NuBus slots, which allowed National Instruments to develop a line of data acquisition (DAQ) cards. The new machines also had large color monitors, higher performance, and much more memory.

Larger memory meant users could make larger programs, which then showed up a host of other limitations in LabVIEW. Basically, the whole architecture of the product broke down with these larger programs. The inability to move an icon once it was wired, an editing nuisance in small programs, became a severe obstacle in the creation of larger programs.

It was clear that LabVIEW 2 not only had to eliminate the performance problem by incorporating a compiler but also had to eliminate the myriad limitations in the editor, loader, linker, and so on, by undergoing a complete rewrite with a new larger-scope architecture.

IBM PCs were still limited to 16-bit applications and still didn't have the necessary graphics capability. So, we forged ahead on the Mac. The development team grew, but our culture remained the same, with the same level of excitement, enthusiasm and passionate debate. But now, our arguments were informed by data: abundant application examples and user feedback.

The hardest part of developing version 2 was stopping. We came back from every trade show and customer visit with a huge list of requirements and suggestions, many of which were so valuable across the board that we had to get them into LabVIEW 2. At the same time, there was mounting pressure from ourselves, the rest of the company, and the customers, to release this new version. We also didn't help matters by demonstrating new features as soon as they appeared to work—there is

an incredible amount of work between the first working case and a fully functional robust feature. Release or keep building? The anxiety level on the team soared.

We set a target completion date in early 1989 and marketing developed a campaign to coincide with the release. We blew past that date, and the campaign had to be scrapped. When the product finally shipped in January 1990, marketing had to create a whole new campaign to reflect the latest LabVIEW appearance and functionality.

In order to generate efficient code, we felt it was necessary to expose a family of numeric types, the usual signed and unsigned integers of various sizes and floating point numbers of various precisions, but whether all this added complexity helped our users was dubious. From a usability perspective, it was clear the built-in primitive functions had to be polymorphic, but creating polymorphic subVIs was beyond our ability at the time (see section 5.3). The polymorphism of primitives extended beyond just numeric type, e.g., Add could add a scalar to an array, a point (pair of numbers) to an array of points, two arrays element by element, and so on.

We were very proud to have preserved the IDE user experience from version 1. There was richer feedback during editing, and the compiler was almost completely hidden. The user could still edit, then click the run button, and experience only about a 1-second delay before the VI started to execute. VIs are compiled individually, and automatically prior to saving, so most of the time there is only a single VI to compile when the run button is clicked.

The performance of the compiler itself was important because we didn't want to introduce a big delay into the edit-run cycle users had relied on for rapid prototyping and debugging of their measurement applications. The performance of the generated code was also important, but we deemed it acceptable if it was within a factor of 2 or so of the comparable C code.

We observed that a VI, with its panel and diagram, was a potent construction, and we reused it in multiple situations. Even when only a portion of its capability was needed, it was still more efficient to reuse the VI construction than to create specialized code. We made the panel of subVIs part of the run-time environment to let users create dialog boxes. We used a special internal VI as the clipboard, able to host items from the panel and from the diagram. We used a panel-only VI as the custom control editor and a diagram-only VI to display the hierarchy of all VIs in memory.

VIs are created and edited within the LabVIEW IDE, but customers need to deploy complete applications on other machines without an IDE, so with LabVIEW 2 we added functionality to build executable files.

4.2 Porting to Windows

Being restricted thus far to the Macintosh platform was very fortunate. It gave us the opportunity to develop and refine LabVIEW, incorporating 6 years of user experience and feedback, while our competitors ignored us.

Shortly after we released LabVIEW 2, Microsoft released Windows 3.1, which was still just a 16-bit OS but it could run on a 32-bit capable 386 processor. Appendix E of their documentation described a mechanism by which a 32-bit application could run under 16-bit Windows. Here was the hook we needed to get LabVIEW to run on a PC. The mechanism involved writing and debugging thousands of lines of assembly code with no tools to help other than hex dumps. It was like programming back in the mid-1970's but we persisted.

LabVIEW was written in C, so, theoretically, it should compile on a different machine, but there were enough gratuitous differences between the Mac and PC compilers that major editing would be required. More significantly, there were major differences in the operating system calls and graphics toolboxes, not just in the APIs but in their fundamental architectures, so major re-architecting of the LabVIEW code would be necessary. Finally, since LabVIEW now contained a compiler, it would

be necessary to write another code generator to target the x86 instruction set in addition to the one targeting 68000.

There were two routes we could have taken to get LabVIEW to the PC. One was to create a copy of the source code and modify it as necessary to port it to the PC, resulting in two different sets of source code that would have to be maintained. The other was to make LabVIEW portable by splitting the code into a platform specific piece and a platform independent piece. This latter approach would be more complicated, but the result would be more valuable and more maintainable. We took the long view and chose the harder path.

Because the tools for developing 32-bit applications on the PC were so poor at the time, we decided that it would actually save time by targeting three platforms as we re-architected LabVIEW: Mac, PC, and SUN. This would also give us greater confidence that the resulting architecture really was portable across multiple platforms.

As we developed this portable version we would test and debug on the Mac and the SUN first where the tools were good, and then we would test on the PC. This flushed out bugs in the platform independent portion of the code, giving us a reasonable chance it would just work on the PC.

Once again, development took much longer than anticipated, but this time we came up with a clever compromise. Rather than wait until the software was completely polished on all platforms, we decided to accelerate the productization of the PC version and release it ahead of the other two platforms—even if it wasn't as fully featured as planned. We reasoned that it would take new users on the PC version a few months to reach the sophistication and expectations of the Mac users, giving us more time to complete all the features. So, in the summer of 1992 we released LabVIEW for the PC to great fanfare, including 57 trade magazine front covers to announce it. We called it version 2.5 because we were reserving the number 3 for the fully complete and fully portable LabVIEW that we would release on all three platforms.

It took another year before we finally completed the fully portable LabVIEW, version 3. We were so thorough in our efforts to make it portable that it was possible to move a VI file from one platform to another without users doing any conversion. In fact, our binary VI files were more portable than simple text files with their maddening end-of-line character differences.

Platform independence caused us to extend the language: we added a path data type in order to have a platform independent way of specifying a location in the file system.

Other major features added in this release included debugging probes for wires, type definitions (a panel-only VI which defined not only data type but also appearance), enumerations, and SI units as an extension of the type system for numerics. There was also a way to create subVIs which were polymorphic with respect to units.

Gary W. Johnson of Lawrence Livermore National Labs was an avid LabVIEW user since version 1. He became the first to write a book on LabVIEW [Johnson and Jennings 2001]. Gary's book not only describes LabVIEW and how to use it effectively, it includes a wealth of information about how to make good electrical measurements, and many in-depth examples. Since that time, many other books have been written about LabVIEW and its use in different applications [Chugani et al. 1998; Paton 1999; Travis 2000].

4.3 The Unique Concerns of a Graphical Language

Creating a graphical language editor is far more complex than a text language editor. Selecting and moving nodes and wires on a 2-D canvas requires much more code than selecting and moving text. The added complexity applies to every aspect of the language. "Adding a keyword" becomes "implementing a graphical UI, with affordances, menus, and dialogs."

Users expected many more features than we had actually implemented. For a long time, it seemed like LabVIEW releases were just catching up to user expectations. New versions improved the

editor, enhanced controls and indicators, improved the compiler and runtime engine, expanded our standard libraries, and of course, fixed bugs.

Our graphical language came with pros and cons. One of its positives is in parallel programming. LabVIEW 5 implemented multithreading and the seamless support of multiprocessor machines. With a dual processor machine, we were now able to deliver the true parallel execution that was expressed in the block diagram. Customers could get an automatic performance improvement by using a multiprocessor machine without making any changes to their programs. A few years later, we would advance the compiler again to compile directly for FPGA, something most languages still cannot do today.

One of the language's negatives is diff: the ability to compare two versions of source and integrate changes. LabVIEW 5 introduced a diff tool to compare two VIs, or two versions of the same VI, and see the differences displayed graphically. Diffing two graphs is considerably harder than diffing text. Merge is more difficult still—something LabVIEW could not do until version 8.0 in 2005, and automerging changelists still remains unimplemented in 2018.

These are just a sample of the issues unique to a graphical language. We try to follow industry practices and use off-the-shelf tools, but regularly we find we have to invent our own solutions.

4.4 Customer Feedback

LabVIEW was an expensive and unique product when it was introduced and many potential customers were even unfamiliar with the Macintosh mouse and graphical user interface. They would need to see demonstrations of LabVIEW and have some hands on experience with it and the Mac before purchasing. Consequently, our sales and marketing people learned to be proficient at demonstrating the Mac and LabVIEW, conducting seminars and teaching about usage and capabilities in training sessions. All of our sales and marketing people were engineers, so they often enjoyed this opportunity. In fact, many field sales people would help customers with their current projects when they made their recurring sales visits. There were also application engineers who provided telephone support as well as example code when necessary. These early customer interactions led to lots of high quality feedback for the LabVIEW developers.

In addition to the sales and marketing interactions, developers also had opportunities to visit customers, conduct demonstrations at various trade shows, and assist presenting training classes. Informal user groups began to form in the early 90s, so National Instruments started to hold an annual conference. This was an efficient and exciting way for LabVIEW developers, as well as the rest of R&D, to meet with lots of customers, and for customers to share their experiences with each other.

Over the years, all these activities have grown and proliferated. Applications Engineering has expanded to include specialists with many years of experience, not only with LabVIEW but with our I/O products as well. Another level of support exists with Systems Engineering. Theses are engineers with domain expertise in various fields from mechanical to electronic, to RF measurements, who can help customers in various industries, and who can create proof-of-concept applications in pre-sales situations. There are now over 1000 Alliance Partners[2] who use LabVIEW and our I/O products to create custom measurement systems for customers in many different industries.

Nowadays, in addition to user groups such as the LAVA user group[3], there are many online discussion forums, most prominently the NI LabVIEW forum[4]. An interactive Idea Exchange[5] helps developers at NI prioritize the many feature requests that are suggested. There are many

[2] https://www.ni.com/en-us/alliance.html
[3] https://lavag.org/
[4] https://forums.ni.com
[5] https://forums.ni.com/t5/LabVIEW-Idea-Exchange/idb-p/labviewideas

LabVIEW Champions[6] who contribute their knowledge and experience answering questions and posting examples in the forums. LabVIEW users can demonstrate proficiency at multiple levels, the highest being Certified LabVIEW Architect (CLA)[7], for which there are multiple conferences per year worldwide. A recently established and noteworthy venue is GDevCon[8]. Most of these internal and external LabVIEW users are not shy about sharing their impressions, expectations, and ideas with our developers.

Perhaps the feedback that had the most significant impact on the evolution of LabVIEW came from users at the Ecole polytechnique fédérale de Lausanne (EPFL), a research institute and university in Lausanne, Switzerland. They ran a lab teaching control theory and applications, an iconic example being the balancing of an inverted pendulum. For these applications they needed to have real-time deterministic execution, and they urged us to make a real-time version of LabVIEW. In fact, they had reverse-engineered some parts of LabVIEW and our I/O drivers in order to achieve deterministic control using LabVIEW 2 on the Mac! We were astounded by their accomplishment but there wasn't any practical way for us to do something similar in the product in the near term. It took another 5+ years for us to develop real-time LabVIEW and the I/O hardware to support it (see section 4.6).

4.5 Major Language Developments Through 2004

At any given time, many changes and additions were in development. When it was time to release a new version, those features that were deemed ready-for-prime-time were enabled. Sometimes we allowed select users to turn on disabled features so that they could experiment with them and give us feedback on further development. As a result, it isn't always exactly clear which version a new feature should be ascribed to, so the description that follows is an approximation.

In the LabVIEW 4 timeframe, I succumbed to customer pressure and added something I have regretted ever since: globals. LabVIEW already had a mechanism for persisting data and allowing access from multiple locations, namely, a serially reusable stateful subVI. It automatically provided protected access to the data and could model a global or a queue or a stack or any other data structure. But it took effort to create such a VI for each use case.

Users clamored for an easier way to create a simple global, so I reused the VI construction, defining a global-type VI containing an unused diagram but whose panel controls were globally accessible from a special type of node. Of course, the node had to implement another mutex mechanism since it directly read or wrote a control's data. So, we paid for my folly with more code and more complexity. It was definitely easier to create a global, but as anyone could have guessed, it was overused and abused and it led to lots of support headaches ever since.

LabVIEW 5, released in 1998, was one of those major milestone releases. On the editor front, it delivered on the long awaited and most-requested feature: undo. We coyly demo'd it in the middle of another VI example we were constructing on stage at our annual user-group meeting and convention (now known as NIWeek, but dubbed back then by Dr. Baroth at JPL as "LabVIEW Woodstock"). The demonstrator "accidentally" deleted a section of the diagram, but nonchalantly hit undo, and continued with his example. The whole room erupted into thunderous applause and cheering.

Since the early 90s, customers expressed interest and excitement in using Apple events, a communication mechanism of the *Open Scripting Architecture (OSA)*[9], to have another application cause the execution of VIs in LabVIEW. Starting with LabVIEW 2, we built a number of APIs to

[6] https://forums.ni.com/t5/LabVIEW-Champions/ct-p/7029
[7] http://sine.ni.com/nips/cds/view/p/lang/en/nid/13477
[8] https://www.gdevcon.com/
[9] https://developer.apple.com/library/archive/documentation/LanguagesUtilities/Conceptual/MacAutomationScriptingGuide/HowMacScriptingWorks.html

find and execute VIs in response to Apple events and other messages, including programmatically from within a VI. By LabVIEW 5, we were able to consolidate these ad hoc mechanisms into a more native one. LabVIEW 5 introduced the VI reference datatype and the ability to call a VI through its reference. A VI reference was strongly typed and it could model a first order or higher order function. Most uses of VI references are just first order functions, but they enable much more dynamic behavior including selecting, loading and executing a VI at runtime.

With the increased use of networking around this time, the need for distributed test and measurement systems arose, and we needed a way to connect distributed LabVIEW instances. The VIServer infrastructure was introduced, which allowed one LabVIEW instance to talk over the network to another and allowed VIs to make remote calls to VIs on other machines.

LabVIEW 5 was a significant advance in multiple dimensions. It garnered 82 trade magazine front covers announcing its arrival.

Until this point, LabVIEW supported purely data-driven programming. To be responsive to a UI event, such as the movement of a slide control on the VI panel, a diagram would have to poll the value of the control in a loop. In order to not consume too much processor time, the polling loop would typically include a `Wait` node. As a result, event-driven elements of test and measurement systems were tedious and difficult to build. This led us to introduce the event structure in LabVIEW 6, which makes it much easier to create applications that responded to a variety of front panel activities as well as other, user-defined events, essentially supporting an event-driven programming paradigm embedded within data-flow. The event structure works well, but I'm still disappointed that we haven't been able to devise a data-flow representation of the source of events and their queuing.

Soon after VI references were added, we began to extend the VIServer by introducing strictly typed references to other objects in LabVIEW such as panels, controls and indicators, diagrams, nodes and structures, terminals, and wires. This enabled a VI to programmatically inspect, and ultimately modify, another VI. This capability wasn't added in response to any specific customer feedback, we just decided it could be a useful capability, though just for internal use. LISP was an early influence for us and because data and programs were both s-expressions, it was easy for a program to create another program. It was intriguing to give LabVIEW a similar capability, which we dubbed scripting. Scripting remained a hidden feature until 2010.

State machines are convenient for describing the behavior of certain parts of an application, e.g., communications protocols and user interactions. Since the beginning, customers had been implementing state machines by using a case structure inside a while loop. Each case contained the code for a corresponding state, including the code to determine the next state. The index of the next state is output from each case and wired to a shift register so it can be used to index the case structure on the next iteration of the loop. While this is a fine implementation strategy, there is no direct representation of the state diagram itself. A user would typically draw the state diagram on paper and then implement it in a VI diagram. Sometime later when a modification was needed, usually after the paper was lost, the user would have to reconstruct the state diagram from the implementation, modify the state diagram, and then modify the implementation.

We introduced an add-on package with LabVIEW 6 to address this. The State Diagram Editor (SDE) toolkit is used for designing and editing state diagrams in a separate window while enforcing consistency with the LabVIEW diagram implementation by using the VI Server scripting capability. The state diagram is saved as part of the VI, and the separate editing window can be reopened to edit it when necessary.

The SDE toolkit never achieved widespread adoption, although there was a small group of customers who valued it. A state diagram is essentially an elaborate control structure with its own graphical representation, which is somewhat orthogonal to the data-flow diagram, so integrating

the two is challenging. While we have had some ideas on how to improve the SDE, we have not invested further effort to do so yet.

By this time in LabVIEW history, there were many customers with teams of developers building large applications consisting of multiple thousands of VIs. These customers needed ways to ensure consistency among their developers. They had their style guidelines and conventions, but they needed automated ways to make sure they were followed. We obliged by introducing the VI Analyzer toolkit in LabVIEW 7, a G-based framework that enables users to create VIs which can inspect other VIs (by internally using the VI Server scripting infrastructure) to verify that proper conventions are followed, e.g., names and colors of controls, etc. Equivalent to StyleCop for C#, the VI Analyzer was pluggable by users with custom analysis even in its first release.

Engineers who had experience with simulation diagrams complained about the excessive wiring when using a shift register as compared to the [1/z] node they were familiar with. So, it was a simple matter to introduce the feedback node as a convenient alternative to a shift register. However, it does precisely what we sought to avoid when we designed the shift register: it creates a cycle in the data-flow graph and requires a special node firing rule: unlike every other node in LabVIEW, a feedback node propagates its output before it receives its input. While the feedback node can be a more compact representation, its unusual behavior currently restricts where it can be used (e.g., it cannot be in a subVI connected to an output because there is no concept of a subVI propagating one of its outputs before executing). The feedback node has the potential to undermine the advantages of LabVIEW's data-flow if it is abused, particularly if we expand the domain where it can be used. So far, that hasn't happened.

4.6 LabVIEW RealTime

Our goal with LabVIEW was to simplify the programming of test and measurement applications which typically involved controlling instruments to generate a stimulus signal and measure a response signal, as described earlier (section 2.2). Very soon after the introduction of LabVIEW, however, we heard from some customers that LabVIEW's graphical data-flow was applicable to control applications as well. In these applications, a measurement is made and compared to a desired value, and then an output is generated to reduce the difference when the next measurement is made. A key characteristic of control applications is the need to execute deterministically and in real-time so that the system remains stable. While we had had experience with some real-time operating systems, we were not very knowledgeable about control systems. One customer at the Ecole polytechnique fédérale de Lausanne (EPFL) in Lausanne, Switzerland, went so far as to demonstrate a real-time control application using LabVIEW 2 running on the Macintosh OS, even though that was not a real-time OS. They implemented a clever interrupt service routine to achieve the determinism, but we weren't able to figure out how we could incorporate something like that into LabVIEW so we weren't able to help.

However, in the LabVIEW 4 timeframe, we initiated a skunkworks project to make a deterministic real-time version of LabVIEW. The goal was to extract the execution framework from LabVIEW and have it run on a small real-time kernel on an embedded processor while optionally communicating with a host processor. A VI's compiled code would be deployed to the embedded processor while its panel would remain in the IDE on the host.

When the host was connected to the embedded processor, it would be possible to operate the panel as well as view and debug the diagram of any downloaded VI. However, the host could be disconnected, allowing the application to run deterministically and unsupervised. It could continue to run even if the host OS crashed.

We built a special data acquisition card with a PC chip set on it and enough memory to host an OS, the LabVIEW Runtime engine, and a reasonably sized application. Not long after LabVIEW 5

shipped, we released LabVIEW RealTime along with NI's first intelligent data acquisition hardware. It successfully delivered real-time deterministic performance with loop times in the millisecond region. In the LabVIEW 7 timeframe a variation of the `while` loop structure was introduced: the `timed loop`. The `timed loop` is a `while` loop tied to a hardware clock, which enables precise scheduling of time-critical loops under LabVIEW RealTime (see section 5.2.2).

4.7 LabVIEW FPGA

I heard about FPGAs from our hardware designers who were using them in our data acquisition products. The idea of a software-defined circuit was intriguing, and I recalled the circuit diagrams Jim included in the original LabVIEW patent application. I set up some joint brainstorming sessions with the hardware designers to explore the idea of compiling a VI diagram to an FPGA. We figured out VHDL constructions that would implement the behavior of structure nodes and basic data-flow. I wrote the first code to generate VHDL from a VI and the others handled the process of getting it compiled and deployed to the FPGA.

By 1997, we had made enough progress that we were able to demonstrate the first barely working prototype at NIWeek. It was clear that a lot more work would be needed to make a product, but the concept was viable. Six years later we released LabVIEW FPGA along with our first user-programmable FPGA-based data acquisition board. Users did not have to know VHDL in order to create a diagram that ran on an FPGA. They could achieve deterministic performance with loop times in the microsecond region.

4.8 The Modern LabVIEW Versions: 2005 to the Present

The LabVIEW team tripled in size leading up to the release of version 8. The new team members had worked together on other projects and brought their prior perspectives with them, instead of adapting to the perspectives and philosophies of the existing LabVIEW team. The culture shifted and curtailed the vision of what was possible with LabVIEW.

A large amount of code was added from other code bases which did not leverage existing code or follow its design strategy. As a result, it severely bloated the LabVIEW code and greatly reduced the stability of the editor. LabVIEW 8 was released in 2005 but it took several 8.x versions over the next few years to return to a semblance of the reliability and stability we had enjoyed with LabVIEW 7.

LabVIEW 8.6 was the last version to use the X.Y numbering scheme. Thereafter, versions were released on an annual basis and labeled with the year: what would have been version 9.0 conveniently became LabVIEW 2009.

LabVIEW 8 represents the transition to modern LabVIEW. It introduced a much-needed project feature to handle the growing scale of applications. The project window could organize and manage the hundreds or thousands of VIs in a large application, along with the required I/O hardware resources.

New editor features included find and replace, which did the graphical equivalent of the eponymous text editor commands; and Quick Drop, a fast way to select nodes using the keyboard instead of navigating through palettes with the mouse. The merge ability mentioned earlier, was introduced at this time.

On the language front, new structures were added: disable and conditional disable structures can "comment out" code and conditionally compile code based on the deployment target; the in-place element structure not only provides a powerful hint to the compiler, but it also helps understandability in much the same way that the C construction: `A[2*i+1] += <expr>;` is much easier to understand than `A[2*i+1] = <expr> + A[2*i+1];`.

On the compiler front, code was optimized by constant-folding, and memory usage was reduced by lazily allocating re-entrant VI data-spaces as needed rather than statically preallocating them.

Also on the language front, new datatypes were added: 64-bit integer, matrix, fixed-point. The most significant addition here, however, was LabVIEW Classes. In keeping with G's data-flow, all objects are passed by-value. Classes are discussed in more detail in section 5.1.7.

Recursion was always part of the language definition, but it hadn't been implemented in the run-time engine, so if a user tried to place a VI's icon inside its own diagram LabVIEW reported it as an error. Recursion was finally implemented as a feature in LabVIEW 2009. That release also contained the parallel `for` (parfor) loop, an extension to the `for` loop which allowed iterations to run concurrently under certain conditions.

In 2010, scripting was finally exposed as a supported feature. Users could now build sophisticated tools and frameworks to automate more of their development and deployment work. Why did we wait so long to expose scripting? The short answer is that we were worried it might become a support nightmare. Scripting is complicated because one has to learn about LabVIEW's internal object architecture, and then the programs one makes with scripting are far removed from the typical test and measurement programs. In retrospect, our concerns were unfounded. There were no excessive support issues with scripting. Those customers who needed it had the skills to use it, and conversely, those who didn't also didn't find scripting to be an appealing feature.

The original compiler in LabVIEW 2 generated machine code for the 68K processor. When LabVIEW was ported to Windows and Unix, the code generator incorporated a layer of abstraction which allowed it to generate code for x86 and Sparc as well (other processors were added later but none resulted in commercially successful releases of LabVIEW). Over multiple versions of LabVIEW, we developed a new compiler with a more formal intermediate representation that could support optimizing transformations. This new compiler used LLVM for code generation. Both compilers were delivered with LabVIEW for several releases. First, the legacy compiler was the default, then the new one was default, and finally the new one was the only one shipped.

We maintained an annual release schedule for LabVIEW so that we would have new features and capabilities to announce at our annual NIWeek conference. Each release included many incremental but useful improvements to the environment, as well as some language additions. However, one of them, called data value references (DVRs), is about as contrary to by-value data-flow as one can imagine: a buffer is allocated and passed around by reference. The user has to keep track of the allocations and deallocations in order to avoid leaking memory, although LabVIEW is at least able to prevent access to a deallocated buffer. Using DVRs can achieve higher runtime performance in some situations, but they may end up being abused as much as globals and lead to as many support issues.

The LabVIEW environment has a Tools menu which includes items that invoke various operations such as re-compiling a directory of VIs, configuring source code control, and launching the library manager, among other things. The Tools menu is populated at LabVIEW launch time from a directory of VIs which are the programs that implement these operations. For a long time it was possible for customers to create a custom application and add it to the Tools menu. When scripting was exposed, it greatly expanded the capabilities of the operations customers could add to the LabVIEW environment. However, it was in 2015 when we made a step-function advancement in the ability to extend the LabVIEW environment.

In the LabVIEW editor, the attributes of various items (controls, nodes, wires, etc.) can be interactively configured through right-click popup menus. These popup menus also contain various shortcut operations, e.g., popup on a node's input terminal to create and wire up a constant. In LabVIEW 2015, we introduced popup menu plugins. A popup menu plugin is simply a VI which adds an item to a context-sensitive popup menu and uses scripting to implement the action. The LabVIEW editor has a built-in wire popup menu item to insert a node. A node popup plugin was created to do the inverse. The plugin was faster to develop than making changes to the LabVIEW

IDE source code and could be accomplished by a knowledgeable user as opposed to an internal developer. We are not able to address all the suggestions customers make for improving the editing experience, but now it is possible for scripting-savvy users to do this themselves in G. The plugin mechanism performs very well (it isn't even necessary to relaunch LabVIEW when modifying a plugin) so most of the time there is no point to moving the operation internal to the IDE.

It is possible to create diagrams that have concurrent loops which communicate, but since the beginning, I have been dissatisfied with LabVIEW's available techniques for doing this. The communication could be accomplished using a shared stateful subVI, or a shared queue reference with one loop enqueuing and the other dequeuing, or even with global variables, but in each case, there is no obvious graphical depiction of the communication. This gnawed at me for 20 years before I began to experiment with an idea, and it took another 10 years to actually implement it, but in LabVIEW 2016 we introduced channels (or channel wires as some prefer to call it). The concept seems obvious in retrospect, so I am still puzzled why it took so long.

Channels use a uniquely styled line, distinct from the usual wires in a diagram, to show the connection from the writer inside one loop directly to the reader inside the other loop. Internally, they employ a shared reference to a stateful subVI which can implement a stream (single-writer single-reader queue), a tag (multiple-writer multiple-reader datum), or a messenger (multiple-writer multiple-reader queue). I consider channels to be the most significant addition to the representation of data-flow in G since LabVIEW 1.

As of this writing, channels are still in the early phase of adoption, but they hold the promise of greatly improving the understandability of applications composed of communicating concurrent loops. I expected, perhaps naively, that the rate of adoption would be faster, but a couple of factors may be slowing it down. New users probably didn't notice channels because it is only now that we are beginning to teach about them in the LabVIEW training classes. Existing users know, and are comfortable using, the previous techniques so the increased data-flow visibility afforded by channels is not compelling enough to switch over. To be fair, there is a level of flexibility that can be achieved using queue references which is not yet available for channels, and some users need that. On the other hand, that flexibility may also further obscure the actual communications taking place (e.g., when a queue reference is stored in some data structure and used later).

When we were preparing the documentation on channels, we did an experiment to evaluate it. We conducted two classes, one with university students new to LabVIEW and another with students who had a couple years experience. The new students picked up the concept easily and completed the example problems with minimal assistance. The experienced students struggled a little more. It seems they had internalized the LabVIEW data-flow model too well: if two nodes are connected by a line, it means there is a scheduling dependency and the second node can begin only when the first completes. Even though channels are distinctively drawn (a pipe-like appearance as opposed to a wire), the experienced user's first reaction is probably to interpret a channel connection as causing a scheduling dependency. In fact, it is just the opposite; and furthermore, connecting two nodes with a wire and a channel is typically an error: the former implies a schedule dependency while the latter implies the nodes are concurrent.

A final interesting note regarding channels is that remarkably little change was needed in the internals of LabVIEW, primarily the type propagation algorithm to account for the lack of schedule dependency, and the compiler where a simple transformation is applied at the start converting the channel to a static VI reference. All the rest is done external to the IDE. A channel itself is defined by a stateful VI, so in addition to the standard ones we supply, a user can define their own channel with custom semantics. A popup menu plugin does the actual work of creating a channel instance: popup on a terminal to create a channel endpoint. The menu plugin VI scripts an instance of the desired channel specialized to the datatype of the terminal.

We always considered an advantage for G to be strongly typed. However, that can be tedious when one has to create multiple VIs to do the same operation on different data types. We had several efforts over the years to create generic VIs, where allowable types and type relations between inputs were defined, but these efforts got bogged down and were shelved. We took a different approach in LabVIEW 2017 when we introduced another language enhancement called malleable VIs (file extension .vim instead of .vi). A malleable VI is similar to a regular VI except that when called from a diagram it is actually inlined and adapts to whatever datatypes it is connected to. Malleable VIs use a "duck typing" approach: any input types are acceptable as long as they don't result in a type-propagation error when the vim is inlined. This avoids the complication of defining an algebra for input type relationships, but the challenge remains for how to reflect an internal type-propagation error at the inputs of the vim.

In conjunction with the vim, we introduced a new structure in G, the type specialization structure (TSS), which is particularly useful within a malleable VI. The TSS is a multi-sub-diagram structure similar to a case structure except there is no selector terminal. Instead, during type-propagation of an inlined malleable VI, each TSS sub-diagram is propagated in turn and the first one to pass without error is accepted as the sub-diagram that will be compiled and executed in the place of the whole TSS. An ongoing research topic is exploring how we could make a recursive vim so that it could work with an arbitrarily nested datatype.

Portions of the LabVIEW code base have been refactored and rewritten multiple times since the major rewrite of LabVIEW 2, but much of it has aged and become harder to maintain. The front panel graphics has become noticeably dated, so in the 2010 timeframe, a new project was undertaken. Originally, it was an effort to update the LabVIEW GUI to a more modern appearance, but it morphed into a complete rewrite of LabVIEW to update the internal architecture as well. Dubbed "LabVIEW NXG", it is a new editor for the same G language. In keeping with tradition, it is taking much longer than expected—version 1 of NXG shipped in 2017 and version 2 in 2018, but it will be several more versions before it includes most of the capability of the previous code base.

5 INFORMAL SEMANTICS OF LABVIEW/G FEATURES BEYOND VERSION 1

The semantics of G's data flow and the core programming structures have remained the same, except for some additional options and increased flexibility. More data types and more programming structures have been added to the language as well. The semantics of the new features is described briefly in this section. The most significant addition to the graphical representation of the language is the channel described in section 5.4.

5.1 Data Types

More data types are now available, and as before, they are defined by a control or indicator on a VI's panel. More display options are available as well, though they do not affect the data type. On the diagram, wire patterns use color to help distinguish data types of signals.

5.1.1 Numeric Data Types. The original numeric data type has been replaced by a set of numeric data types: signed and unsigned integers of 8, 16, 32 and 64 bit size; floating point numbers of single, double, and extended precision; complex floating point numbers of single, double, and extended precision; fixed-point numbers; and high precision timestamps for absolute or relative time.

There are typical rules for "widening" numeric types, e.g., adding an integer and a floating point number produces a floating point output, but there are also other places where a conversion is done, e.g., the data type of a case structure output tunnel will compute the widest type from each sub-diagram. Wherever such a conversion takes place a small red "coercion dot" is drawn at the edge of the terminal where the signal connects.

Unsigned integer data types can have an associated list of unique strings turning it into an enumeration. An enumeration acts like an unsigned integer in most places but some operations can use its boundedness or the associated strings to beneficial effect, e.g., an enum wired to a selector will cause the cases of the case structure to be labeled with the enum strings.

Floating point and complex data types can have an associated unit. A double precision numeric control with a display unit of foot-pounds (ft lb) will create a data type of double precision meter-kilogram (dbl m kg) since meter-kilogram is the SI base unit corresponding to foot-pound. The conversion to and from display unit and base unit is handled by the control/indicator, so that the internal data type is always in terms of SI base units. The built-in nodes compute types using the usual rules, e.g., multiply (dbl m/s) and (dbl kg s) will produce an output type (dbl m kg). There is an easy way to create a subVI which is polymorphic with respect to input units, but that is beyond the scope of this paper.

5.1.2 String and Related Data Types. The string data type is the same but there are more display options available. A related data type has been added, called path. A path is a platform-independent restricted syntax string for specifying the location of a file in a file system. Another related data type has been added, called picture. A picture is a restricted syntax binary string for specifying drawing commands to create an image in a picture indicator on the panel.

5.1.3 Typedef Data Types. A typedef is a wrapper around a type and it functions like a typedef in C. It is implemented as a special type of VI (stored in a file with .ctl extension instead of the usual .vi extension) which has an empty diagram and a single control on its panel. A typedef can be placed on a VI panel as a control or indicator, just like any other, and it operates the same as the control without the typedef wrapper. A typedef can also be placed on a VI diagram as a constant, and again, it operates the same as the constant without the typedef wrapper. A typedef can be used inside a cluster, an array, and another typedef. If the definition of a typedef changes, all the instances in all VIs are updated.

5.1.4 Variant Data Type. A variant is a generic type which can hold any type and value. A variant can be used inside a cluster, an array, and another variant. A special node is used to convert a variant to a specific type and retrieve its value. If the specific type doesn't match what is in the variant, the node returns an error.

5.1.5 Waveform Data Types. The waveform data type is a cluster of a timestamp starting time (t0), a double precision float period (dt), and a one dimensional array of numbers (Y), where the number can be integer, floating point or complex. It is a built-in type in order to encode additional attributes and support custom computational behavior.

5.1.6 Reference Data Types. Reference data types, often called refnums, are integer-sized, strongly typed, "cookies' that are used to refer to a wide variety of objects that are difficult or impossible to treat "by value". These include: refnums for operating system objects such as files and TCP sockets; refnums for I/O session objects imported into LabVIEW by plugins; Data Value Refnums (DVRs) for data structures that are accessible only from within a critical section (defined by the InPlace Element structure); refnums for shared queues, explicit synchronization objects such as occurrences, notifiers, semaphores and rendezvous, as well as built-in and user-defined events; LabVIEW object refnums that refer to the elements of the LabVIEW environment, VIs, panels and diagrams through a rich introspection and program generation API (see section 5.5 on Scripting); refnums for interoperation with objects defined by other languages such as .NET; and more.

5.1.7 LabVIEW Class Data Type. A LabVIEW class (lvclass) is defined by a private opaque cluster containing the member data, a collection of VIs implementing the methods of the class, and an optional parent class. Figure 16 shows a list of VIs that comprise an lvclass.

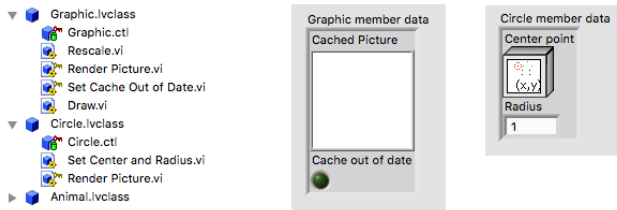

Fig. 16. Class VIs
Left: Expanded list of VIs comprising lvclasses Graphic and Circle. Center, Right: The panels of the .ctl files (similar to a typedef VI described in section 4.1.3) defining the clusters of member data of the Graphic and Circle lvclasses. The remaining VIs are the various methods which operate on the class data.

An object is passed on a wire by-value, and can be thought of as simply a by-value cluster of data (the lvclass member data). However, only the member functions of the class are allowed to access the elements of the cluster. When an object wire forks, there are two independent objects. An object on a wire may be upcast to travel on a parent wire without losing its actual type, so it can be downcast later. The implementation of classes in LabVIEW has two notable differences from other languages.

First, LabVIEW supports automatic version mutation for serialized objects. When an object is flattened to a string (either binary or XML), LabVIEW embeds the class name and version number in the string. When the string is unflattened, the information in the class specifies how to mutate from any old version to the latest version. What is most interesting is that the mutation code can be completely auto-generated. In a text language, the compiler has no knowledge of the before- and after-states of the class—it only knows the fields as they stand at the moment of compilation. But G is edited inside the LabVIEW editor. Every time a user reorders fields, renames a field, deletes/adds/replaces a field, the editor sees that edit. By recording the user's edits and cancelling out any opposite edits (such as adding then deleting a field), the compiler can generate an optimal mapping from any old cluster layout to the new cluster layout. Users are able to write objects to disk or send them across the network and not run into many of the versioning problems that plague other object systems. Not every mutation difficulty is prevented, but many of the hardest ones are.

Second, LabVIEW supports automatic downcasting after method calls. It is perfectly legitimate for a child wire to connect to an input terminal of type parent—that child object will be upcast to its parent, the same as assigning a child variable to a parent variable in other languages. The problem comes at the output: if the subVI is written to take a parent as an input, it almost certainly returns a parent as the output. But most operations on objects do not change their run-time type. The G compiler analyzes subVI diagrams and recognizes when all the modifying operations on a block diagram change only the value of the object, not its type. If the type is preserved all the way from input to output, that means on the caller diagram, if a user passes in a child, the output terminal can be reduced to a child automatically, without any runtime checking to make sure that is legal. This innovation saves users from having to put a downcast node after every method call to the parent class, which otherwise would be an odious burden (both the runtime overhead of type checking and the user annoyance of having to include such nodes).

5.2 Structures

The core structures, `for` and `while` loops, case and sequence structures work as they did originally, but they have additional features for more flexibility. New structures have been added to make more programming patterns feasible.

5.2.1 Case Structure. The case structure still executes a single case based on the selector value but it is much more flexible than originally designed. String is now an allowed type for the selector, as is an enumeration. Cases may be labeled by one or more values or ranges. All case labels must be non-overlapping, and if collectively they do not span the entire range of the selector datatype then there must be a default case. Every case must produce a value for every output tunnel, but now an output tunnel can be configured to supply a default value (0, false, empty, etc.) if it is not wired in a particular case.

5.2.2 Loop Structures. Loop structures iterate as originally designed but additional features improve convenience and performance.

The `while` loop condition terminal can be configured to stop on true or stop on false. A `for` loop can have a condition terminal as well. If a `for` loop stops early, output tunnels accumulating arrays will produce shorter arrays. Loop output tunnels may be configured to concatenate successive array values rather than accumulating into a higher dimensional array. Loop output tunnels may also be configured to have an extra Boolean terminal. If `false` is supplied to this terminal the data propagating to the tunnel in that iteration will be ignored. See Figure 17 for examples.

Fig. 17. For Loop Conditional Terminal and Conditional Tunnel Examples Left: The `for` loop stops early if name matches an element of the roster array. The bottom output tunnel is configured to output the last value, which will be `true` if a match is found. The top output tunnel is configured the same and will be the index of the match if one is found. If roster is an empty array the boolean output is `false` indicating no match was found. Right: the output tunnel will accumulate only those values which are greater than or equal to 18.

The `for` loop can be configured to execute iterations in parallel, in which case it is referred to as a parallel `for` (parfor) loop. An extra terminal is used to specify the number of concurrent workers to use, which by default is equal to the number of processors on the computer.

The timed loop is a `while` loop, decorated with extra terminals on the left and right sides, which provide options for precisely timing when diagram iterations begin, as well as defining priority and processor affinity for code within the loop.

The single cycle timed loop (SCTL) is a specialization of a timed loop designed for high performance on an FPGA. The sub-diagram must execute every clock cycle. Static timing analysis at compile time verifies that the sub-diagram execution time is no more than a clock period. Shift registers on an SCTL map directly to flip-flops enabled by the stop condition of the loop. Combinatorial logic, flip flops, and static timing analysis are the fundamental building blocks of synchronous digital logic design and so FPGA applications built with the single cycle timed loop have very low implementation overhead.

5.2.3 Sequence Structure. Viewing one frame at a time can be tedious, especially if sequence locals are used, so the flat sequence structure was created where each frame appears side-by-side and they execute from left to right. It is easy to switch between stacked and flat appearance, but the

semantics are slightly different. The flat sequence behaves more like a set of single frame sequence structures because outputs from earlier frames get propagated when the frame finishes rather than when the entire structure finishes.

The timed sequence structure is a flat sequence structure, decorated with extra terminals on the left and right sides, so the start time of each frame can be specified.

5.2.4 Event Structure. The event structure is a multi-sub-diagram structure which responds to an event by executing the sub-diagram corresponding to that event. Events include value changes to controls, mouse gestures, key presses, panel window changes, VI state changes, other LabVIEW IDE actions, and timeout waiting for an event. The event structure is usually placed inside a `while` loop so it can be used to process a stream of events in the order in which they occurred. The event structure also has provisions for dynamically configuring to respond to user defined events.

5.2.5 Disable Structure. The diagram disable structure is a multi-sub-diagram structure where exactly one sub-diagram is enabled while the others are disabled. A typical use is to âĂIJcomment outâĂİ code, e.g., a section of diagram which may have broken wires and other errors, so that the remainder of the diagram can be tested before fixing the broken part.

The conditional disable structure is a multi-sub-diagram structure where exactly one sub-diagram is enabled based on a symbol available during compile. A typical use is to conditionally compile code, e.g., a section of diagram which has multiple versions depending on the target to which it is deployed.

5.2.6 In-Place Element Structure. The in-place element structure not only gives a strong memory-allocation hint to the compiler but it can also improve understandability of a diagram by making the intent of the designer clearer. The in-place element structure is used to take a data aggregate apart, operate on a component, and put it back together. This can be done without the in-place element structure, of course, but it may not be as clear what is happening and the compiler may not always recognize the pattern, resulting in an unnecessary data copy.

5.2.7 Type Specialization Structure. The type specialization structure (TSS) is a multi-sub-diagram structure where only one sub-diagram is compiled and executed. The sub-diagram chosen is the first one to complete type-propagation without any errors. The TSS is typically used in a malleable VI (see section 5.3.1), so multiple instances of it may end up in different locations with different data types connected to it.

5.3 Nodes

Many built-in nodes are polymorphic. For instance, Add can add an integer to a float, or a number to an array of numbers, or a number to a cluster of numbers, or two arrays of numbers element by element, and so on.

Comparison functions are also polymorphic. When comparing arrays and clusters they may be configured to compare the aggregates as a whole, or the individual elements.

A global is a special node that is statically linked to a global VI (stored in a .ctl file). It can be configured to write or to read a control on the panel of the global VI, and does so atomically. A local is a special node which refers to a control on the front panel of its VI. It is effectively a global which is scoped to the VI containing it.

A feedback node is a special object that is similar to a free-standing shift register. Technically, it is not a node because it propagates its output before it receives its input. It creates an explicit cycle in the data-flow graph, a cycle which the shift register was designed to eliminate. In certain cases, the use of a feedback node can make a diagram easier to understand. Figure 33 shows an example of an infinite impulse response filter which benefits slightly from using a feedback node.

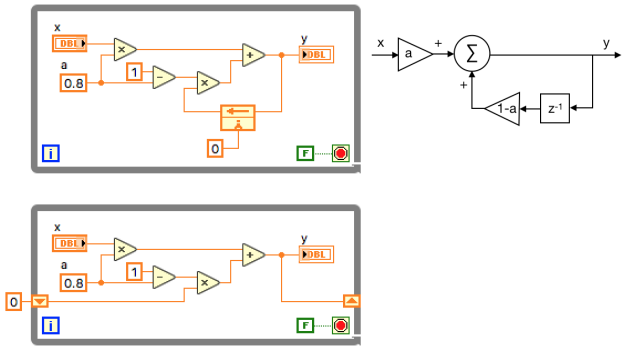

Fig. 18. Infinite Impulse Response Filter Using a Feedback Node
Top: An IIR filter using a feedback node closely resembles the typical textbook representation shown to the right. Bottom: The same filter using a shift register.

A property node is a growable node used to read and write attributes of an object by-reference. An invoke node is used to call a method of an object by-reference. Property and invoke nodes are often used to adjust the appearance of a control on the front panel of a VI.

A formula node contains text-based code which can be a more convenient or recognizable way to represent some calculations.

An occurrence node is a built-in primitive for synchronizing diagrams. It behaves like a semaphore. The output is a reference (refnum) which may be passed to multiple locations to wait for the occurrence to fire, and at least one location which fires it. When the occurrence fires, all the waiters are woken up.

5.3.1 Polymorphic SubVIs. The Split Array node divides a 1D array into two pieces. If you often want to split an array into three pieces, however, it isn't possible to create a VI to do that because unlike built-in nodes, VIs are not polymorphic. You would need a VI to split an array of floats into three pieces and another to split an array of doubles, and another for 16-bit integers, etc. A polyVI is a wrapper VI which can contain several statically typed versions of a VI, where the appropriate one is automatically selected at edit time based on the input data types. A malleable VI (a VI with file extension .vim) is a more recent and simpler way to accomplish even more polymorphism. A malleable VI adapts to whatever types are presented at a call and inlines the resulting instance. The instance is good if no errors occur when propagating the adapted types in the malleable VI diagram.

5.4 Channels

A channel (or channel wire) is a graphical depiction of a data-flow connection from a writer to a reader, but one which does not impose the scheduling dependency of a conventional data-flow wire. A channel has a distinctive representation to easily distinguish it from conventional LabVIEW wires.

There are three primary kinds of channels: stream, tag, and messenger. A stream channel implements a queue with a single writer and a single reader. A tag channel implements a shared variable with multiple writers and multiple readers. A messenger channel implements a queue with multiple writers and multiple readers: elements will be non-deterministically merged from the multiple writers, and non-deterministically distributed to the multiple readers. Other channels

exist as well: lossy stream, accumulator tag, etc., and since all of the channel kinds are implemented using G, it is possible to define more kinds with more elaborate semantics.

Access to a channel is via a write endpoint, which has an element input and a channel output, and a read endpoint, which has a channel input and an element output. Endpoints also have additional inputs and outputs, e.g., for timeout, status, etc.

Figure 19 shows a producer loop connected to a consumer loop using a stream channel. On each iteration of the producer loop, the write endpoint enqueues a value in the stream channel. If the channel is full, the write endpoint blocks, thus stalling the producer loop, until space becomes available. On each iteration of the consumer loop, the read endpoint dequeues a value from the stream channel. If the channel is empty, the read endpoint blocks, thus stalling the consumer loop, until a value becomes available.

The Boolean input to the write endpoint indicates whether the element being enqueued is the last one: if so, the producer loop finishes (the channel will ignore any attempt to write to a stream after the last element was written). The Boolean output from the read endpoint indicates whether the element just dequeued is the last one: if so, the consumer loop finishes (the channel will ignore any attempt to read from a stream after the last element was read).

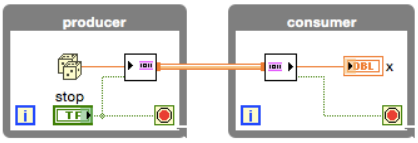

Fig. 19. Stream Channel Example
Left: The `while` loop, labeled producer, calls the write endpoint of the stream channel each iteration to enqueue a new value. When the stop button is pressed the channel is notified that the last element has been written and the producer loop finishes. Right: The `while` loop, labeled consumer, calls the read endpoint of the stream channel to dequeue the next value. When the read endpoint reports that the last element has been dequeued the consumer loop finishes.

If two loops are connected by a channel as well as a conventional wire, it is often an error—the latter wire causes one loop to be scheduled after the other, while the former channel implies they run concurrently.

5.5 Scripting

The VI scripting subsystem provides the ability to programmatically inspect and manipulate items on the front panel and block diagram of a VI. VI scripting is useful for LabVIEW programmers who want to add their own features to the IDE through hooks in existing features like Quick Drop Keyboard Shortcuts (since 2009) and Right-Click Menu Plugins (since 2015).

Scripting is also useful for code inspection, where developers may write their own tests to analyze code for adherence to style guidelines, or to detect coding patterns that may cause unexpected behavior or inefficient memory usage. The VI Analyzer (introduced in version 7) provides a framework for a LabVIEW programmer to run these kinds of tests on his codebase.

5.6 Projects and Libraries

The Project window uses a tree control to organize the VIs and other resources contained within the project as shown in figure 20. The project contains all the programmable targets associated with the application, such as desktop computer, real-time embedded processor, and FPGA. VIs

placed within a target will be syntax-checked for appropriateness on the target. For example, if a VI that contains a File Write node, that VI will be broken when moved to an FPGA target.

Hardware resources associated with the project, such as DAQmx channels, FPGA I/O modules, etc., appear in the project window as well. There are several different ways to "build" a distributable component from VIs in a project, e.g., EXE, DLL, Zip file, etc. All of these build types are managed as build specifications within the project window.

The project window organizes an application as a whole, but groups of VIs can also be organized into a library. For historical reasons, several types of libraries exist. The simplest one has the .llb extension and is basically a Zip file containing the VIs. The library with extension .lvlib provides namespacing and scoping, and it may contain any type of file, not just LabVIEW files. The library with extension .ppl is similar, but it is optimized for packaging code as a distributable component.

The LabVIEW application builder creates an executable file (.exe) from a collection of VIs. However, in order to run the executable file, the LabVIEW Run-Time Engine (RTE) must be installed. The RTE provides support for executing VIs, but none of the editing capability of the LabVIEW IDE.

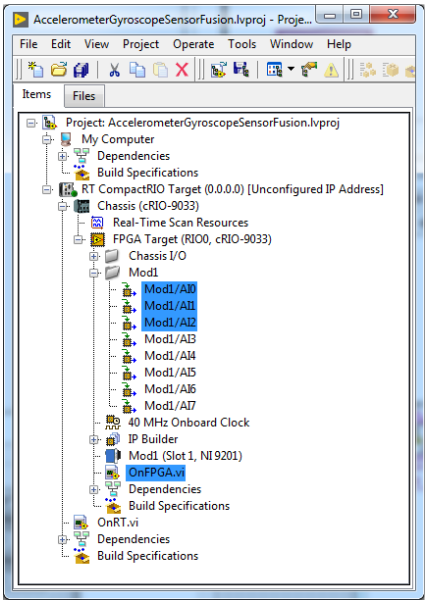

Fig. 20. Project Window
The project window is a tree control showing VIs, libraries, classes, programmable targets, hardware resources, and other items which comprise the project.

5.7 Simulation Diagram

The control & simulation loop can be used to simulate and deploy dynamic system models and control system models. These models may be described by either ordinary differential equations (ODEs) or difference equations. The models may be linear, or they may have nonlinearities. The control & simulation loop is a version of the timed loop. Both loops can iterate deterministically according to a specified period and priority. Wires connecting blocks within the Control & Simulation Loop include arrows that show the direction of the data-flow, and the background of the diagram is a pale yellow to indicate that the diagram semantics is not exactly the same as for G.

The control & simulation loop executes the simulation diagram according to the ODE solver you choose for continuous-time systems or you can choose "Discrete States Only" for discrete-time systems. Like other loops and structures, you can use tunnels to pass data in and out of the control & simulation loop.

5.8 State Diagram

The typical implementation of a state machine has a case structure inside a `while` loop. A shift register holds the current state. The current state is wired to the selector of the case structure. An output tunnel of the case structure supplies the next state to the shift register. Each case contains whatever computation is appropriate for the corresponding state, as well as logic to determine what the next state ought to be.

The state diagram editor (SDE) is a separate window which facilitates the construction of a state machine. Editing states and transitions in the SDE automatically scripts the corresponding changes to the case structure and the next-state indexing. In fact, the case structure and next-state indexing are not directly editable in G. They are locked by the SDE so they always remain consistent with the state diagram representation.

6 RESEARCH AND FUTURE DIRECTIONS

6.1 Structures

There are many hardware devices featuring an embedded processor running a real-time OS, and an FPGA, so it is not uncommon for users to design distributed applications with portions running on the host computer, on the real-time OS, and on the FPGA. The LabVIEW project window has made it possible to do this, but it could be much easier. The target structure is designed to do just that. Target structures can be added to a diagram indicating the deployment of software onto multiple programmable targets as shown in figure 21. The sub-diagram inside a target structure is compiled for a specific target CPU or FPGA, deployed to it, and executed. The infrastructure for doing all this is part of the target structure rather than being exposed in text form in the project tree.

It is clear that the target structure will need to be nestable, e.g., to represent a soft-core processor within an FPGA, but what isn't clear yet is whether and how a target structure will nest within other G structures. Early work is focusing on statically defined targets, but it will be important to figure out how to dynamically assign them at run-time. It will also be important to determine how to disconnect and reconnect to a target in a controlled fashion, as well as how to recover from the failure of a target.

There are other structures which have been described internally in more or less detail, but not yet implemented. A generate structure is similar to a parallel `for` loop except that it is designed for compile-time replication of its sub-diagram, and would find particular utility for FPGA targets.

A race structure is a multi-sub-diagram structure where all sub-diagrams begin executing concurrently. The first diagram to finish causes the others to abort. The outputs of the race structure are the outputs of the diagram that finished first. The non-determinism of the race structure is by design. A timeout capability can easily be added to any activity by placing the activity in one sub-diagram of a race structure and a timer in another.

A closure structure is essentially an anonymous subVI. The output of the closure structure is a reference to this subVI which can be called from one or more locations.

6.2 Time

Time is an important aspect of measurement systems. Control of the temporal behavior of software execution in early versions of LabVIEW relied on various delay nodes. With the introduction of the

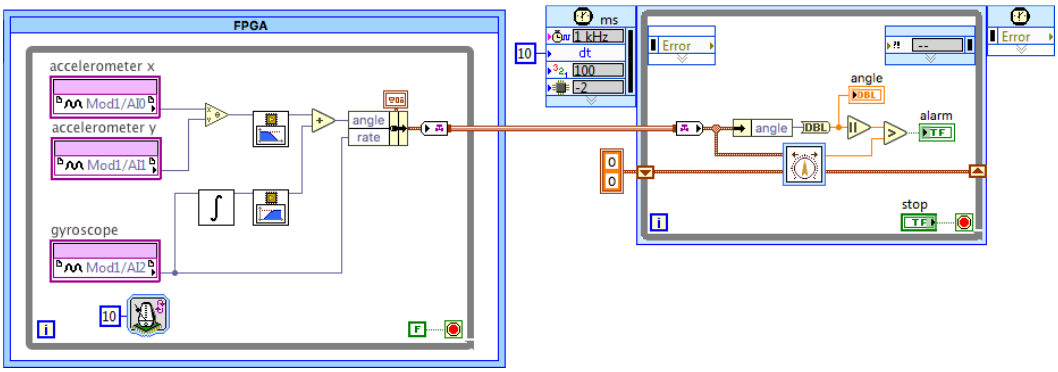

Fig. 21. Target Structure Example
This diagram depicts a sensor-fusion application spanning across a CPU and an FPGA. The diagram within the blue box on the left is compiled and deployed to the FPGA. The timed loop on right runs on the processor. The compiler maps the interconnecting tag channel to a hardware register accessible to both the FPGA and the processor (a stream channel would map to a DMA FIFO).

timed loop and its support for multiple clock sources, execution priorities and processor affinity, much more precise software scheduling became possible, particularly when running on a real-time operating system. But this focus on software scheduling is superfluous. What really matters is the precise timing relationships among the I/O events.

Today, this is accomplished by careful configuration of the timing subsystem of the I/O hardware, and then setting up the software timing to match, as shown in figure 22.

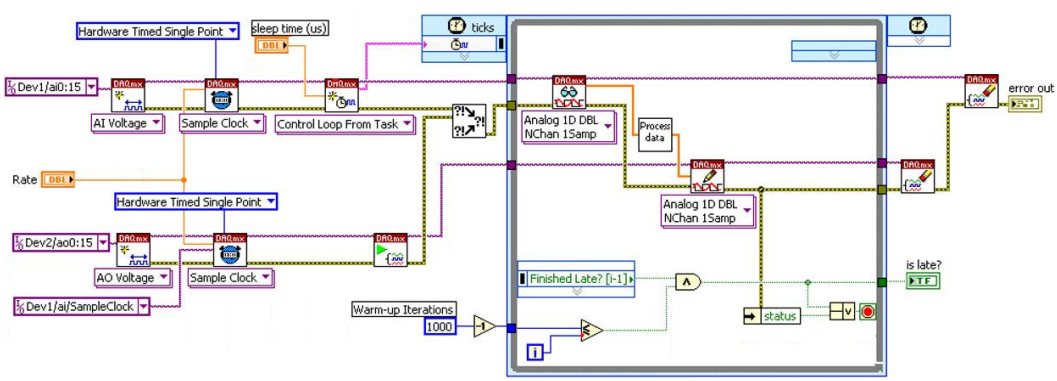

Fig. 22. Timed Loop Data Acquisition
The timing of the data acquisition hardware is done with the DAQmx Config nodes at left and the timed loop is configured to match. When execution of the diagram reaches the DAQmx Start node (green arrow glyph) the I/O hardware is triggered.

An alternative approach would be to directly time the I/O by using time events, as shown in figure 23. In this approach a "future time event" is passed to the Analog Input node. Internally, this node waits until real time catches up to the future time event and then takes the sample. The future time event is updated in the loop—the timing information is co-located with the I/O making the diagram easier to understand.

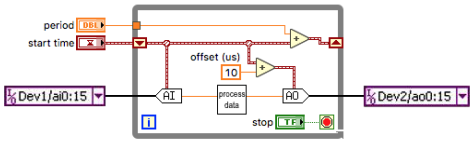

Fig. 23. Directly Timed Data Acquisition
The timing of the data acquisition hardware is done with a future-time event. On the first iteration of the `while` loop, the Analog Input (AI) node waits until `start time` to take the first sample; on the second iteration it waits until `start time + period`. The Analog Output node is scheduled to output its value 10 microseconds after the Analog Input sample is taken.

6.3 Abstraction

The design of a measurement system application often begins with a high-level whiteboard sketch showing the major components and the connections between them. The sketch is an abstraction of the system and can be a good overview for explaining the system to new people. When the application is fully implemented, it may consist of many files with lots of code, but there is no way to bridge to the sketch.

If there were a way to capture the sketch as a formal high-level abstraction of the system, and refine it to a working implementation, capturing the intermediate-level abstractions along the way, that could have a major impact on the design process. A rich development environment would be like a "Google Maps" editor—it would have the ability to semantically zoom out to higher levels of abstraction or zoom in to lower levels and all the way down to the implementation code. Making an edit at one level would generate an obligation to make corresponding edits at lower and higher levels, though some of the obligations could be fulfilled automatically.

Figure 24 shows an example of a measurement system at two levels of abstraction. In addition to showing the software, the diagrams show depictions of the hardware connected to the measurement system. The depictions could simply be pictorial elements or they could be nodes containing simulations, allowing the application to run in a simulated mode.

An editing environment supporting design at multiple levels of abstraction is still a nebulous concept, but LabVIEW's graphical data-flow language seems to make it a feasible prospect.

7 RETROSPECT

7.1 Principles

During the implementation of LabVIEW version 2 (compiler) and version 3 (Windows port) we identified some principles we wanted our process to adhere to.

An early one is "innovate first, then provide backward compatibility", where backward compatibility is the ability to load and run a VI in a later version of LabVIEW. This reflected our belief that major compromises to the internal architecture for the sake of compatibility would constrain and ultimately prevent future innovation. We also felt that if we were sufficiently innovative, then our users would forgive occasional incomplete backward compatibility.

Another principle is to preserve the core integrity of G. We felt that if the data-flow behavior (node firing-rules) were changed, then the foundation of LabVIEW would be destroyed. The core has been preserved, but over the years some new capabilities have been added that in some cases make diagrams harder to understand. There have always been nodes with side-effects, e.g., those that do I/O, but the introduction of globals and references circumvented data-flow, sometimes with deleterious effects. The introduction of the feedback node added the complexity of cycles in the diagram, and a modified firing-rule, though its use appears to be sufficiently confined that

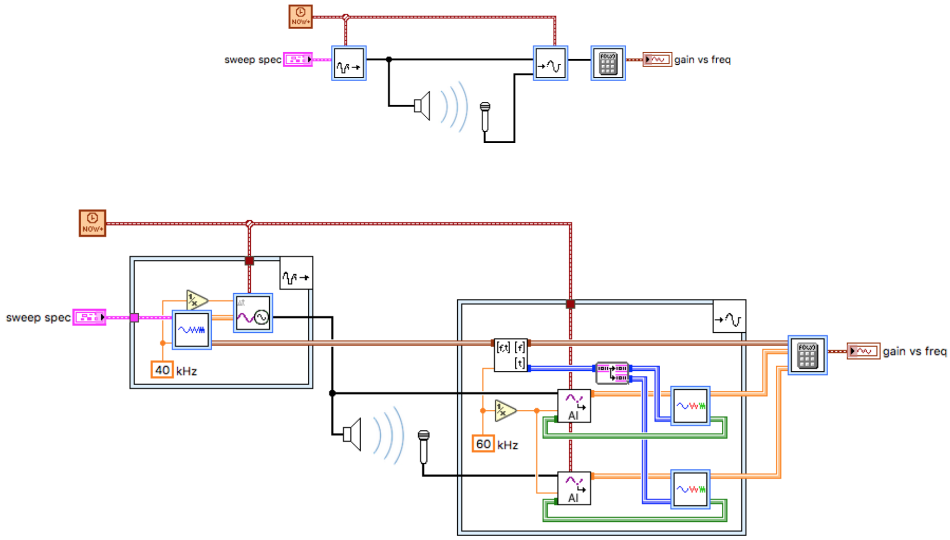

Fig. 24. High Level and Intermediate Level Abstract Diagrams
Top: A diagram showing a measurement application at a high abstraction level. Abstract nodes (with blue borders) for signal generation, acquisition and calculation are shown. Black wires show abstract connections, including connections to hardware avatars. Bottom: The application at an intermediate abstraction level. Two of the high level abstract nodes are expanded to show the next level of refinement. Inside, some of the nodes are concrete implementations (primitives or subVIs) and others are abstract.

understandability doesn't suffer. The introduction of channel wires adds cycles in the diagram, but those cycles replace sections of the code where cycles had implicitly existed through the use of references, so channels actually improve diagram readability.

Another principle is to "make difficult tasks possible before making simple tasks trivial". We felt it was more valuable to a user to extend the scope of what he could do with LabVIEW than to further simplify something he can already do. This principle can expand our market but may also cause some user annoyance.

Over the years, more parts of the LabVIEW IDE were written using G, e.g., the navigation window in version 7.1 and the new icon editor in version 2009. We referred to this as "eating our own dogfood" and believed more was better. If we could benefit from higher productivity using G to write parts of LabVIEW, that would be good. If the resulting performance was not as high as desired, we could rewrite in C++, or we could invest in improving the G compiler. The latter course would benefit us as well as all other users. Furthermore, parts written in G could be released independently of LabVIEW itself, and those parts could be customized or extended by savvy users. Dogfooding continues to be used in new releases of LabVIEW.

7.2 Impressions

LabVIEW is used mostly in test, measurement, and control applications employing measurement instruments, analog-to-digital (A/D), digital-to-analog (D/A), and digital I/O hardware [Czerwinski and Oddershede 2011; Jamal and Wenzel 1995; Morris and Langari 2016; Wang et al. 2012; Wrobel et al. 2016]. LabVIEW's popularity is certainly due in part to its tight integration with the driver software that interfaces to the I/O hardware, and the breadth of hardware products that work with

LabVIEW. The wide range of libraries including vision, motion, control, simulation, analysis, signal processing, and communication also contributes to the popularity.

There are also innate characteristics of LabVIEW that drive its popularity. The one most often mentioned by users is the fast development cycle. No boilerplate is needed to get started—the empty VI executes (doing nothing). It is easy to incrementally add to its panel and diagram, and periodically run it during development to check its operation. This fosters early detection of bugs, resulting in a more efficient development process.

There are many anecdotal reports of productivity gains using LabVIEW compared to languages used previously. Users report gains of up to 10x, e.g., a project that would previously have taken a year took about a month. The only side-by-side comparison I'm aware of is a project at the NASA Jet Propulsion Laboratory (JPL), which had a team of C programmers and a team of LabVIEW programmers implement the same application. It was difficult for them to get an accurate comparison because the C team didn't finish, but their estimate was roughly a factor of 15 times faster for the LabVIEW team [Wells and Baroth 1992].

Many users cite the ease with which they can troubleshoot their systems, including hardware configuration and cabling issues. It is easy to add a strip-chart indicator to a panel, connect it to a signal of interest, and observe it as the application runs to see if it behaves as expected.

One published study compared LabVIEW diagrams to text code and found benefits to the graphical representation in several regards including debugging [Whitley et al. 2006].

An applied mathematician reported using a strip-chart to monitor the convergence of a numerical algorithm he was developing. Based on the behavior he observed he was able to improve his algorithm so that it converged much quicker.

Another frequent user comment is that LabVIEW is accessible and fun to use. Users are inspired to attempt projects they wouldn't have in the past, and then to their surprise (and their supervisor's) they succeed quicker than expected. Some users claim LabVIEW is a useful tool masquerading as a video game.

From my own perspective, I also find LabVIEW enjoyable to use. I am reasonably proficient using a variety of programming languages, but I find thinking in terms of data-flow to be more insightful, especially when designing parallel and distributed applications. And I find programming graphically to be helpful too. If I am unable to make a diagram look neat it is often because I haven't designed the algorithm well.

As mentioned before, I consider my biggest blunder to be the introduction of global variables. A tightly scoped global is much better represented as a tag channel. And a serially reusable stateful subVI is a much better implementation in all other cases because it is more flexible, can sometimes be more efficient (e.g., for global arrays), and the slight extra effort to create it can keep it from being abused.

I find designing algorithms with by-value data-flow to be safer and easier, so I have regretted seeing the proliferation of references in G, particularly the data-value-reference. In contrast, I feel particularly proud of the introduction of channels. A common use case for references, communication among concurrent loops, can now be done with channels, a new additional representation of by-value data-flow.

7.3 Reasons for Success

Why has LabVIEW achieved the success it has as a visual language? This is a question I have been asked occasionally but I don't have a satisfying answer. Part of the reason is that while I have learned about various diagram types (state diagrams, flow charts, signal flow graphs, data flow diagrams, etc.), I never studied or even used another visual language, although I have seen some example screen images.

I thought, perhaps naively or arrogantly, that LabVIEW was a good idea with a great deal of potential, and it would be a better use of my time to expand its scope and capabilities than to investigate other visual languages. I suppose I still suffer from the same prejudice even today.

One visual language that has been very successful and has seen wide adoption is Simulink from Mathworks. It is a domain specific language for simulation and modeling based on signal flow graphs, a notation that has been in use by control and simulation experts since analog computer days. Simulink is very good at what it does, but it is not a general programming language.

Two other visual languages that I am aware of and which have had a measure of success are VEE and Prograph. Both are based on data flow, although VEE has extra execution provisions.

VEE was designed primarily to help users easily create applications using Hewlett-Packard instruments. It appears to do that well and enjoys a loyal following, however, it may not be general enough to be attractive to more users.

Prograph is the most interesting case to me. It was designed by computer science researchers so it has the concepts and rigor of a traditional programming language, but they are expressed graphically. Why has it not enjoyed greater success?

I think there are both cosmetic reasons and pragmatic reasons, and comparison with LabVIEW may offer some insight.

LabVIEW fully embraces graphics: functions have icons and users are encouraged to design their own icons for functions (VIs) they create. Icons allow diagrams to express a lot yet still be compact. It takes time to create an icon, but an elegant and distinctive icon is easy to recognize and distinguish from others.

Prograph is only partly graphical: functions are nondescript text boxes and very little is discernible at a glance without reading the text. Since the text can be lengthy, the boxes are short and wide. In order to more effectively use the vertical dimension, data flows top to bottom, but even still, diagrams cannot be very compact. And there may be a cognitive penalty in having to read the text left to right while following the data flow top to bottom.

LabVIEW uses a stylized box to represent a loop. As a result the body of the loop is visible in the context of the rest of the diagram. In Prograph, the body of a loop is only visible in a separate window. That means more windows must be open to see a program, and the inability to see the body of a loop in the enclosing context adds a cognitive burden.

Perhaps the most significant issue is the difference in perspectives taken by LabVIEW and Prograph. LabVIEW is an engineering tool containing a graphical programming language, G. It is designed to appeal to scientists and engineers who are building measurement and control systems, so it is tightly integrated with drivers that connect to a variety of hardware devices for physical I/O. LabVIEW can be thought of as a CAD tool, a CAD tool which generates revenue to sustain its continued development, including the continued development of the G language.

Prograph is a graphical programming language designed to compete with text-based languages. The target customer is a programmer. But most programmers actually enjoy text-based programming. They are comfortable with the syntactical complexity of a text language and they feel most productive with fingers on the keyboard. Convincing them to switch to a graphical language is almost impossible. Even programmers who work in a test and measurement area, and who could benefit most, ignore LabVIEW in droves. How much harder it must be for Prograph to gain traction.

Designing a graphical language is a small part of the total effort. The largest part is building the environment for editing the language. Editing a diagram is vastly more complex than editing text. Same for diff and merge. Lots of free or low-cost tools exist for managing text, but the corresponding tools for diagrams often have to be developed from scratch, requiring substantial funding.

In the end, maybe the limited success of Prograph and other visual languages is due to insufficient funding, and the difficulty of generating revenue by competing against low cost text languages in doing the same jobs.

8 SUMMARY

LabVIEW is a tool for implementing automated measurement and control systems. Its core graphical language, G, was originally designed to be a "non-programming" way to construct the software for such systems, but as it evolved, it has become recognized as a productive general programming language.

The breadth of applications users have built with LabVIEW is extraordinary: from programs testing medical devices [VI Engineering Inc. 2009; Weisberg 2007] to programs controlling medical devices [Stevens 2008; Wiltberger 2007]; from monitoring large generators [de Carvalho 2018] to controlling a 3-D printer [Schacht 2018]; from semiconductor testing [Fordor 2017; Schwarz 2018] to controlling the beam line at the CERN Large Hadron Collider [Losito 2008; Losito and Masi 2008].

These applications benefit from the productivity of LabVIEW, the tight integration with hardware I/O and timing, and the performance available with LabVIEW FPGA. The success our users achieve, and their enthusiasm for LabVIEW, are powerful motivators for us to continue our research and development.

We anticipate extending the language with features found in other languages (e.g., interfaces and closures) as well as programming constructs not typically found elsewhere (e.g., non-deterministic race structure and target structure). We also anticipate inventing additional representations that obviate the need for using references in some cases (e.g., channels).

Another major area of focus for us will be representing aspects of measurement systems that go beyond the scope of other languages. In particular, we have a keen interest in graphically representing time and time-driven I/O, as well as I/O configuration, in the language. We also believe our graphical representation lends itself well to representing a system at multiple levels of abstraction with the ability to "semantically zoom" between levels.

Graphically depicting programs, especially distributed parallel programs, makes it much easier to understand them. And if the graphical depiction *is* the program, that makes it much easier to construct as well.

ACKNOWLEDGMENTS

Jack MacCrisken was an ideal partner in brainstorming. He had a special knack for distilling our rambling discussions into the nuggets worth keeping. And when we formed the initial vision for LabVIEW, Jack brought his considerable experience and skill to bear in defining a software architecture capable of supporting the vision.

Jim Truchard posed the challenge of creating a product in the first place, after having intuited the market need and the suitability of National Instruments addressing it. Jim was tireless in his encouragement and support as Jack and I struggled early on. Jim vetted our ideas, as well as contributing his own insights and suggestions during our joint brainstorming sessions. During the implementation phase, Jim's enthusiasm and support kept us inspired and motivated. He was the foremost champion for our work, always urging us on but exhibiting utmost patience in the face of the inevitable delays and setbacks.

Steve Rogers and Rob Dye are two original LabVIEW developers who continue to work with me. We have had countless debates and discussions bordering on the philosophical, and their insights are stamped everywhere in LabVIEW.

The rest of the original team: Jeff Parker, Cathie Wier, Lynda Pape, Lalo Perez, Paul Ponville, Ken Rozendal, and Rosy Mehrotra, also contributed heroic efforts and long hours to the development of LabVIEW 1. Their spirited camaraderie in our skunkworks environment created a once-in-a-lifetime experience we all still cherish.

The enthusiastic feedback from the early LabVIEW users was crucial to our success. Their loyalty and tolerance and advocacy made them the best customers we could hope for.

Over the years, many other developers, too many to list, have made LabVIEW what it is today. The vibrant community of users continues to inspire our efforts.

I thank Rob Dye, Jacob Kornerup, Patricia Derler, Greg McKaskle, Duncan Hudson, Darren Nattinger, Newton Petersen, Doug Bendele, Jeannie Falcon and Joe Peck for review and suggestions in preparing this paper. Special thanks to Stephen Loftus-Mercer for his careful reading and detailed comments and suggestions which have greatly improved this paper.

I also thank co-chairs Guy Steele and Dick Gabriel for suggesting I write this paper for HOPL IV, and the anonymous reviewers for their insightful suggestions and comments, and finally, Crista Lopes for her help in the final editing.

REFERENCES

Samah Abu-Mahmeed, Cheryl Mccosh, Zoran Budimlić, Ken Kennedy, Kaushik Ravindran, Kevin Hogan, Paul Austin, Steve Rogers, and Jacob Kornerup. 2009. Scheduling Tasks to Maximize Usage of Aggregate Variables in Place. In *Proceedings of the 18th International Conference on Compiler Construction: Held As Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009* (York, UK) (*CC '09*). Springer-Verlag, Berlin, Heidelberg, 204–219. https://doi.org/10.1007/978-3-642-00722-4_15

Mahesh L. Chugani, Abhay R. Samant, and Michael Cerna. 1998. *LabVIEW Signal Processing*. Prentice-Hall, Upper Saddle River, NJ, USA.

Fabian Czerwinski and Lene B. Oddershede. 2011. TimeSeriesStreaming.vi: LabVIEW program for reliable data streaming of large analog time series. *Computer Physics Communications* 182, 2 (Feb.), 485–489. https://doi.org/10.1016/j.cpc.2010.10.019

Alan L. Davis and Robert M. Keller. 1982. Data flow program graphs. *Computer* 15, 2 (Feb.), 26–41.

André Tomaz de Carvalho. 2018. Modular Instrumentation for Online Partial Discharge Monitoring. https://web.archive.org/web/20181109120855/http://sine.ni.com/cs/app/doc/p/id/cs-17673.

> Cepel developed a modular instrumentation system for online monitoring of partial discharge in hydrogenerators using PXI modular instrumentation from NI and signal processing algorithms implemented with LabVIEW. The company has adopted the system as an effective predictive maintenance tool.

Accessed: 02-07-2020.

Ferenc Fordor. 2017. Probing of Large-Array, Fine-Pitch Microbumps for 3D ICs. https://web.archive.org/web/20181002045341/http://sine.ni.com/cs/app/doc/p/id/cs-17674. April 2017.

> Imec needed to perform die tests prior to stacking 3D ICs to achieve sufficient compound stack yield by probing the interconnect microbumps for pre-bond test access. The company built a unique, fully automatic system to characterize prototype probe cards for large-array, fine-pitch microbumps on its advanced test wafers using NI PXI instruments, the NI Semiconductor Test System (STS), and LabVIEW.

Accessed: 02-07-2020.

D. D. Gajski, D. A. Padua, and D. J. Kuck. 1982. A Second Opinion on Data Flow Machines and Languages. *Computer* 15, 2 (Feb), 58–69. https://doi.org/10.1109/MC.1982.1653942

Per Brinch Hansen. 1975. The Programming Language Concurrent Pascal. *IEEE Trans. Softw. Eng.* 1, 1 (March), 199–207. https://doi.org/10.1109/TSE.1975.6312840

IEEE. 1988. IEEE Standard Digital Interface for Programmable Instrumentation. *ANSI/IEEE Std 488.1-1987* 1, 488 (June), 1–96. https://doi.org/10.1109/IEEESTD.1988.81527

R. Jamal and L. Wenzel. 1995. The applicability of the visual programming language LabVIEW to large real-world applications. In *Proceedings of Symposium on Visual Languages*. IEEE, Darmstadt, Germany, Germany (Sep.), 99–106. https://doi.org/10.1109/VL.1995.520791

Gary W. Johnson and Richard Jennings. 2001. *LabVIEW Graphical Programming* (3rd ed.). McGraw-Hill Professional, New York, NY, USA.

J. Kodosky, J. MacCrisken, and G. Rymar. 1991. Visual programming using structured data flow. In *Proceedings 1991 IEEE Workshop on Visual Languages*. IEEE, Kobe, Japan, Japan (Oct), 34–39. https://doi.org/10.1109/WVL.1991.238853

Jeffrey L Kodosky, James J. Truchard, and John E. MacCrisken. 1994. Graphical Method for Programming a Virtual Instrument.
A method for programming a computer to execute a procedure, is based on a graphical interface which utilizes data flow diagrams to represent the procedure. The method stores a plurality of executable functions, scheduling functions, and data types. A data flow diagram is assembled in response to the user input utilizing icons which correspond to the respective executable functions, scheduling functions, and data types which are interconnected by arcs on the screen. A panel, representative of an instrument front panel having input and output formats is likewise assembled for the data flow diagram. An executable program is generated in response to the data flow diagram and the panel utilizing the executable functions, scheduling functions, and data types stored in the memory. Furthermore, the executable functions may include user defined functions that have been generated using the method for programming. In this manner, a hierarchy of procedures is implemented, each represented by a data flow diagram.
U.S. Patent 5,301,336, Filed July 12, 1989, Issued April 5, 1994.

Roberto Losito. 2008. CERN Uses NI LabVIEW Software and PXI Hardware to Control World's Largest Particle Accelerator. https://web.archive.org/web/20190719005950/http://sine.ni.com/cs/app/doc/p/id/cs-10795.
CERN needed to measure and control, in real time, the position of bulk components to absorb energetic particles out of the nominal beam core with high reliability and accuracy at the Large Hadron Collider (LHC). They used LabVIEW, LabVIEW Real-Time, LabVIEW FPGA, and NI SoftMotion software with NI R Series I/O hardware for PXI to develop an FPGA-based motion control system for approximately 600 stepper motors with millisecond synchronization over the 27 km of the LHC.
Accessed: 02-07-2020.

Roberto Losito and Alessandro Masi. 2008. Controlling and measuring ion beams in real time - Researchers at CERN overcome unique and stringent test and measurement contraints using an integrated software and data acquisition system. *R and d -Newton-* 50 (04), 54–54.

Alan S. Morris Morris and Reza Langari. 2016. *Data Acquisition with LabVIEW. Chapter 12 of Measurement and Instrumentation: Theory and Application* (2 ed.). Elsevier, North Andover, MA, USA. 347–374 pages.

Barry E. Paton. 1999. *Sensors, Transducers, &Amp; LabVIEW*. Prentice Hall PTR, Upper Saddle River, NJ, USA.

Stijn Schacht. 2018. The Materialise Control Platform: Using CompactRIO to Revolutionize 3D Printing. https://web.archive.org/web/20181002045341/http://sine.ni.com/cs/app/doc/p/id/cs-17674. May 2018.
Materialise used CompactRIO and LabVIEW to develop a ready-to-start, software-driven, embedded controller platform specifically for laser-based 3D printing applications.
Accessed: 02-07-2020.

Yoram Schwarz. 2018. Building A Next-Generation Scanning Electron Microscope to Streamline Semiconductor Manufacturing. https://web.archive.org/web/20181008000146/http://sine.ni.com/cs/app/doc/p/id/cs-17668.
PDF Solutions needed to build a control system for a scanning electron microscope (SEM) to acquire images with nanometer alignment and autofocus. The company used FlexRIO and LabVIEW to develop a system to perform control of the SEM and built a .NET interoperability assembly to communicate with external applications.
Accessed: 02-07-2020.

Jeff Stevens. 2008. Using Graphical System Design for Tumor Treatment. https://web.archive.org/web/20160531220709/http://sine.ni.com/cs/app/doc/p/id/cs-11023. August 2008.
Sanarus designed, prototyped, and deployed an FDA-approved, Class II medical device used to treat breast tumors in a less invasive and nearly painless procedure. The company used CompactRIO, LabVIEW Real-Time, and LabVIEW FPGA, maintained the design process within strict regulatory guidelines, and met an extremely short time-to-market window.
Accessed: 02-07-2020.

Jeffrey Travis. 2000. *Internet Applications in LabVIEW*. Prentice Hall PTR, Upper Saddle River, NJ, USA.

James J. Truchard and L. Wayne Ashby. 1978. AN/FQM-12(V) sonar test set. I: an automated acoustical measurement system. *The Journal of the Acoustical Society of America* 64, S1, S7–S7. https://doi.org/10.1121/1.2004382

VI Engineering Inc. 2009. Medical Device Testing with National Instrument Software and Modular Instruments. https://web.archive.org/web/20170302154732/http://www.ni.com/white-paper/5847/en/. May 2009.
This paper describes the testing of medical devices such as defibrillators, pacemakers, hearing aids, glucose monitors, and more.
National Instruments LabVIEW and instrumentation tools are well suited for the test of medical device testing for the following reasons:
- Sophisticated analysis required—complex digital circuitry requires custom signal analysis specific to the device under test. Advanced analysis capabilities in LabVIEW provide this capability.
- Complex waveform generation and capture—unique waveforms must be generated to simulate the subject input. A combination of LabVIEW and NI modular instruments provide this waveform generation and capture.

- Tight timing and synchronization requirements—the generated input and measured output may need to be synchronized. A common timing bus between the instrumentation modules and high-level software control provide this timing capability.
- Variety of signals used—a wide variety of signals which can encompass analog, digital, visual images, and motion control may be used for automating the final test in production. The wide variety of instrumentation modules encompassed in the LabVIEW environment provides a complete test environment for medical device testing.

Accessed: 02-07-2020.

Zhongyuan Wang, Yongheng Shang, Jiarui Liu, and Xidong Wu. 2012. A LabVIEW based automatic test system for sieving chips. *Measurement* 46 (11). https://doi.org/10.1016/j.measurement.2012.07.015

Dave Weisberg. 2007. Building a Test System for Medical Stents using NI LabVIEW. https://web.archive.org/web/20121018011112/http://sine.ni.com/cs/app/doc/p/id/cs-15. August 2007.

Using National Instruments LabVIEW and PXI and SCXI data acquisition hardware, Cal-Bay designed a state of the art production test system. The system tests the expansion of 30 stents at one time, using linear variable displacement transducers (LVDTs) to determine the diameter of the stents as they are heated from a supercooled state to normal body temperature.

Accessed: 02-07-2020.

George Wells and Edmund C. Baroth. 1992. dTelemetry Monitoring and Display Using LabVIEW. https://trs.jpl.nasa.gov/handle/2014/35064.

One application was created as a result of a competative [*sic*] effort between a graphical programming language team and a text-based C language programming team to verify the advantages of using a graphical programming language approach. With approximately eight weeks of funding over a period of three months, the text-based programming team accomplished about 10% of the basic requirements, while the Macintosh/LabVIEW team accomplished about 150%, having gone beyond the original requirements to simulate a telemetry stream and provide utility programs. This application verified that using graphical programming can significantly reduce software development time.

JPL TRS 1992+, Accessed: 02-07-2020.

Kirsten N. Whitley, Laura R. Novick, and Doug Fisher. 2006. Evidence in Favor of Visual Representation for the Dataflow Paradigm: An Experiment Testing LabVIEW's Comprehensibility. *Int. J. Hum.-Comput. Stud.* 64, 4 (April), 281–303. https://doi.org/10.1016/j.ijhcs.2005.06.005

Michael Wiltberger. 2007. Improving Retinal Disease Treatment with LabVIEW FPGA and Intelligent DAQ. https://web.archive.org/web/20190719014626/http://sine.ni.com/cs/app/doc/p/id/cs-698.

OptiMedica developed a system for the automated delivery of precisely timed, highly accurate laser pulses to treat retinal diseases. The company used LabVIEW FPGA and intelligent data acquisition hardware to deploy a controller for its innovative pattern scan laser photocoagulator.

Accessed: 02-07-2020.

Pawel Wrobel, M Bogovac, H Sghaier, Juan Leani, Alessandro Migliori, Roman Padilla Alvarez, Mateusz Czyzycki, Janos Osan, Ralf Kaiser, and Andreas Karydas. 2016. LabVIEW interface with Tango control system for a multi-technique X-ray spectrometry IAEA beamline end-station at Elettra Sincrotrone Trieste. *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment* 833 (07). https://doi.org/10.1016/j.nima.2016.07.030