Experience Report: Visual Programming in the Real World*

Ed Baroth, Ph.D. & Chris Hartsough
Measurement Technology Center
Jet Propulsion Laboratory
California Institute of Technology

ABSTRACT

This paper reports direct experience with two commercial, widely used visual programming environments. While neither of these systems is object oriented, the tools have transformed the development process and indicate a direction for visual object oriented tools to proceed.

This paper reports *on* real world applications of visual tools that have exposed a perhaps unexpected effect of the systems development environment: The most dramatic gains in productivity are attributed to the communication among the customer, developer and computer that are facilitated by the visual syntax of the tools. If a similar level of communications support can be achieved in the visual object oriented programming environment, even greater productivity gains then now available can be expected. In our environment of test and measurement, visual programming currently provides productivity improvements of from four to ten times compared to conventional text-based programming.

Two specific projects are discussed. The first is an application created as the result of parallel development between a visual programming team and a text-based ('C') programming team. The second application involved ground test and characterization of two aspects of a large space-based instrument. These examples were selected from over forty projects completed using these tools.

*The research described in this paper was carried out by the Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration

Experience Report: Visual Programming in the Real World*

Ed Baroth, Ph.D. & Chris Hartsough
Measurement Technology Center
Jet Propulsion Laboratory
California Institute of Technology

*The research described in this paper was carried out by the Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration

INTRODUCTION

This paper reports direct experience with two commercial, widely used visual programming environments. While neither of these tools presently are object oriented, our experience proves that it is possible to use visual programming for realistic programming applications. Because little proof of this exists in the literature, we feel this is an important contribution and an opportunity for others to gain from our experience. The tools have transformed the development process and indicate a direction for visual object oriented tools to proceed.

Our use of real world applications of visual tools has exposed a perhaps unexpected effect of the systems development environment: The most dramatic gains in productivity are attributed to the communication among the customer, deveioper and computer that are facilitated by the visual syntax of the tools. If a similar level of communications support can be achieved in the visual object oriented programming environment, even greater productivity gains then now available can be expected. In our environment of test and measurement, visual programming currently provides productivity improvements of from four to ten times compared to conventional text-based programming.

Two examples of applications created using visual programming tools will be discussed. The first is an application created as the result of parallel development between a visual programming team and a text-based ('C') programming team. Although not a scientific study, it was a fair comparison between different development methods and tools. With approximately eight weeks of funding over a period of three months, the visual programming effort was significantly more advanced, having gone well beyond the original requirements than the 'C' development effort, which did not complete the original requirements. As a result of this application, additional follow-on work was awarded to the visual programming team. The second application involved ground test and characterization of two aspects of a large space-based instrument, What was written was a flexible and reliable way to write tests that coordinated commands to and data from the instrument and test equipment.

BACKGROUND

The Measurement Technology Center (MTC) evaluates commercial data acquisition, analysis, display and control hardware and software products that are then made available to experimenters at the Jet Propulsion Laboratory. The MTC specifically configures and delivers turn-key measurement systems that include software, a user interface, sensors (e.g., thermocouples, pressure transducers) and signal conditioning, plus data acquisition, analysis, display, simulation and control capabilities.[1,2]

Visual programming tools are frequently used to simplify development (compared to text-based programming) of such systems, specifically National Instruments' LabVIEW and Hewlett Packard's Visual Engineering Environment (VEE). Employment of visual programming tools that control off-the-shelf interface cards has been the most important factor in reducing time and cost of configuring these systems. The MTC consistently achieved a reduction in software / system development time by at least a factor of four, and up to an order of magnitude, compared to text-based software tools tailored specifically to our environment.[3,4,5] Others in industry are reporting similar increases in productivity and reduction in software / system development time and cost.[6,7,8]

Our use of visual. programming provides an environment where programs (not simply user interfaces) are produced by creating and connecting icons, instead of traditional text-based programming. The icons represent functions (subroutines) and are connected by 'wires' that are paths which variables travel from one function to the next. Visual 'code' is actually the diagram of icons and wires rather than a text file of sequential instructions. Previous terms have included diagrammatic, iconic or graphical programming. The tools discussed here are based on the data flow diagram (DFD) paradigm. Visual programming is not synonymous with visual object oriented programming, nor does it imply object oriented programming.

Both systems discussed share salient characteristics, and the results from using both systems are comparable. Those results include increased productivity and customer acceptance of both our products and processes. It is felt that the key features of LabVIEW and VEE illuminate a direction that visual object oriented programming systems of the future could follow. The key feature of both systems is that they implement a visual syntax. LabVIEW and VEE blend the visual syntax's of data flow diagrams and flow charting; they also include (or are strongly influenced by) Hierarchical Input Process Output (HIPO) charting and, for VEE at least, a bit of decision tables. Both systems support the use of text for labels, notes, and expressing mathematical formulae.

The visually based syntax is the key factor in the acceptance of the tool by our customers. Our experience is that the development paradigm of a 'Requirements' definition followed by an implementation phase is obsolete in our test and measurement environment. The process now more closely represents rapid applications development (RAD),[9] and eliminates a separate implementation phase because, in general, when the requirements definition has been completed, so has the system. Traditionally, the Requirements definition is part of the communications chain that ultimately end with the developer coding at the computer. Using these tools shortens the communications chain between customer, developer and computer because coding usually is implemented interactively with the customer and developer together at the computer.

Our evidence shows that these tools facilitate communications because they provide a common expression that can be read by our customers, the developer and the computer. There are different details of each syntax that facilitate this communication, but the details are unimportant: what is important is the transformation of requirements from a statement to a dynamic conversation that results in system components as a natural outcome of the process.

VISUAL COUPLED WITH OBJECT ORIENTED PROGRAMMING

A similar characteristic of the customer's ability to interact with the model as it's being developed needs to be present in a visual object oriented programming system. Our experience with commercial visual object oriented programming systems is limited to Prograph, which we consider to be a visual extension of a text-based object oriented programming paradigm, not the implementation of a graphical paradigm like the DFD. Our sense is that the support provided by Prograph is primarily to the developer, not compatible with the mindset of our customer base, who are typically engineers and scientists with training in FORTRAN, BASIC, and recently 'C.' Few are comfortable with the details of a full object oriented programming paradigm while almost all have experience or exposure to a DFD and /or flowcharting.

Citing Yourden (page 125) "...If you can't draw a precise picture of the pattern, it doesn't exist. In an OOPL environment, the problem vocabulary and the implementation vocabulary are close enough that we might consider using the same graphical notation for 00A and OOD."

In a sense, the commercial visual programming tools discussed here have the same graphical notation for analysis, design and implementation. It is that commonality that supports the communication and it's the interactive process supported by that communication that allows the increase in productivity. It is in that regard that the visual programming tools discussed can illuminate a path for substantial gain in a visual object oriented programming environment. While it is certainly possible to have productivity improvements without visual syntax, or even without object oriented programming, if the customer cannot understand the coding language, than the interactive and dynamic advantages we've witnessed using visual programming are substantially diminished. Visual object oriented programming needs to be more than text-based object oriented programming made visual. It needs to be understandable, or at least readable, by the customer.

The combination of a graphical syntax, diagrams and patterns, specifically tailored to object oriented programming systems and accessible to the 'average customer,' then supported directly by a

tool, have the potential to produce improvements in productivity and accuracy greater than the sum of the improvements currently available from visual programming and object oriented programming individually.

VISUAL PROGRAMMING ENVIRONMENTS IN THE MTC

Our entry into visual programming environment came through vendors of test equipment. We began using these tools simply as a better way to meet our customers needs. Since the initial uses, we have become aware of more general uses of the paradigm and are now expanding our use of these tools and investigating the applicability of other visual and object oriented programming environments (e.g., Prograph). We recognize that these are not the only visual programming environments, but these are the two with which we have extensive experience to date.

For those interested in developing commercial visual object oriented programs for this market, the criteria for tool selection we consider relevant for providing the service of configuring measurement systems is given in reference 10.

National Instruments LabVIEW

LabVIEW is a graphics-based language environment for developing, debugging, and running programs. Initially designed to work with National Instruments' data acquisition and control boards that plug into a Macintosh, it has now been expanded to the PC and Sun platforms. Because it is closer to a general purpose language, however, it can and has been used for many different types of applications, including simulations and pure data processing and analysis. The first version of LabVIEW appeared in 1986 and was interpreted and monochrome. The LabVIEW 2 compiler was released in 1990 and supports color. LabVIEW 3.0 supports cross-development between the Mac, Sun and PC and 100Ic and operate essentially the same way on all three platforms.

The LabVIEW program is used to create and run LabVIEW document files that are called VI's (Virtual Instruments). The front panel of the VI appears in a window when opened and may contain an assortment of input and output objects such as knobs, dials, meters, charts, animated graphics and text boxes. Inputs are called controls and outputs are called indicators. The front panel may also contain passive graphics and text.

Associated with the VI is an icon that can be any small graphic image. 'Behind' the icon is a connector pane that can have an active region associated with each control and indicator on the front panel.

In addition to the front panel of each VI is a diagram that appears in another window when opened. The diagram contains an icon for each control and indicator. These icons are 'wired' together or to other built-in icons representing various functions and structures or to other VI files that have previously been 'collapsed' into their own icons and are referred to as subVI's (subroutines).

The process of developing a VI starts with creating a front panel with the required controls and indicators. If the intent of the VI is to function as a user interface, then emphasis may be placed on visual impact and useability. If the VI will be used primarily as a subVI, with parameters passed to and from it by other VI's, then simple numeric controls and indicators maybe used instead.

The programmer then typically works in the diagram window, developing the overall structure of the program using the built-in icons. Many of these icons are dynamic, in that they can be expanded to accommodate more inputs or outputs or re-sized to provide more area for other icons and wires. Other icons are added as needed and any that represent subVI's that haven't been developed yet can also be included as dummy functions that will automatically switch over to the actual subVI's as they become available. This feature can also be used to simulate hardware functions in the early stages of programming and then switched over to the actual hardware interfaces.

At any stage of the programming process, if the 'RUN' icon for the VI does not appear broken, the programmer can test the VI. LabVIEW will automatically compile and execute the VI. After each test, changes can be made on the front panel or the diagram or to any subVI's. Most LabVIEW programming is done with the mouse rather than the keyboard.

For more details on the LabVIEW environment, see references 11-14 and Figures 1-4.

Hewlett Packard Visual Engineering Environment (VEE)

VEE *is* HP's visual programming tool for test engineering data acquisition and analysis. It provides the ability to gather, analyze and display data without conventional (text-based) programming. This iconic application was targeted for either the engineer and scientist or the test and measurement professional. Version 1.0 was released in 1991 and Version 2.0 extends its capabilities.

Data flow diagrams, or models, are created in VEE by selecting and placing graphical objects on the screen and connecting them with lines or information paths. These models are built, modified, and run by selecting options from the menus.

VEE has three types of menus from which the user can select: full-screen, pull-down menus, that allow the user to select design elements, perform actions, and set states; design elements menus, that are specific to the selected design elements; and cascading menus, that are submenus available from the main menus and design elements menus.

Menu options are accessed by clicking and dragging the mouse. Design elements are easily manipulated in the same manner and can be collapsed into icons or double-clicked to reveal their Open views, that allow the user to view and edit a components configuration. For instance, the number of input or output terminals maybe set from a components open view.

VEE is similar to LabVIEW in that the customer and developer see a cross between a draw program and a computer aided design (CAD) package. In VEE there is Detail View and a Panel View. These correspond, roughly, to the LabVIEW front panel and diagram windows. With VEE however, the Detail View contains all the information and the Panel View has selected controls and display with none of the interconnecting wires. In LabVIEW there is no synoptic view.

Where LabVIEW provides subVI's, VEE has UserObjects and UserFunctions. A UserFunction most closely corresponds to a subroutine and a UserObject is a convenient method of encapsulation and information hiding. (UserObjects often become a raw materials source. As a matter of practice, developers will copy a UserObject and modify it rather than start from scratch. While not exactly inheritance, it is better than blank paper,) Like LabVIEW subVI's, each VEE block can be opened and closed. In this way, information can be hidden and accessed in the hierarchy with a few mouse clicks. Debugging and execution are nearly identical, Several simple to use and powerful tools support debugging. Breakpoints, data probes, highlighted execution display, and animated data flows are all available. In addition, because of the close linkage of most VEE applications with non-computer elements (e.g., test equipment), monitors of 1/0 traffic are provided.

A major difference between VEE and LabVIEW is the form of execution. With VEE, there is no compile step. VEE is not an object oriented system for the developer, however, the underlying implementation is object oriented programming (Objective C). When a VEE model is created, a large collection of objects are incarnated and interact in standard object oriented fashion.

For more details on the VEE environment, see references 13 and 15-17.

CASE STUDY NO. 1

GALILEO MISSION TELEMETRY ANALYZER

The MTC is currently supporting the redesign of the computer data system for NASA's Galileo mission to Jupiter. LabVIEW is being used to assist in the ground test of the flight software redesign as one of a series of tools used to configure measurement systems.

The MTC was approached to create a telemetry analyzer. A similar task was given to another group using text-based coding ('C') on a Sun workstation and a single-board computer. Both groups were given equal time and funding. The purpose was to determine if the LabVIEW software environment could perform telemetry stream decommutation and decoding and to verify advantages of visual over text-based programming in the time to create and modify code.

Two Macintosh computers were used, a Mac IIfx and a Quadra 950. The Mac IIfx was used to generate the telemetry stream using an interface board's digital to analog channels. The Quadra 950 functioned as the telemetry analyzer using the analog to digital converter on a similar board to capture a data channel triggered by a clock channel.

The original requirement and expectation for the task was to use one computer for simulation and analysis of the telemetry channel (using global variables) but because that requirement was met long before the deadline, the actual generation of the telemetry channel was done. The separation of

the generator and analyzer into two separate computer systems was done to approximate more closely the real environment and to allow more precise measurement of performance parameters.

The telemetry data stream consists of a two-line serial interface (data and clock). Although it only needed to operate at 200 bits per second, it was tested at up to 5000 bits per second to measure the CPU margin.

Telemetry Generator

There was no actual requirement for a telemetry generator, only an analyzer. The generator was created to test the analyzer. Figure 1 is the Telemetry Generator Sequence. The source of the data is predetermined 'Predict Tables' containing random bytes. For each instrument there is a separate table. These tables are stored in both the generator and the analyzer computers, The overall approach is that each instrument sends packets of up to 220 bytes at random intervals of time and eventually these packets are picked up by the analyzer and verified in its Predict Tables. Packets that are received out of sequence or are not present in the Predict Tables will register as errors. It is a verification of the transmitting and receiving process, not the data itself.

The entire telemetry scheme can be thought of as having an independent channel for each instrument. Packets from an instrument are not actually sent until enough are received to fill a segment of 223 bytes (including some overhead). A Reed-Solomon error-correcting code of 32 bytes is appended to each segment and when eight segments are assembled (not necessarily from the same instrument), an eight-byte PN Sync word is attached. The bytes are sent starting with the PN Sync word and continue through the first byte of each segment followed by successive bytes of each segment. Before being sent over the telemetry channel, the stream is run through a convolution algorithm that provides additional error-correcting capabilities but doubles the number of bits.

The implementation of the various algorithms (e.g., Reed-Solomon and Convolution Coders) has traditionally been done in hardware using shift-registers, ex-OR gates and counters. The close analogy of LabVIEW's icons to actual circuitry enabled easy and straight forward implementation of otherwise complex coding.

Figure 2 is the user interface (LabVIEW Front Panel) for the telemetry generator, The range of packet sizes and time intervals can be specified for each instrument as well as the data rate. As each packet is generated, it is displayed on the strip chart as a dot. For example, instrument (Instr ID) = O attempts to output a packet of between 2 and 4 bytes every 1/6 of a second (1 tick = 1/60 second), while ID = 1 attempts to output packets containing between 20 and 132 bytes every 1 to 2 seconds. These rates are adjusted by simply clicking on the up or down arrows, but are limited by the Bit Rate set on the front panel. Two types of errors can also be created to test the diagnostic capabilities of the analyzer. They are initiated by clicking the buttons.

Figure 3 is the actual program (LabVIEW diagram) for the telemetry generator. It is responsible for temporarily storing the packets from each instrument until enough are available for a segment. A detailed explanation of the diagram is given in the Appendix.

After a complete telemetry frame is assembled, the bits are passed through a convolution coder that doubles the number of bits. The bits are eventually doubled again (so that each bit is present for two clock times) and used to control two voltage levels (zero and five volts) on one charnel of the double-buffered digital to analog converter. The other charnel generates an alternating bit pattern to form the clock.

Telemetry Analyzer

The Telemetry Analyzer uses the double-buffered analog input with samples taken on each falling edge of the clock. Each analog sample is immediately converted to a Boolean by a simple comparison test and passed through the convolution decoder.

The remainder of the Telemetry Analyzer basically performs the same functions that the Generator performs but in the opposite order. The same Reed Solomon calculations are done on each segment but instead of correcting errors they are simply flagged on the analyzer user interface (LabVIEW front panel, Figure 4). The header information from the packets is then stripped away, the data compared to the Predict Tables and appropriate error flags set. It has front panel logging turned on so that the status "from each frame can be saved to disk. Each frame will have status information for eight segments including the Instrument ID, the current sequence number, the previous sequence number,

error flags to indicate if the Reed-Solomon code is incorrect, if the sequence is not properly incremented, if the packet bytes were found in the Predict Table but not in the proper place, and if the packet bytes were not found in the Predict Table. The strip chart indicates which Instrument ID's each of the eight segments goes with but it is not useful for data logging since only the last segment of each frame will be logged. Several other parameters relating to the frame itself *are* also indicated at the top of the front panel including the frame count, the number of bits found before the PN Sync code (which should always be zero after the first frame), the current time, the actual data rate, and finally the percentage margin assuming a data rate of 2000 bits per second. All testing was done at ten times the targeted rate to save testing time.

Several utility programs were created to read the previously logged data, even while logging is in progress. Among these is a monitor to graph any of the parameters on the front panel against any of the others, e.g., to plot the bit rate against the frame count; a monitor to indicate Instrument ID utilization in the form of a bar graph; and a scrolling graph to display Instrument ID's as a function of real time.

With approximately eight weeks of funding over a period of three months, the visual programming effort was significantly more advanced, having gone well beyond the original requirements than the 'C' development effort, which did not complete the original requirements. The visual programming team worked in a very interactive mode with the customer, meeting frequently and actually 'coding' together at the computer. At times, coding of the task was actually ahead of the requirements discovery. The text-based team took the initial requirements and did not meet the customer again until their demonstration at the end of the task.

LabVIEW was able to perform advanced data analysis tasks such as telemetry stream emulation and monitoring plus display. It proved that it is possible to use visual programming for realistic programming applications. This task succeeded in convincing people in our organization that visual programming can significantly reduce software development time compared to text-based programming. As a result of this demonstration, additional follow-on work was awarded to the visual programming team.

CASE STUDY NO. 2

FLIGHT TEST FOR NASA SCATTEROMETER INSTRUMENT

To support the test of the flight electronics for the NASA Scatterometer (NSCAT) instrument, a software application to coordinate the activities of the flight instrument and a suite of fifteen test instruments and software interfaces was required. Since the test engineers did not know, in fact could not know, the details of the tests until just before the tests began (sometimes five minutes before) a very reliable yet flexible system was required. What was delivered was a very simple language processor for commanding the system elements. This processor was used to sequence pre-debugged routines in flexible ways. Intentionally, there was little flexibility in the language, as both the software and scripts were debugged with the flight instrument. Operations support was provided through a control panel that not only informed the operations team of the current activity but also allowed the operations team to run, pause, abort, or step through the current test. Operations flexibility was further provided/managed by having a set of standard scripts readily available as utilities, e.g., power on, power off, data delivery.

The system was developed, in VEE, from scratch in three months. Requirements were limited to a few sheets of paper that said, in effect, "We need drivers for these test instruments: . .." A senior designer/analyst/programmer and two knowledgeable test analyst /engineers produced two versions in the three month period. Surprisingly, there were only two days devoted to bug fixes after the development team said the system was ready for operations. Since these deliveries, this program has undergone nearly continuous revision and upgrade.

For the same project, NSCAT, another completely different application was also created using VEE. This application required the real time collection of data from a receiver as an antenna was turned, or as an radio frequency probe was moved. In addition, the use of a laser range finder had to be coordinated with the movement of the antenna. The data formats for the output were already defined by an existing set of FORTRAN programs and VEE produced this format with only slight difficulty. To satisfy the requirements, a classic Structured Design approach was taken and it was both adequate and successful. Real time performance was isolated to two modules, one for antenna movement, one for probe

movement. Adequate performance was provided from VEE and coding the real-time modules in 'C' was not required.

As a result of these efforts, there are laurels for VEE, most notably the speed and reliability that VEE enables in system implementations, and the reliability of VEE in operation. VEE does have a few bugs. There are a few problems, notably the overall performance and use of screen real estate. While there are no big holes in VEE, some 'quirky' behavior is being addressed by Hewlett Packard, notably the suspension of all activity when the console is supplying data and the suspension of internal multi-tasking when executing a UserFunction.

WHERE THE GAINS OF VISUAL PROGRAMMING LIE: REAL WORLD EXPERIENCE

The advantages / disadvantages of any programming environment are dependent on the context in which they are being used. Our specific environment is the production of measurement systems, usually under schedule pressure. In three years, the MTC has created over forty applications, from the intended *uses* of data acquisition and control to areas not originally intended including simulation, analysis, telemetry, training, and modeling. We have found productivity increases (compared to text-based tools) in all applications, domain specific or not.

It is important to understand that the MTC is not, in general, a center for software research or general purpose development. As such, the software used to create these systems is simply a tool, not a language on which to conduct studies. In the area of data acquisition, analysis, display and control, using visual programming tools simply allow the MTC to perform its function more effectively than using text-based tools.[1,2,3,13,18]

It is significant to note that these productivity gains are not the result of a basic paradigm shift. As stated, both LabVIEW and VEE directly implement hybrids of the data flow diagram paradigm. Both tools, in effect, collapse the phase called 'coding' because the diagram executes. Programming, of course, is still taking place, but there is no programming activity as it is integrated with the requirements discovery and systems design process. Keeping a well-known paradigm has both positive and negative effects. In the plus column is ease of learning, ease of communication with the user, speed, and adaptability. In the minus column are mostly implementation effects, excluding the major limitation: the underlying paradigm, The quibbles are lack of effective 'find' utilities in the graphic environment, lack of zoom capability, and small idiosyncratic details that are always in newer products. Quibbles aside, without any paradigm shift, these tools have transformed system development in the MTC.

There have been few studies comparing visual with other types of programming, and those that do exist have focused on aspects that do not seem to correspond with our use of visual programming in the real world. The study by Green, Petre and Bellamy[19] compared readability of textual and graphical programming (LabVIEW). Their clear overall result was that graphical programs took longer to understand than textual ones. The study by Moher et. al.[20] essentially duplicated the study by Green et. al. but compared petri-net representations with textual program representations. They duplicated some of the earlier results, but did find areas where the petri-net representation was more well suited, albeit with reservations.

Both studies focused on experienced users of visual or textual code. In neither study was the time to create or modify the programs discussed. It is in these areas, that of user (not programmer) experience and time to create and modify programs, that we find advantages in visual over textual programming in our real world.

Our customers are mostly engineers and scientists with limited programming experience with either visual or text-based code. Most, if not all, understand data flow diagrams, so the question becomes one of which representation is easier to understand with little or no prior experience. We have consistently found users with little or no experience in LabVIEW or VEE could 'understand' at least the process, if not the details, of the program. In fact, we usually program together with the customer at the terminal, and they follow the data flow diagrams enough to make suggestions or corrections in the flow of the code. It is difficult to imagine a similar situation using text-based code, where someone with little or no understanding of 'C' could correct a programmer's syntax or flow. Actually, it is difficult to imagine anyone 'watching' someone else program using text-based code at all.

The study by Pandey and Burnett[21] did compare time, ease and errors in constructing code using visual and text-based languages. The programs chosen were on the level of 'homework' type tasks,

certainly not real world problems, but even at that level they did find evidence that matrix and vector manipulation programs were more easily constructed and had fewer errors using visual programming.

Using visual programming at this last stage of the coding process, however, removes much of the advantages we've seen. Once specifications are determined, it simply becomes a race to see who can type faster or who has access to more or better libraries of code or icons. The real benefit we find in using visual programming is the flexibility in the design process, before requirements have been determined. The user-programmer-computer communication is substantially improved because *of the* speed at which modifications can be made.

None of the existing studies have dealt with the ability of visual vs. text-based programming to solve real world problems, i.e., determine specifications, create, modify code and user interfaces as well as train inexperienced users to both operate and modify systems, Studies need to be done which allow creativity in constructing, testing and modifying models using visual and textual programming.

The most important advantage the MTC has found in using visual programming is the support for communications among the customer, developer and hardware that visual programming enables. This ease of communication provides the ability to go from conception to simulation of components, subsystems and systems, to testing of actual hardware and control functions using a single software environment (on multiple platforms). Modules or icons that represent simulations of instruments, processes or algorithms can be easily replaced with the actual instruments or components when they become available. In our opinion, object oriented programming provides the technical ability mentioned, but without the visual component, the support for communications is not present. Visual object oriented programming systems need to do both to be truly effective.

A limitation of the discussed visual programming tools is they are based on the data flow diagram paradigm. For problems that won't yield to a data flow diagram analysis, these tools are not particularly useful. Neither tool produces a conventional test-based programming language representation of the model. For many programmers, this is perceived as a major disadvantage and in some cases precludes acceptance. The authors have not found this to be a problem in our use of the tools.

Regarding training, MTC personnel, including new hires, co-op and summer students (in Mechanical, Aerospace or Electrical Engineering) have learned to program in about a month and within two have actually delivered working programs to customers.

The fundamental limits of these tools are scaling and maintenance. The MTC has produced real systems of moderate scope. We have not hit the scaling limit yet, but it is clearly present, Our systems have short to intermediate lifespans, a few weeks to a few years. Our customers tend to maintain the systems we develop for them, but have not attempted major revisions without support of the original author. If these systems had to be maintained over ten years, we're not certain that our current implementation techniques would hold up.

These tools are best used on problems that are functionally intensive, not data intensive. These systems excel in transformation and display of volatile data sources, as opposed to the maintenance of large data repositories. Currently they are not appropriate for image data, although they could be extended into that arena comfortably. Beyond this, we are reluctant to bias a reader away from any area: we've been successful at using them in areas that were purported to be inappropriate too many times.

CONCLUSIONS

- Both visual programming tools we have used provide comparable productivity improvement

It is apparent that the DFD environment (not simply one program from one developer) has shown real capability of reducing software development time in areas not domain specific. This productivity improvement is due primarily to the improved communication between the customer, developer and computer that the visual syntax provides. If the visualization component of a visual object oriented programming system does not support the customer, developer and computer communication, the productivity improvements associated with LabVIEW and VEE will not be present.

- Existing system development methodologies are inadequate in this new environment.

Because existing system development methodologies presume the existence of one or more coding phases, and that these phases are conducted outside the presence of the customer, they do not address the work environment that we find ourselves in. The lack of viable methodology is not a simple issue. If you simply compare coding time between a visual and text language you miss the point. Using these systems essentially blurs the requirements, design, and coding phases into a single activity. In many cases the MTC has found that it is faster to build the system using informal specifications than to write a formal requirements document to then build the system.

The environment of visual programming has changed the communication between developer and customer. Instead of communicating in writing or meetings, the definition of requirements takes place *using* visual programming while the 'code' is being diagramed. Development becomes a joint effort between developer and customer. In these working sessions, it is often the developer that waits while the customer considers what is wanted or what next needs to happen.

What to do with these patterns in the context of a conventional development model is any body's guess. This is a serious issue. There are almost no predictors of job resource requirements: by the time the traditional measures are available, the job is nearly done. Worse, it is difficult to manage these projects because there is no realistic model against which to measure progress. So far, the only measures we use consistently are measures of system behavior.

We are doing tasks that are not small, but not very large either. When we 'scale up' for larger projects, issues of predictable methods will become more serious.

- The ***visual*** aspect of these tools is not an add-on but integral to the underlying method of expression.

Both LabVIEW and VEE are tools that automate a graphic syntax already in common use. Within both are features that have been adapted from text paradigms. Where the text form is imported directly, e.g., FORTRAN or C equation expressions, it works well. When a basic text construct such as data structure has graphics components appended to a well-understood text syntax, the whole thing falls a bit flat. Some attempts to put object oriented features in a graphical language have had some of the same problems, i.e., graphics were simply added and not part of an underling graphical syntax. This is not to say that the graphics don't help, they do, it is just that the results are not as dramatic as automating graphic syntax directly,

- Without the *visualization* component of these tools in viewing program execution, the tools would be of limited or no value.

Visualization in this context is the ability to graphically communicate the state of execution of a system to the customer. This capability to see what the 'code' is doing directly is of inestimable value, The graphics description of the system without the animation would be not much more than a CASE tool with a code generator; with the animation, the boundaries between requirements, design, development, and test appear to collapse. Seamless movement from one activity focus to another makes the development different in kind, not degree. This is because we can sustain the communication among the customer, developer and computer. If there were substantial time lags in changing tools, (e.g., conventional debuggers) the conversational environment would break down.

- Failure to incorporate standard hardware drawing control capabilities places a burden on the memory (mental and paper) of the developers / maintainers of very large systems.

Managing large sets of drawings using parts lists and reference designators is not new. Configuration management support in visual languages is not yet present. The single largest problem we

face in scaling up the use of thesetools into larger systems devolves to configuration management. Presently, we have no effective answer to this problem.

SUMMARY

These applications, plus an additional forty or more, have convincingly demonstrated to us and our customers that visual programming significantly reduces system development time. The two tools discussed are effective and real. The MTC consistently finds that visual programming reduces development time by at least a factor of four, and up to an order of magnitude.

As stated, the most dramatic productivity increases we've observed occur when the communication between customer, developer and computer is facilitated by the tools. With this communication, the boundaries between requirements, design, development, and test appear to collapse, It is in that regard that the visual programming tools discussed here can illuminate the path for substantial gain in a visual object oriented programming environment. If the customer cannot understand the coding language, than the interactive and dynamic advantages we've witnessed using visual programming are substantially diminished. Visual object oriented programming needs to be more than text-based object oriented programming made visual. It needs to be understandable, or at least readable, by the customer.

ACKNOWLEDGEMENTS

The authors wish to acknowledge the contributions of George Wells towards the writing of this paper.

REFERENCES

1. *Acquisition, Analysis, Control, and Visualization of Data Using Personal Computers and a Visual-Based Programming Language,* E. C. Baroth, D. J. Clark and R. W. Losey, Conference of American Society of Engineering Educators (ASEE), Toledo, Ohio, June 21-25,1992.
2. *An Adaptive Structure Data acquisition System using a Visual-Based Programming Language,* E. C. Baroth, D. J. Clark and R. W. Losey, Fourth AIAA/Air Force/NASA/OAI Symposium on Multidisciplinary Analysis and Optimization, Cleveland, Ohio, September 21-23, 1992.
3. *Telemetry Monitoring and Display using Lab VIEW, G.* Wells and E. C. Baroth, National Instruments User Symposium, Austin, Texas, March 28-30,1993.
4. *Software Makes Its Home in the Lab,* Michael Puttre', Mechanical Engineering Magazine, October, 1992, pp. 75-78.
5. *Today's Equipment Tests Tomorrow's Designs,* Debra Bulkeley, Design News Magazine, May 17, 1993, pp. 82-86.
6. *Automated RF Test System for Digital Cellular Telephones, G.* Kent, Proceedings from NEPCON West '93, Anaheim, CA, February 7-11, 1993, pp. 1055-1064.
7. *Sequential File Creation for Automated Test Procedures,* J. R. Henderson, Proceedings from NEPCON West '93, Anaheim, CA, February 7-11,1993, pp. 1065-1077.
8. *Cutting Costs the Old Fashioned Way, S. C.* Jordan, Proceedings from NEPCON West '93, Anaheim, CA, February 7-11, 1993, pp. 1921-1931.
9. Decline and Fall of the American Programmer, E. Yourdon, PTR Prentice Hall, NJ, 1992, pp. 30.
10. *A Survey of Data Acquisition and Analysis Software Tools, Part 1,* E. C. Baroth et. al., Evaluation Engineering Magazine, October 1993, pp. 54-66.
11. *Diagram Compilers Turn Pictures info Programs,* Charles H. Small, EDN Special Report, June 1991, pp. 13-20.
12. *Graphical Programming for Windows/Sun,* Evaluation Engineering Magazine, September 1992, pp. 60-63.
13. *A Survey of Data Acquisition and Analysis Software Tools, Part 2,* E. C. Baroth et. al., Evaluation Engineering Magazine, November 1993, pp 128-140.
14. National Instruments Catalog, 1994, pp 17-112.
15. *Hewlett Packard VEE Visual Engineering Environment,* Technical Data, 5091-1142EN, 1991.

16. *Complete Data Acquisition Solutions with HP VEE-Test,* Application Note, 1206-01, 5091 -1139E, 1991.

17. *Design Characterization Using HP VEE-Test,* Application Note, 1206-02, 5091-1140E, 1991.

18. *Jet Propulsion Lab Aids in Space Craft Project,* Scientific Computing and Automation, November 1993, pp 26-28.

19. *Comprehensibility of Visual and Textual Programs: A Test of Superlativism Against the 'Match-Mismatch' Conjecture,* T. R. G. Green, M. Petre and R. K. E. Bellamy, Fourth Workshop on Empirical Studies of Programmers, New Brunswick, NJ, December 7-9,1991, pp. 121-146.

20. *Comparing the Comprehensibility of Textual and Graphical Programs: The Case of Petri Nets,* T. G. Moher, D. C. Mak, B. Blumenthal and L. M. Leventhal, Fifth Workshop on Empirical Studies of Programmers, Palo Alto, CA, December, 1993.

21. Is *It Easier to Write Matrix Manipulation Programs Visually or Textually? An Imperical Study,* R. K. Pandey and M. M. Burnett, Oregon State University, Department of Computer Science, 93-60-08.

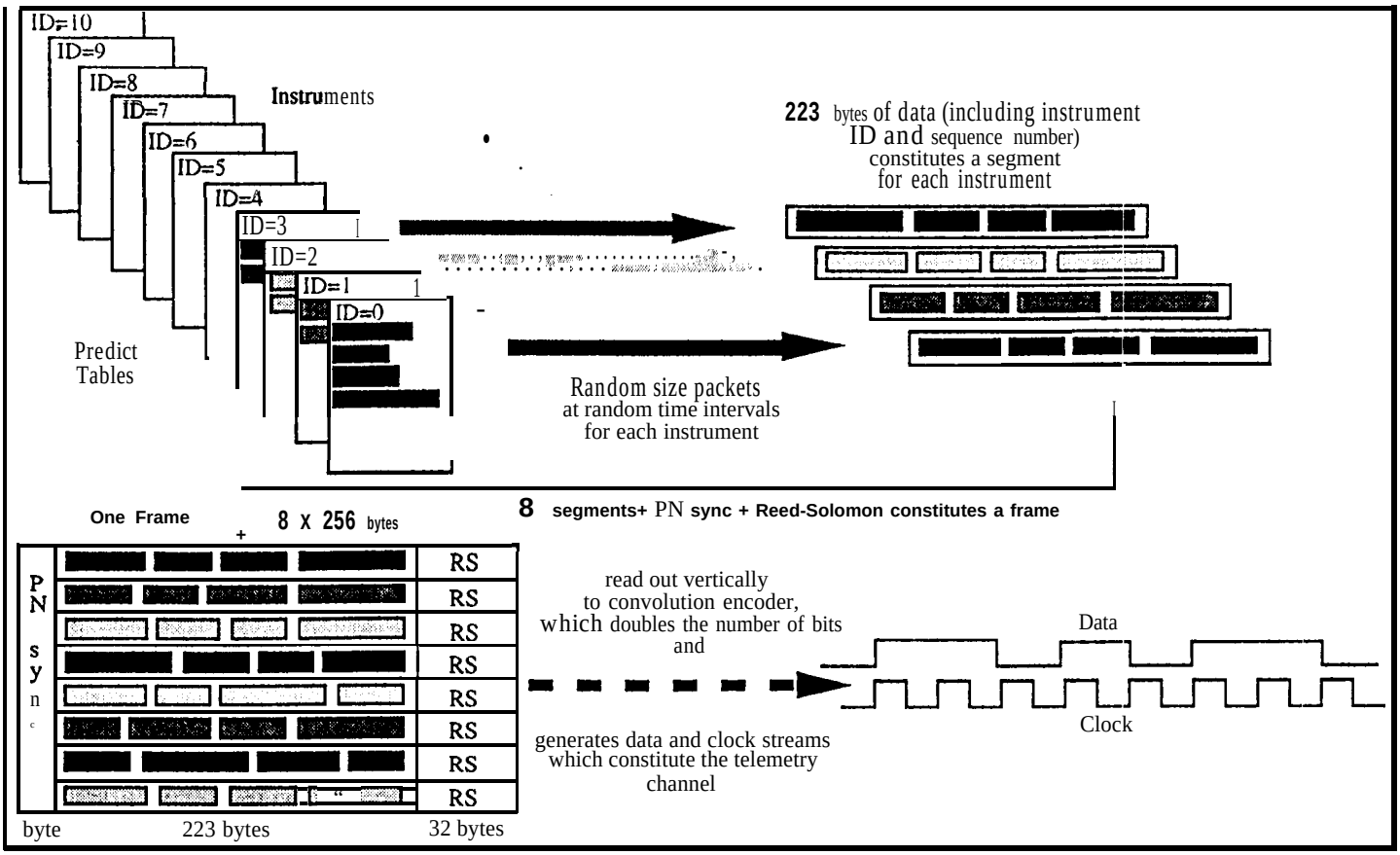

Figure 1. Telemetry Generator Sequence

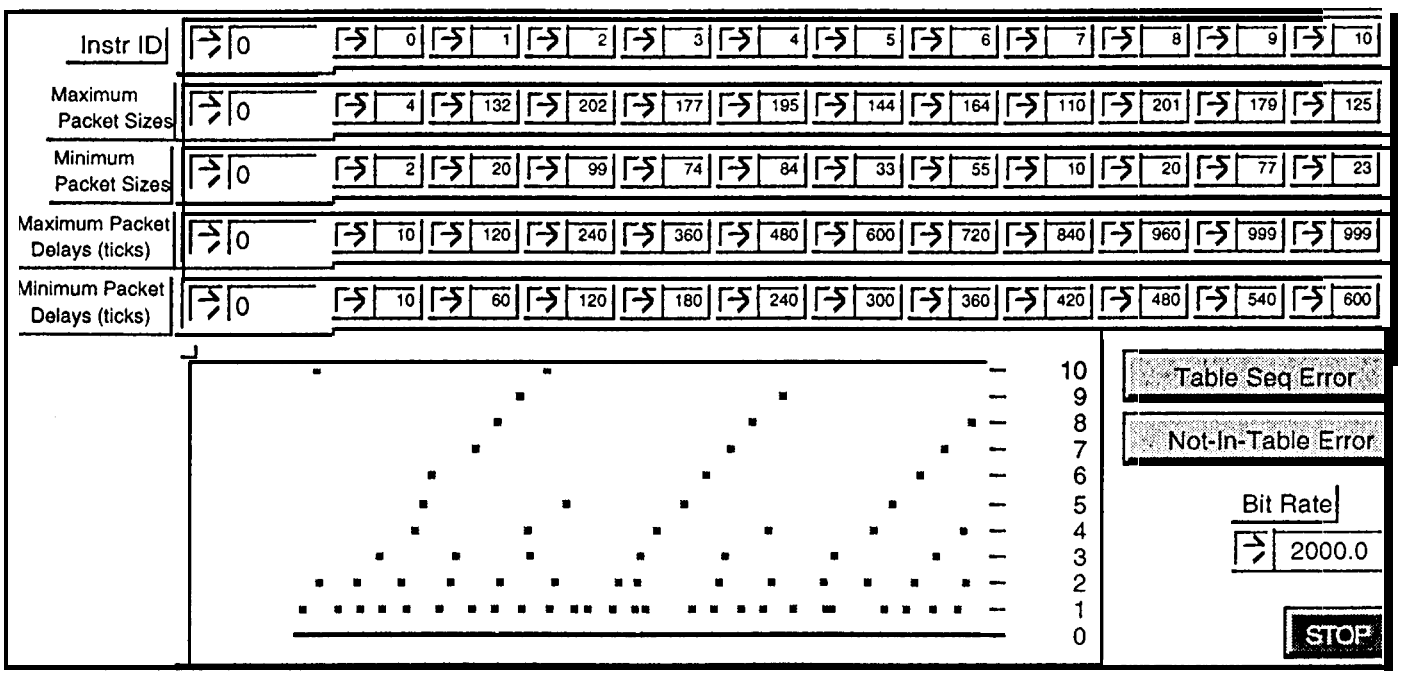

Figure 2. Telemetry Generator User Interface (LabVIEW Front Panel)

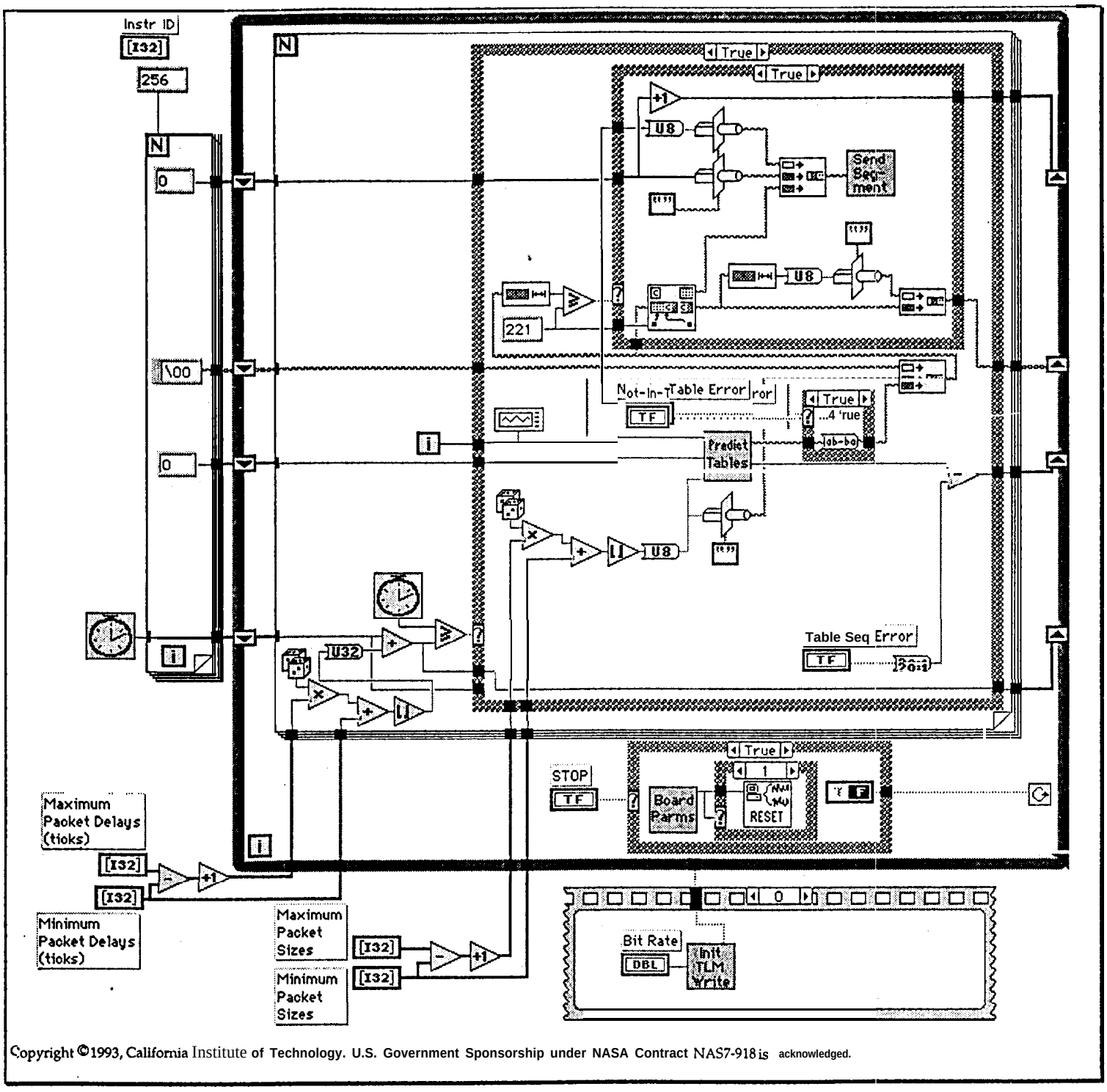

Figure 3. Telemetry Generator Program (LabVIEW Diagram)

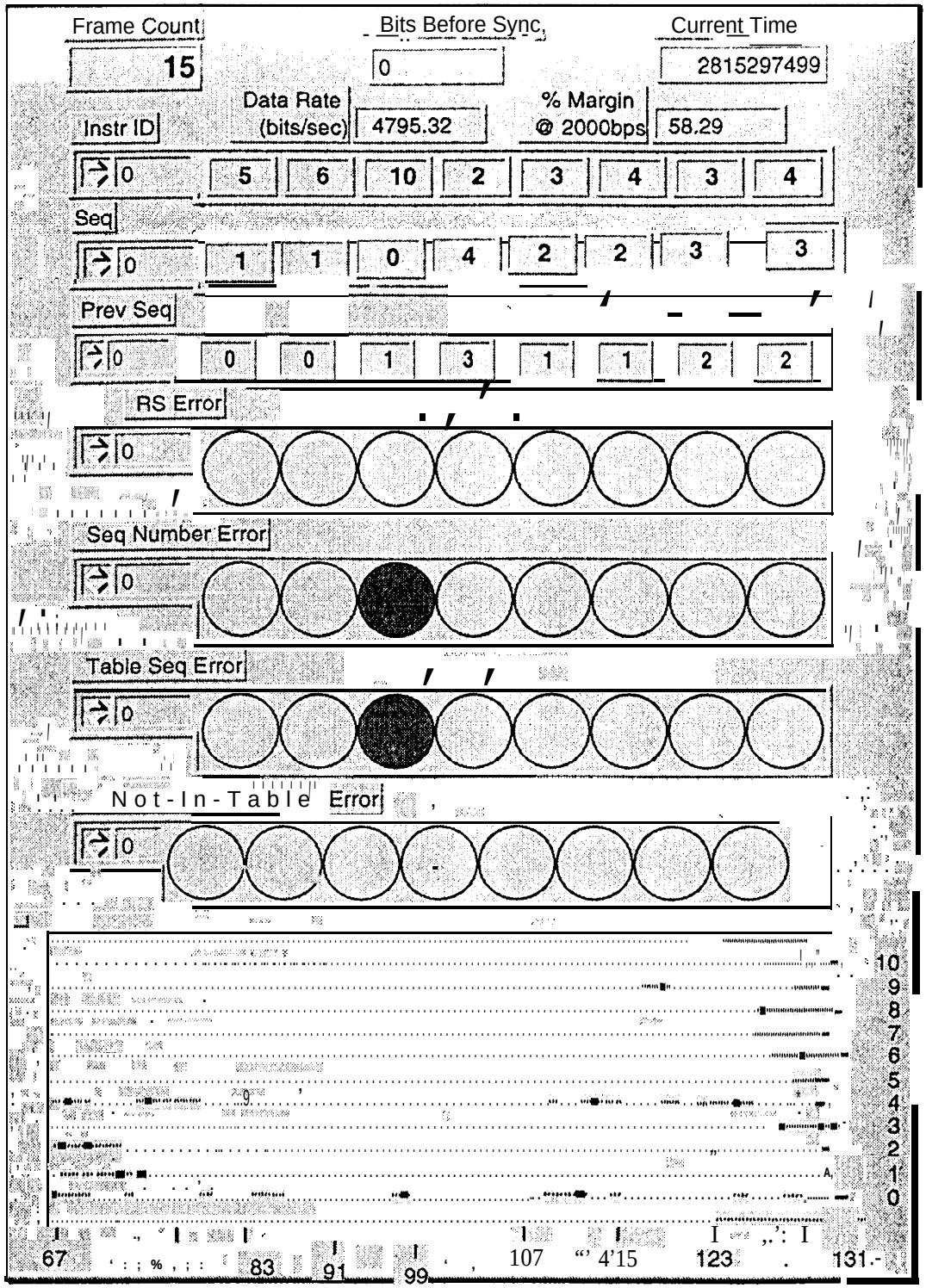

Figure 4. Telemetry Analyzer User Interface (LabVIEW Front Panel)

APPENDIX. DESCRIPTION OF FIGURE 3

Figure 3 appears exactly as it does in the VI's diagram (program) window. Each of the controls and indicators on the front panel (user interface) has a corresponding icon in the diagram. The strip chart's icon, for example, appears near the center of the diagram and looks like a miniature strip chart. All of the controls' icons are a similar size and can be identified by labels next to them corresponding to their labels on the from panel. These icons disclose the type of each input: TF for Boolean, DBL for double-precision floating-point number, and [132] for an array of 32-bit integers.

All four LabVIEW structure types are also shown in Fig. 3. First, there are two FOR-loops (which look like a stack of papers with an "N" icon in the upper-left corner). The tall narrow FOR-loop on the left side of the diagram is hard wired to iterate 256 times. It generates four arrays of 256 elements each. Note how the wires get thicker as they exit the FOR-loop along its right edge to indicate that they carry arrays rather than scalars. The iteration "i" icon shown near the bottom is not wired.

The second structure type is the WHILE-loop (which appears as a large square box formed by a thick 'arrow' encircling it in a counter-clockwise direction). Everything inside this box will continue to execute until the small icon with the circular arrow in the lower right comer receives a false value (which is the state shown in the figure). This WHILE-loop contains four 'shift-registers' that are two-part devices shown by the small rectangular boxes embedded along the left and right sides of the loop. The ones on the right side are the inputs to the shift-registers and have small arrowheads pointing up to indicate that the values wired into them will propagate around the loop, on the next iteration to their corresponding outputs on the left side with the arrowheads pointing down. The values available from the outputs on the first iteration of the loop are the ones that are wired into them from the left side of the loop. The iteration "i" icon in the lower left comers of this WHILE-loop is not wired.

The third structure type is the Sequence structure and appears as a frame from movie film. An example appears near the bottom of the figure and shows its first frame (numbered "O"). A sequence structure can have any number of frames although only one appears on the diagram. 'lie programmer can display any frame with the controls at the top but all of them will execute in order when the program is run. Since this Sequence structure is used to initialize the telemetry channel, an arbitrary output from it is wired to the border of the WHILE-loop above it to guarantee that the Sequence structure will finish executing before the WHILE-loop begins. This illustrates the data-dependency inherent in LabVIEW; i.e., no structure or icon will begin to execute until all the data wired into it is available.

The fourth structure type is the Case structure that can also have any number of frames but only one of them will execute depending on the value wired into the "?" icon on its left border. There are five Case structures shown in the diagram. Most of them are controlled by Booleans and will have only two cases, True and False. The one near the bottom of the large WHILE-loop is controlled by the STOP button on the front panel and shows what happens when the user presses it with the mouse. In addition to resetting the board used for generating the telemetry signals, it provides a False constant that stops the loop from iterating again. The other frame supplies a True constant so the WHILE-loop will continue to iterate. A small amount of arithmetic is done on arrays of numbers (indicated by the thicker lines) outside the structures in the lower left comer of the diagram. This demonstrates the polymorphism of many of LabVIEWs built-in icons, i.e., the same icon can operate on different data types.

The major portion of the processing to assemble packets into segments is done in the large FOR-loop just inside the WHILE-loop. Note that this FOR-loop does not have a constant wired to its "N" icon. Instead, it determines the numbers of times to iterate based on the sizes of the arrays brought into it. Although the four arrays coming into the loop from the left side have 256 elements, the four coming into the bottom only have eleven elements because the customer set them up that way and so the loop executes eleven times, once for each instrument.

The watch, dice, and arithmetic icons in the lower left corners of the FOR-loop determine whether it is time for a given instrument to generate another packet as indicated by the True state of the largest case structure. The False state simply passes the outputs from the shift registers (on the left side of the WHILE-loop) to their inputs (on the right side).

Whenever the True case is executed for a given instrument, a random number of characters are selected from its Predict Table and concatenated with any previously collected characters along with a byte containing the length of the character string. The Predict Table icon is a previously written subVI (subroutine) that has three inputs (on the left) and two outputs (on the right). The top input is the instrument number and comes from the iteration "i" icon of the FOR-loop. The middle input is an index into the Predict Table for that instrument, The bottom input is the number of characters to return. They come out of the top output and the bottom output is the new index. Note that the new index eventually becomes the index input the next time this instrument generates a packet. Since the number of characters is a numeric type, it is first converted to an unsigned byte (the U8 icon) and then type-cast to a string character type with the icon that looks like a square peg being forced through a round hole.

After the three character strings are concatenated together, their length is compared to 221. If it is at least that long, then the string is split into one part that is 221 bytes long and the other part is whatever is left over. The left over part is concatenated with a byte specifying its length and the result is sent to the shift register, (The entire original string is sent to the shift register in the other frame of the case structure if its length is 221) The part that is exactly 221 bytes long has two more bytes concatenated in front of it specifying its instrument number and its pocket number and the final 223 byte string is sent to the previously written Send Segment sub VI,

Because LabVIEW does not publish text-based code, there is no way to determine how much 'C' code this diagram is equivalent to without actually writing the code.

AUTHOR INSTRUCTIONS

Visual Object-Oriented Programming book

Form of Manuscript

There are 3 forms of the manuscript that must be submitted: hardcopy, plain ASCII text on floppy disk, and some common word processing file format on floppy disk. All three of these are needed; the first two are required, and the third is also requested if possible.

1. Hardcopy (REQUIRED):

Four paper copies of the manuscript including the original are required. Manuscripts should be type-written, double-spaced on one side of 8.5x1 1 inch white paper with 1-inch margins:

Page 1: should contain the article title, author(s), and affiliations(s). In addition, this page must provide the name, e-mail, telephone, fax, and complete mailing address of the author to whom correspondence should be sent. Any footnotes to the title or authors (indicated by *, †, **, etc.) should be placed at the bottom of page 1.

Page 2: should contain a short abstract. This abstract will not appear in the book, but will be used by the editors for placement and summary purposes.

The chapter itself should start on page 3.

2. Plain ASCII text (REQUIRED):

Submit a plain ASCII text version on a 3.5-inch Mac or PC floppy. Unix floppies cannot be processed -- if this presents technical difficulties, contact your editor and he/she will arrange for a conversion of e-mailed ASCII. Do not include formatting information or figures in this file.

3. Common word-processing file format (REQUESTED):

If your manuscript was created on a Mac or PC, submit the word processing file on a 3.5-inch Mac or PC floppy.

Section Names, Numbers, and Placement

Number your sections with Arabic numerals (e.g. ***3,3.1, 3.1.1).*** Do not exceed ***3*** levels.

The first section must be numbered 1 and named “Introduction”.

The last numbered section must be named “Summary”.

If you wish to include acknowledgments, name that section “Acknowledgments”, and place it after the Summary section. Do not number it.

The section containing the references comes at the very end and must be named “References”. Do not number it,

Fonts

Times-Roman 12 point is the only font to be used in your text. Multiple fonts will not be used within the article. Use italics where needed for emphasis, introducing new terms, etc. Also use italics if you wish to refer to variable names in the main part of your article. Do not use a different font for code.

Except for screen dumps, lettering for the figures should be in Helvetica. (Screen dumps will of course contain the actual fonts used by the system producing the screen).

Footnotes

Indicate footnotes with superscript numbers in the text, with the footnote given at the bottom of the page on which it is referred. Restart footnote numbering at 1 on each page.

Refererices

References substantiating points made in the text or citing previous important and relevant work are listed in a separate section at the end of the article. Citations in the text are accomplished by means of bracketed references as follows:

I author: [name year], e.g.: [Smith 1985]
2 authors: [name1 and name2 year], e.g.: [Smith and Jones 1985]
3 or more authors: [first name *et al.* year], e.g.: [Smith et al. 1985]

If there is more than one citation from the same year with the same authors, they may be differentiated by appending lower-case letters to the year, e.g.: [Smith 1985a].

A referenced article should be complete with all authors' names, title of the article, name of the publication, volume and number, publication date, and inclusive page numbers. For a referenced book, it should be complete with the author's name, title of the book, publisher, place of publication, year of publication. They should be listed alphabetically. Style and punctuation must be in accordance with the following examples:

[Harlick 1981] R. M. Harlick, "Edge and Region Analysis for Digital Image Data," in *Image Modeling* (A. Rosenfeld, cd.), Academic Press, New York, 1981, pp. 171-184.
[Smith et al. 1981a] J. M. Smith, K. C. Jones, and M. L. Johnson, "A Conference Paper," *Proceedings Someconference*, Skokie, Illinois, Nov. 18-20, 1981, pp. 163-170.
[Smith et al. 198 lb] J. M. Smith, K. C. Jones, and M. L. Johnson, *Somebook,* McGraw-Hill, New York, 1982.
[Watson and Gurd 1982] L. Watson and J. Gurd, "A Practical Data Flow Computer," *Computer,* Vol. 15, No. 2, Feb. 1982, pp. 51-57.

In general, the references should be accessible to the public, such as articles in standard journals and open conference proceedings. Internal technical reports should be avoided if possible.

Tables

Tables should be numbered with Arabic numerals in order of appearance in the text. They should be typed double-spaced. Each table should have a short descriptive caption. Table footnotes (indicated by superscript lower-case letters) should be typed at the foot of the table.

Figures

Figures should be numbered with Arabic numerals in order of appearance and should have short descriptive captions.

All 4 of the forms below are needed for each figure; the first two are required, and the other two are also requested if possible.

1. Copies of captioned figures must be included in the document at approximately the locations and sizes you would like them to appear,
2. For each figure, a large original is required, Use a separate page for each figure. Number it at the bottom of the page. This page will be used to photograph or scan in the figure.
3. For@ figure, if possible include a separate postscript file on the floppy disk.
4. For each figure, if possible include a file in some common file format (e.g.: PICT) on the floppy disk.