A Visual Engineering Environment for Test Software Development

Software development for computer-automated testing is dramatically eased by HP VEE, which allows a problem to be expressed on the computer using the conceptual model most common to the technical user: the block diagram.

by Douglas C. Beethe and William L. Hunt

For many years, the cost of developing computer-automated testing software has grown while the cost of the computer and instrumentation equipment required to run tests has dropped significantly. Today, in many test systems, the hardware costs represent less than 25% of the total cost of the system and software costs consume the other 75%. HP VEE was designed to dramatically reduce test software development costs by allowing the test to be expressed on the computer using the conceptual model most common to the technical user: the block diagram. This article will provide a general overview of the development of HP VEE, its feature set, and how it applies the concept of the executable block diagram. Further details of the architecture of HP VEE can be found in the articles on pages 78 and 84.

There was a time when business and finance people dreaded using a computer because it meant an extended question-and-answer session with a primitive mainframe application being displayed on a dumb terminal. Even after the first personal computers were introduced, very little changed, since the existing applications were simply rewritten to run on them. When the electronic spreadsheet was developed, business users could finally interact with the computer on their own terms, expressing problems in the ledger language they understood.

The technical community was left out of this story, not because the personal computer was incapable of meeting many of their needs, but because their problems could seldom be expressed well in the rows and columns of a ledger. Their only options, therefore, were to continue to work with the question-and-answer style applications of the past, or to write special-purpose programs in a traditional programming language such as Pascal, C, or BASIC.

Technical people often find it difficult to discuss technical issues without drawing block diagrams on white boards, notebooks, lunch napkins, or anything else at hand. This begins at the university where they are taught to model various phenomena by expressing the basic problem in the form of a block diagram. These block diagrams usually consist of objects or processes that interact with other objects or processes in a predictable manner. Sometimes the nature of the interactions is well-known and many times these interactions must be determined experimentally, but in nearly all cases the common language of expression is the block diagram.

Unfortunately, the task of translating the block diagram on the lunch napkin into some unintelligible computer language is so difficult that most technical people simply cannot extract real value from a computer. Staying up on the learning

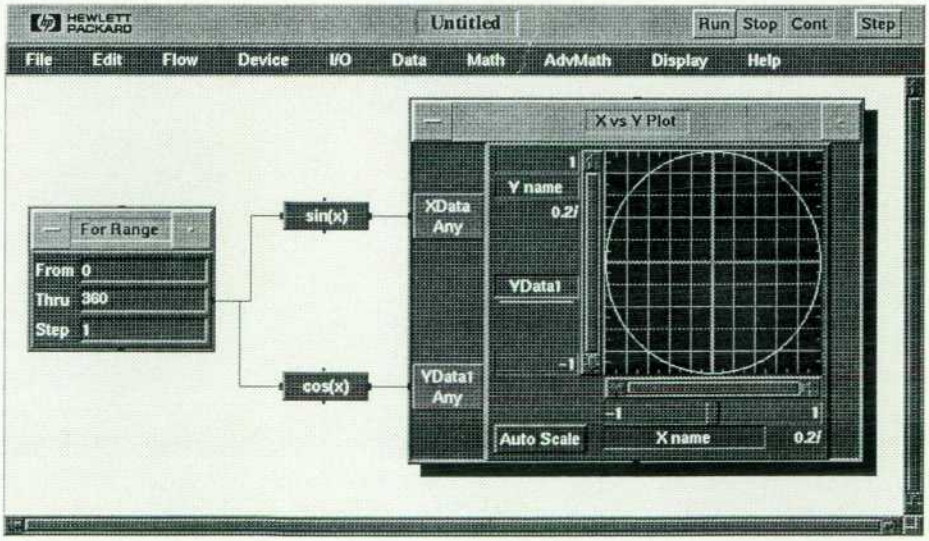

Fig. 1. A simple HP VEE program to draw a circle.

curve of their own problem domain is a sufficient challenge that embracing a whole new learning curve (programming) just to translate problems for the computer's benefit hardly seems worth the effort. While it is true that many wonderful solutions to certain kinds of problems have been generated over the years, most of the potential usefulness of computers has never been realized. In many cases, a good calculator is still the best bet because it makes a manual solution relatively easy to compute.

What is HP VEE?

HP VEE, Hewlett-Packard's visual engineering environment, is a software tool that allows users to create solutions by linking visual objects (icons) into block diagrams, rather than by using traditional textual programming statements. HP VEE provides objects for data collection, analysis, and presentation, in addition to objects and features for data storage, flow, modularity, debugging, documenting, and creating graphical user interfaces. The objects work together in the form of an interconnected network or block diagram constructed by the user to represent the problem at hand. The user selects the necessary objects from the menu, links them in the manner that represents how data flows from one object to another, and then executes the resulting block diagram. No translation to some other language. No other intermediate step.

To understand HP VEE better, consider a simple graphical program to draw a circle. By connecting a loop box, two math boxes (sin and cos), and a graph, this simple program can be built in less than one minute (Fig. 1). Although this is not a difficult task using a traditional language that has support for graphics, it is still likely that it will be quicker to develop it using HP VEE.

HP VEE eases the complexity of data typing by providing objects that can generate and interpret a variety of data types in a number of shapes. For example, the virtual function generator object generates a waveform data type, which is just an array of real numbers plus the time-base information. It can be displayed on a graph simply by connecting its output to the graph object. If its output is connected to a special graph object called a MagSpec (magnitude spectrum) graph, an automatic FFT (fast Fourier transform) is performed on the waveform (Fig. 2). By connecting a noise generator through an add box, random noise can be injected into this virtual signal (Fig. 3). If we had preferred to add a dc offset to this virtual signal, we could have used a constant box instead of the noise generator.

User panels allow HP VEE programs to be built with advanced graphical user interfaces. They also allow the implementation details to be hidden from the end user. To complete our waveform application, we can add the slider and the graph to the user panel (Fig. 4). We can adjust the presentation of this panel by stretching or moving the panel elements as required for our application.

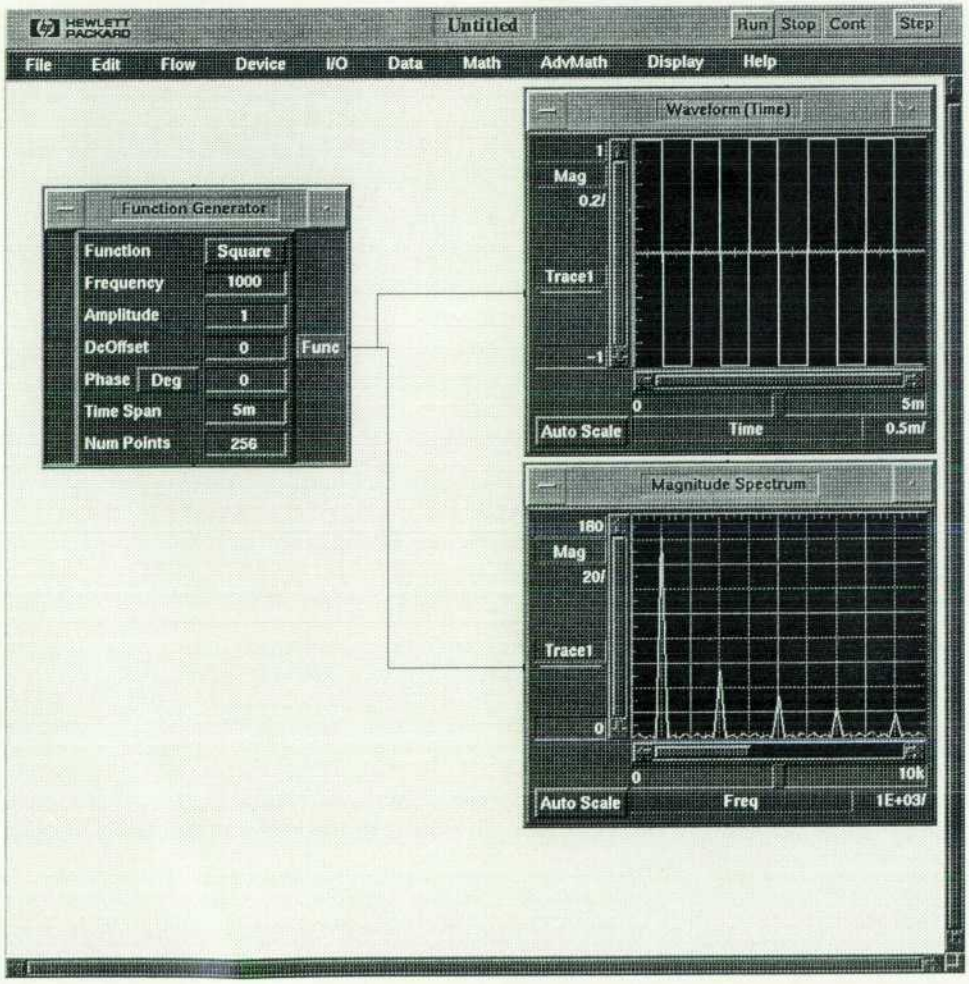

Fig. 2. A waveform displayed in the time and frequency domains.

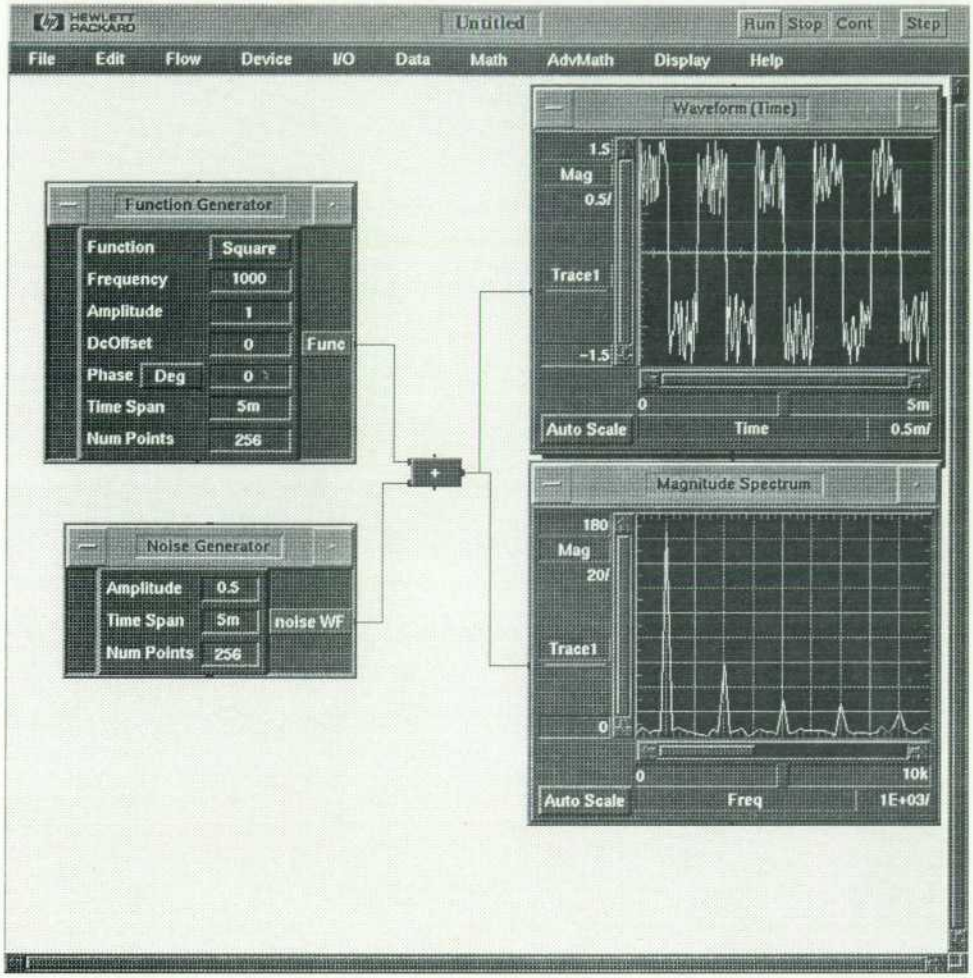

Fig. 3. Noise added to a waveform in the time and frequency domains.

This is just a trivial overview of the basic concept behind HP VEE. Other major features not covered include objects for sending data to and from files, data translation and conversion, advanced math capabilities, and data display functions. HP VEE actually consists of two products. HP VEE-Engine is for the analysis and presentation of data gathered from files or programs or generated mathematically. HP VEE-Test is a superset of HP VEE-Engine and adds objects and capabilities for device I/O and instrument control.

Development Philosophy

The team's goal for HP VEE was a new programming paradigm targeted not only at the casual user, but also at the advanced user solving very complex problems. One simple approach would have been to assign an icon to each statement in a traditional language and present it to the user in a graphical environment. The user would simply create icons (statements) and connect them in a structure similar to a flowchart. However, such a system would be harder to use than a traditional language, since the graphical program would require more display space than the older textual representation and would be more difficult to create, maintain, and modify. This would actually have been a step backward.

We decided that a genuine breakthrough in productivity could only be achieved if we moved to a higher level of abstraction to more closely model the user's problem. We therefore chose to allow users to express their problems as executable block diagrams in which each block contains the functionality of many traditional program statements. Many elements in HP VEE provide functionality that would require entire routines or libraries if the equivalent functionality were implemented using a traditional language. When users can work with larger building blocks, they are freed from worrying about small programming details.

Consider the task of writing data to a file. Most current programming languages require separate statements for opening the file, writing the data, and closing the file. It would have been relatively easy to create a file open object, a file write object, and a file close object in HP VEE. Such an approach would have required at least three objects and their associated connections for even the simplest operation. Instead, we created a single object that handles the open and close steps automatically, and also allows all of the intermediate data operations to be handled in the same box. This single To File box supports the block diagram metaphor because the user's original block diagram would not include open and close steps (unless this user is also a computer programmer), it would only have a box labeled "Append this measurement to the data file." The open and close steps are programming details that are required by traditional programming languages but are not part of the original problem.

Or, consider the task of evaluating mathematical expressions. In some common dataflow solutions, a simple operation such as 2×A+3 would require four objects and their

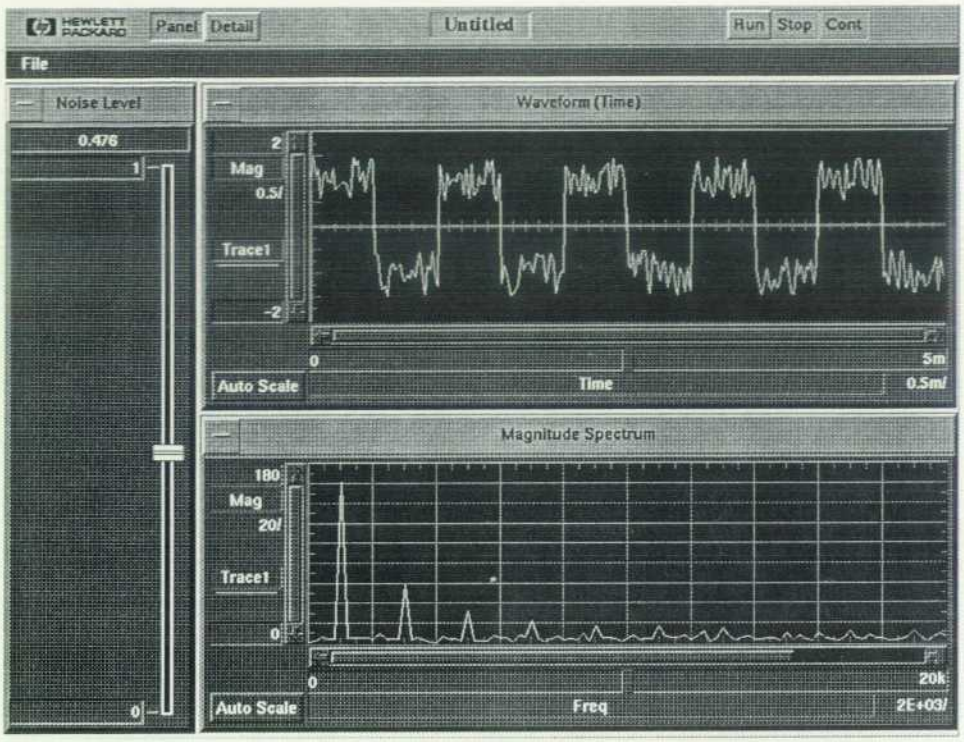

Fig. 4. User panel for waveform plus noise application.

associated connections (two constants, one add operation, and one multiply operation). Using HP VEE's formula box requires only the single expression object to solve this problem. The point of a block diagram is to show an overview of how a complex system operates without regard to implementation details. Had HP VEE been implemented without a higher level of abstraction, the resulting graphical program would have had so many boxes and lines that it would have resembled a maze rather than a block diagram.

Development Process

We followed a fairly informal development lifecycle for HP VEE. It was based on the spiral lifecycle,[1] which divides the development phase into a series of design/build/test cycles with a risk assessment before each. This worked very well for us for several reasons. Probably the most important factor was that the team was small and highly motivated. This made rigorous checkpoints and detailed design documents unnecessary since all of the team members worked very closely together toward the same goals. Another important factor was the use of an object-oriented design approach coupled with very careful design practices. This allowed us to design classes according to their interactions with the rest of the system without spending a great deal of time implementing the internals of the classes. This is important in a spiral lifecycle because during each cycle, an entire class or set of classes may need to be reimplemented. Without an object-oriented approach, this would require an excessive amount of time rewriting other seemingly unrelated parts of the system. Another successful development decision was the early incorporation of a full-time software testing team to help us with the test phases of the lifecycle.

The Search for Primitives

The initial functionality was specified by the team based on our experience as frustrated users of third-generation languages (3GLs) such as Pascal, C, and BASIC. Certain tasks appeared over and over on the "I wish there were some other way to do this ..." list. Experience had already shown that because of limited flexibility, the usual subroutine library approach did not offer the type of productivity increase being sought. However, with our executable block diagram metaphor, we felt that many of these tasks could be implemented as primitives in HP VEE while still providing the necessary flexibility.

Foremost among these tasks were data management, engineering graphics, instrument control, and integration of multiple applications. In each case we were convinced that a higher level of abstraction could be developed that would be flexible yet simple enough to require only minor configuration specification from the user in most situations.

Data Management

To tame the basic data management problem we developed the container architecture. Containers hold data, either arrays or scalars, of a wide variety of data types, and provide a rich set of mathematical intrinsics to operate on that data. Many operations such as type conversion and array processing, formerly left to the user, are incorporated into these object abstractions in a fashion that makes them relatively transparent.

Another aspect of data management involves interfacing with the file system because so much effort must be expended on it when using 3GLs. We developed objects that offer the powerful input/output capabilities of many 3GLs,

Object-Oriented Programming in a Large System

The biggest problem with a large software development effort is that there is just too much complexity for the human mind to manage. The obvious solution is to add more people to the project so that the members are not asked to manage more than their individual abilities permit. Unfortunately, the law of diminishing returns applies, since each additional team member adds a very large communication and training load on the rest of the team. In addition, there are increased opportunities for disagreement and conflict.

In some ways, development of large software systems is like one person trying to dig a canal using only a shovel. Yes, it is possible, but probably not in that person's lifetime. If more people are assigned to the task, it can be done more quickly, but only at an enormous cost. However, if equipped with the right tools (backhoes, earth movers, etc.), one person can accomplish so much that only a small number of people are required to complete the project within a reasonable amount of time.

This is exactly the idea behind object-oriented programming. By reducing the amount of complexity that one software developer must manage, that one person can be responsible for a much larger portion of the system. The result is that much higher productivity is attainable since smaller teams can be used, thereby avoiding the effects of the law of diminishing returns. Features of object-oriented programming such as encapsulation and inheritance allow one person to maintain a much larger portion of a large system than would be possible with a traditional approach.

Encapsulation is probably the strongest reason to use an object-oriented approach for a large system. Object-oriented systems encapsulate functionality by combining data and associated routines into one package (the class) and then disallowing access to the data except through one of the routines. When this is done, code outside of the class is less likely to have dependencies on the structure or meaning of the data in the class since its only access to the data is through the access routines rather than directly to the data. This allows a class to define the externally visible interface separately from the internal implementation. Because of this basic structure, a class or even an entire hierarchy of classes can be completely rewritten without affecting other parts of the system as long as the externally visible interface remains constant.

Inheritance is another reason to use an object-oriented approach in a large system. Inheritance allows a new class to be written simply by specifying additions or changes to an existing class. This means that just a few lines of added code can provide a significant increase in functionality. The other benefit of inheritance is that code reuse of internal routines is increased substantially. For example, there is only one single-line text editor in HP VEE, which is used for all single-line text entry fields. However, since it is easy to add to the behavior of the editor class through inheritance, the numeric fields that allow constant expressions as numeric input are just a very small incremental effort over the original line editor. They simply add to the "accept" mechanism at the end of an editing session and pass the typed string to the parser for evaluation as an expression before attempting to record the numeric result.

However, features such as encapsulation and inheritance do not automatically result in a system that is easier to maintain and build. Very careful design practices must be followed and the team members must be motivated to do high-quality work. Probably the most important design practice is careful partitioning of the system so that complexity in one area is not visible in an unrelated area.

An object-oriented approach coupled with careful design practices will often cause the software development effort to seem harder than with a more traditional approach. For example, in a traditional approach, if a variable in one module needs to be accessed in another module, it is easy to declare that reference directly to the compiler. In an object-oriented approach, it is common for these variables to exist only as instance variables, which are not allocated until the owning class has been instantiated. This means that the compiler cannot reference a value directly because it doesn't exist until run time. Therefore, a more complete solution must be devised to find the required value. This usually means that a message asking for the value must be sent to the object that knows the answer without ever directly accessing the variable. This sounds harder, and it is, but in the long run the resulting code is much more maintainable and extendable.

William L. Hunt
Development Engineer
VXI Systems Division

but present them to the user by means of an interactive dialog box to eliminate the need to remember syntax. Each of these dialog boxes represents a single transaction with the file such as read, write, or rewind, and as many transactions as necessary can be put into a single file I/O object.

Engineering Graphics

For engineering graphics, the task of finding a higher level of abstraction was relatively easy. Unlike data management, engineering graphics is a fundamentally visual operation and as such it is clear that a single element in a block diagram can be used to encapsulate an entire graphical display. Therefore, we just developed the basic framework for each type of graph, and we present these to the user as graph displays that require only minor interactive configuration. In this area we had a rich set of examples to draw from because of the wide variety of highly developed graphs available on HP instruments. In some cases, we were even able to reuse the graphics display code from these instruments.

Instrument Control

Instrument control is a collection of several problems: knowing the commands required to execute specific operations, keeping track of the state of the instrument, and (like file I/O) remembering the elaborate syntax required by 3GLs to format and parse the data sent over the bus. We developed two abstractions to solve these problems: instrument drivers and direct I/O.

Instrument drivers have all of the command syntax for an instrument embedded behind an interactive, onscreen panel. This panel and the driver behind it are developed using a special driver language used by other HP products in addition to HP VEE. With these panels the task of controlling the instrument is reduced to interacting with the onscreen panel in much the same fashion as one interacts with the instrument front panel. This is especially useful for modern card-cage instruments that have no front panel at all. Currently HP provides drivers for more than 200 HP instruments, as well as special applications that can be used to develop panels and drivers for other instruments.

In some situations instrument drivers are not flexible enough or fast enough, or they are simply not available for the required instruments. For these situations, we developed direct I/O. Direct I/O uses transactions similar to the file I/O objects with added capabilities for supporting instrument interface features such as sending HP-IB commands. Direct I/O provides the most flexible way to communicate with instruments because it gives the user control over all of the commands and data being sent across the bus. However, unlike instrument drivers, the user is also required to know the specific commands required to control the instrument.

To simplify the process of reconfiguring an instrument for a different measurement, direct I/O also supports the uploading and downloading of learn strings from and to the instrument. A learn string is the binary image of the current state of an instrument. It can be used to simplify the process of setting up an instrument for a measurement. A typical use of this feature is to configure an instrument for a specific measurement using its front panel and then simply upload that state into HP VEE, where it will be automatically downloaded before making the measurements. Thus, the user is saved from having to learn all of the commands required to initially configure the instrument from a base or reset state before making the measurement. In most cases the user is already familiar with the instrument's front panel.

Multiple Applications

Multiple application integration turned out to be one of the easiest tasks in HP VEE, since the inherent parallelism of multiprocess operations can be expressed directly in a block diagram. Each element of a block diagram must execute only after the elements that provide data for its inputs. However, two elements that do not depend on each other can execute in any order or in parallel. This feature, along with the powerful formatting capabilities provided for interprocess communication, allows the integration and coordination of very disparate applications regardless of whether they exist as several processes on one system or as processes distributed across multiple systems. The only object abstractions required to support these activities are those that act as communication ports to other processes. A pair of objects is available that supports communication with local processes (both child and peer) using formatting capabilities similar to those used by file and instrument I/O.

Finally, we needed to develop objects that would encapsulate several other objects to form some larger user-defined abstraction. This abstraction is available using the user object, which can be used to encapsulate an HP VEE block diagram as a unit. It can have user-defined input and output pins and a user panel, and from the outside it appears to be just like any other primitive object.

Refining the Design

While still in the early cycles of our spiral lifecycle, we sought a limited number of industry partners. This enabled us to incorporate design feedback from target users attempting real problems well before encountering design freezes. Although there were fears that such attempts would slow our development effort because of the additional support time required, we felt that the payback in design refinement for both user interface elements and functional elements was substantial.

One example of such a refinement in the user interface is the automatic line routing feature. Before line routing was added, our early users would often spend half of their time adjusting and readjusting the layouts of their programs. When asked why they spent so much time doing this, they generally were not certain, but felt compelled to do it anyway. We were very concerned about the amount of time being spent because it reduced the potential amount of productivity that could be gained by using HP VEE. Thus we added automatic line routing and a snap grid for easier object alignment so that users would spend less time trying to make their programs look perfect.

An example of a refinement in the functional aspects of the product is the comparator object. Several early users encountered the need to compare some acquired or synthesized waveform against an arbitrary limit or envelope. This task would not have been so difficult except that the boundary values (envelope) rarely contained the same number of points as the test value. Before the comparator was developed, this task required many different objects to perform the interpolation and comparison operations on the waveforms. The comparator was developed to perform all of these operations and generate a simple pass or fail output. In addition, it optionally generates a list of the coordinates of failed points from the test waveform, since many users want to document or display such failures.

Conclusion

Early prototypes of HP VEE were used for a wide variety of technical problems from the control of manufacturing processes to the testing of widely distributed telecommunications networks. Many began exploring it to orchestrate the interaction of other applications, especially where network interconnections were involved.

Current experience suggests that the block diagram form of problem expression and its companion solution by means of dataflow models has wide applicability to problems in many domains: science, engineering, manufacturing, telecommunications, business, education, and many others. Many problems that are difficult to translate to the inline text of third-generation languages such as Pascal or C are easily expressed as block diagrams. Potential users who are experts in their own problem domain, but who have avoided computers in the past, may now be able to extract real value from computers simply because they can express their problems in the more natural language of the block diagram. In addition, large-scale problems that require the expert user to orchestrate many different but related applications involving multiple processes and/or systems can now be handled almost as easily as the simpler problems involving a few variables in a single process.

Acknowledgments

We would like to thank design team members Sue Wolber, Randy Bailey, Ken Colasuonno, and Bill Heinzman, who were responsible for many key features in HP VEE and who patiently reviewed the HP Journal submissions. We would also like to thank Jerry Schneider and John Frieman who pioneered the testing effort and provided many key insights on product features and usability. More than any other we would like to thank David Palermo without whose far-sighted backing through the years we could not have produced this product.

Reference

1. B.W. Boehm, "A Spiral Model of Software Development and Enhancement," *IEEE Computer*, May 1988.

Developing an Advanced User Interface for HP VEE

Simplicity and flexibility were the primary attributes that guided the user interface development. Test programs generated with HP VEE can have the same advanced user interface as HP VEE itself.

by William L. Hunt

HP VEE, Hewlett-Packard's visual engineering environment, was developed as a programming tool for nonprogrammers. In the past, computer users were required to move into the computer's domain. Our goal for HP VEE was to bring the computer into the user's domain. This meant developing a system that operates in the way that our target users expect.

To accomplish this goal, ease of use was of critical importance. However, because most target users of HP VEE are highly educated technical professionals, simple-minded approaches to user interface design were not appropriate. For this audience, the system must be powerful and flexible, but must not become an obstacle because of overprotection.

With these constraints in mind, we decided that the primary attributes of HP VEE should be simplicity and flexibility. Learnability was also considered to be important, but we felt that no one would bother to learn the system unless it were a truly useful and powerful tool. Therefore, we felt that we could compromise some learnability in situations where a great deal of the power of the system would be lost if learnability were our primary goal. Our overall approach, therefore, was to design a system that is as natural to learn and use as possible and powerful enough that our customers would be excited about learning how to use it.

Development Guidelines

In general, simplicity is very important in a user interface because it frees the user from having to worry about unnecessary details or rules. The underlying goal of a good user interface is to increase the communication bandwidth between the computer and the user. However, this does not mean that there should be a myriad of displays and indicators. In fact, quite the opposite is true. The more things there are for the user to comprehend, the greater the chance that something will be missed. The goal, therefore, should be to reduce the amount of data that the user must be aware of and present the essential data in the simplest and most compact way possible. Similarly, any piece of data presented to the user should always be presented in a consistent way because this is known to increase comprehension dramatically.

An example of a simple way to present information to the user is the 3D look used in the OSF/Motif graphical user interface and more recently in other systems such as Microsoft® Windows. When used properly, the 3D borders can be used to communicate information about the state of individual fields without consuming any additional display space. Fig. 1 shows how HP VEE uses the 3D look to identify how fields will respond to user input. Fields that are editable are displayed as recessed or concave. Buttons and other fields that respond to mouse clicks are shown as convex. Fields that are only used as displays and do not respond to input are shown as flat. These states are very simple to comprehend because the three states are unique in the way that they look and operate. Without realizing it, users will naturally learn how to identify which fields are editable, which fields can be activated, and which fields will not respond to input. This 3D display technique allows these states to be displayed without any additional display area.

Fundamentally, HP VEE was designed around the concept of direct manipulation. This means that wherever possible, a setting can be changed by operating directly on the display of that setting. This results in a significant simplification for the user since special operations or commands are not generally required to make changes to settings. For example, the scale of a strip chart is shown near the edges of the graph display (Fig. 2). If the user wants to change the graph scaling, the limit fields themselves can be edited. It is not necessary to make a menu choice to bring up a pop-up dialog box for editing the scale. In many other systems, making any change requires a menu pick. This effectively reduces a system to a command-line interface that happens to use a mouse and menus instead of the keyboard. Such a system is no easier to use than the command line interface systems of the past.

Flexibility is more important for an easy-to-use system than for more traditional systems because there is a perception that power and ease of use cannot be combined in the same system. In the past, powerful systems have generally been

Fig. 1. A view containing a noneditable field, two buttons, and some editable fields.

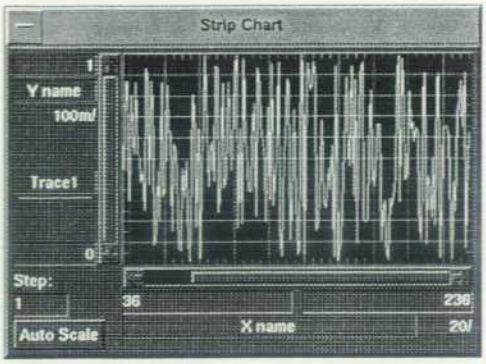

Fig. 2. Direct manipulation is useful for settings such as graph limits.

hard to use, and easy-to-use systems have generally not been very flexible or powerful. To overcome this perception, therefore, an easy-to-use system must be very powerful so that potential customers' fears can be overcome. When designing HP VEE, we were very careful to avoid limiting flexibility wherever possible. It often seemed as if we were faced with a choice between ease of use and flexibility. However, with careful consideration of the alternatives, we usually found that the more flexible approach could be implemented with an easy-to-use interface.

Flexible and powerful systems are naturally very complex because there are so many features and capabilities to remember. In these systems, it is extremely important that each area of the system operate in a way that is consistent with the rest of the system because even the most advanced users are rarely familiar with all aspects of the system. Users must be able to rely on their experience with other parts of the system to help guide them through the unfamiliar areas. For this reason, consistency was an important guideline throughout the development of HP VEE.

High performance for interactive operations is critical because users will become frustrated using a product that is too slow. Very few users will be happy if they must wait an inordinate amount of time before a particular operation is complete. The allowable time for the system to complete a task depends on the nature of the task and what the user is likely to be doing at the time. For example, a key press should be echoed back to the user within about 100 ms or so. If it takes longer, the user may press the key again thinking that the system didn't get the first one. A pop-up dialog box or menu should appear within about 500 ms. Other tasks such as loading a file can take many seconds before the user will become annoyed because of sluggish performance. We created a list of about ten different interactive operations for which we felt that performance was an important goal. On all supported platforms, many of the operations in this list such as the pop-up menus and dialog boxes are completed within the required time. Unfortunately, there are still a few operations that are completed within the specified time limits only on the very fast HP 9000 Series 700 workstations. On the other hand, we have received very few complaints about interactive performance, so our time limits may have been overly stringent.

In some situations, such as saving a file to the disk, performance simply cannot be guaranteed. In these cases, continuous feedback indicating progress being made is important. Otherwise, it isn't easy to tell whether something is happening or not. In HP VEE, all user-invoked operations that could take more than about 200 ms will result in a change to the mouse cursor. Some of these cursors represent specific activities such as reading from or writing to the disk. For other situations, a general hourglass cursor is used. Any action that is expected to take longer than one or two seconds is also accompanied by a pop-up message box that indicates that the operation is in progress.

Reducing the total number of mouse clicks, menu choices, and various other adjustments required of the user was a major challenge. Each adjustment required of the user, no matter how easy, will reduce the user's overall effectiveness. For this reason, HP VEE is designed to do as much as possible with default settings while allowing adjustments if more control is desired. Other systems often require that the user fill out a form each time a new object is selected from the menu. In most cases, HP VEE will insert default values for all settings and then allow the user to change them later if it becomes necessary.

System messages for errors and other reasons are an especially important source of difficulty or frustration for users. Most software developers seem to take the attitude of a hostile enemy when presenting the user with an error message. However, errors are seldom true user mistakes, but instead are usually triggered by failings in the system either because it misled the user or because it did not adequately protect the user from making the mistake in the first place. In many cases in HP VEE, we were able to avoid generating errors simply by restricting available choices to those that would not result in an error. For example, if a certain combination of selections will cause an error, we show them as mutually exclusive choices. In cases where such restrictions could not be used to avoid the potential for an error, we worked to simplify the interface so that users would be less likely to make mistakes in the first place. In cases where errors were unavoidable, we kept the attitude that error messages should help the user complete a task. We tried to remember that the user generally has a valid reason for performing the operation that resulted in an error.

Two kinds of messages that are common in many systems are not present in HP VEE. The first is the message "Please wait." It is irritating to users because they don't want to wait and they are being instructed to do so. The message is also unnecessary since more descriptive messages can be used instead. Messages such as "Reading from file program2" are much more informative and are not nearly so annoying. The other common message is a confirmation box that asks "Are you sure?" This is also very annoying because the user seldom initiates any operation without being pretty sure about wanting to perform that operation. There are really two reasons for asking "Are you sure?" One is to caution the user about data loss and the other is to protect against accidental requests.

In HP VEE, we solve the first situation by asking a question such as "Save changes before clearing workspace?" This has the same result as "Are you sure?", but does not sound as if the user's choice (or sanity) is being questioned.

In the second situation, accidental requests are better solved by making the input mechanisms easier to operate without error or by making corrections easy to perform. For example,

instead of asking "Are you sure?" to find out if the user really wants to delete an object, HP VEE puts the deleted object into the cut buffer so that if the user decides that a mistake was made, the paste operation can be used to restore the deleted object.

Attention to detail is very important to a user. Most systems available today lack the small details that make a system more convenient and easier to use. In HP VEE, we have attempted to pay attention to as many of these small details as possible. For example, when connecting a line to a box, an outline is displayed around the pin that would be connected if the line were released at that point. Another example of a very small detail is that HP VEE allows objects to be resized as they are being placed on the workspace for the first time. These seemingly minor details help reduce the amount of frustration that users often feel.

Program Visualization Features

In a traditional programming environment, the programmer must spend a large fraction of the development time thinking about details of the programming process including the language syntax, debuggers, and so on. Since HP VEE allows the user to think almost exclusively in terms of the problem domain, most of the development time and effort is spent on solving the problem instead of the programming details. This is the primary source of the productivity gains that many users of HP VEE have experienced. However, even though HP VEE allows programs to be expressed directly in terms of the problem, not all user-written programs will run correctly the first time. Although the so-called accidental complexities[1] of program development such as language syntax and semantics have been reduced or even eliminated, the fundamental complexities of the problem itself still remain. Therefore, once an HP VEE program is developed, it is likely that some aspect of it will not quite work as expected. For this reason, we developed several tools that can be used to visualize the execution of a program to help identify the source of any problems.

Show Execution Flow animates the execution of the program by outlining each object as it begins to execute and then erasing that outline when execution is complete. Besides helping to visualize how the program executes, this is useful when trying to understand performance issues, since an object in the program that consumes a lot of time will be highlighted for more time than other objects. HP VEE also has a timer object, which allows a more objective way to measure performance.

Show Data Flow animates the movement of data as it travels between objects in the program by displaying an icon moving rapidly along each line as data flows across it. This helps the user visualize the relationships between the data and the execution of the objects of a dataflow program. Both Show Execution Flow and Show Data Flow slow the execution of HP VEE programs so much that they are designed to be turned on and off separately.

All data in HP VEE has additional information such as size and shape associated with it. This information is maintained so that one operation can work with a variety of different data types and shapes. For example, math functions such as sin( ) can operate on either an individual number or an array of numbers with any number of elements. This is possible because the size and number of dimensions are packaged with the data. Other information such as the name of the data and mappings (the implied domain of the data) can also be associated with the data. The line probe feature allows the user to examine the data and this associated information at any time.

The execution of a program can be halted when execution reaches a particular object simply by setting that object's breakpoint flag. Breakpoints can be set on any number of objects at any time. When execution reaches an object with its breakpoint flag set, the program will pause and an arrow pointing to that object will appear. At that point the step button can be used to single-step the program one object at a time or the line probe can be used to examine data.

If an error occurs during execution of the program and no error recovery mechanism has been attached, a message will be displayed and an outline will highlight the source of the error visually. This allows the user to locate the source of the error more quickly.

User Interface for HP VEE Programs

Since a user of HP VEE should be able to generate programs with the same advanced user interface as HP VEE itself, several important capabilities have been incorporated into HP VEE to make the task of building impressive-looking programs simple.

For example, data can be entered using a variety of data entry objects. The simplest of these is a text field that accepts a single line of textual data. Numeric fields of various types such as integer, real, complex, or polar complex accept the appropriate numeric data. In addition, these numeric fields can accept constant expressions such as "SQRT(45)" or system-defined constants such as "PI." When typed, these constant expressions are immediately evaluated and the result is converted to the expected type by the input field. Since all editable fields in HP VEE share the same editing code internally, any numeric field in the system that requires a numeric entry can also accept a constant expression.

There are other more advanced mechanisms for entering data or specifying selections to an HP VEE program. Integer or real numeric input can be generated within a predefined range by using the mouse to drag the handle of a slider object. Selections from a list of acceptable values can be made using an enumerated list box, which can be displayed as radio buttons, as a single button that cycles through the list of values, or as a button that accesses a pop-up list box of choices. An HP VEE program can be designed to pause until the user is ready to continue by using the Confirm button.

Data can be displayed in a variety of ways. In addition to textual displays, real or integer numbers can be displayed on a meter object, which can show visually where a number falls within a range. Graphical displays such as XY graphs and polar plots show two-dimensional plots of data and can be interactively scrolled or zoomed. Stripcharts graph a continuous scrolling history of the input data.

All of these input and output types would have limited value if they could only be displayed when the rest of the HP VEE program with all of its lines and boxes is also visible. For this reason, HP VEE is designed with a feature called user panels, which allows an advanced user interface to be attached to a user-written HP VEE program. The user panel is built using an approach similar to many of the available user interface builders. Elements to be placed on the user panel are selected from the HP VEE program and added to the panel. The user can then move and resize these elements as appropriate for the design of the panel. Other layout options such as whether a title bar appears can also be adjusted. Since the elements on the user panel are selected from the user's program, no external code is required and the finished program is easier to build than with most user interface builder tools.

Another important aspect of an advanced human interface is the ability to hide data until the user has asked to examine it. HP VEE is designed with a feature called Show On Execute which allows HP VEE programs to use pop-up windows to hide data until a user request is received. This works by associating a user panel with a user object (similar to a subroutine in traditional programming languages) and enabling the Show On Execute feature. When the user object begins executing, the associated user panel is automatically displayed. When execution of the object is complete, the user panel is erased. Messages such as "Writing test results to file" can be displayed using this mechanism simply by putting the appropriate message on the associated user panel. If it is desirable to pause the program until the user has finished examining the displayed panel, a confirm object can be used.

Programs developed in HP VEE are highly malleable; they can be changed and adjusted as much as desired. However, in many situations it is desirable to protect the program from being changed. The secure feature in HP VEE accomplishes this by displaying only the user panel and making it impossible to alter the program or even look at it after the program has been secured.

Fig. 3. Simplified class hierarchy of HP VEE.

Using all of these features allows users to generate complete application programs with professional appearances without having to work outside of the simple dataflow environment.

Internal Architecture

Fig. 3 shows a simplified class hierarchy for HP VEE showing some of the key classes in the system and how they relate to each other in the inheritance hierarchy. The Object class is at the root of this hierarchy and implements the fundamental protocol for all objects in the system. This includes creating, freeing, and copying objects. The key subclasses of Object include View, which maintains a rectangle on the display, Container, which holds a unit of data, and Device, which represents the inner workings of an element in an HP VEE block diagram.

The fundamental visible element in HP VEE is implemented with the class called View. It maintains a single rectangular region on the display and can be transparent or a composite of other views. The View3d class adds a solid background color and a 3D border to View.

Views are organized into a hierarchy tree based on the display stacking order. The root of this tree is called DispDriver. It is always mapped to overlay the system window allocated to HP VEE. It performs all low-level screen display operations such as drawing lines and filling regions. It also handles the window system interface functions such as repaint requests and dispatching of input events. Fig. 4 shows a composite of views in a view hierarchy with some of the views labeled with the name of their associated class. Fig. 5 shows the complete hierarchy tree of the views in Fig. 4.

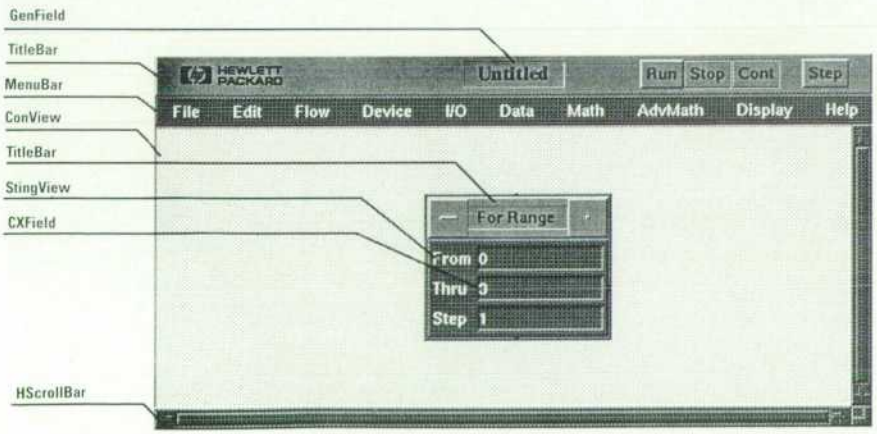

Fig. 4. A composite view with some of the component views labeled.

Fig. 5. Display hierarchy tree.

Subviews are views that are attached to another view called the superview in the display hierarchy tree. Subviews are clipped at the edges of their superview. In this way, each level of the view hierarchy tree limits the visual boundaries of all views below it. This view hierarchy indirectly describes the view stacking order, which ultimately controls which views appear to be on top and which ones are hidden.

Each view maintains a description of the region on which it is allowed to display itself. This clip region is calculated by taking its own bounds, subtracting any region that falls outside the bounds of any view in its superview hierarchy, and then subtracting any views that partially or completely cover it or any view in its superview hierarchy.

Repainting

When repainting an area that it is maintaining, a view may either use the clip region to limit the areas it actually changes on the display, or it may paint any area that it owns and then paint every view that appears closer to the user in the view stack. The full view stack repaint method has nothing to calculate or check before it begins painting itself completely and then painting anything that might be on top of it. If nothing is on top of it, then the complete stack repaint is very efficient because it is so simple. However, if there are many other views covering the view to be repainted, the full stack repaint will be very slow because of all of the unnecessary repainting that must be done. The clip region repaint method is much more efficient in this situation since only those areas that are directly visible to the user will be repainted. This means that no unnecessary repainting must be done.

Unfortunately, the clip region is at best an approximation since views are allowed to display images of arbitrary complexity (such as text) and be transparent in other areas. This gives the user the illusion that views can have any shape without incurring the performance penalties associated with nonrectangular views. It would be very costly to calculate these regions accurately, so only the approximation based on the rectangular view bounds is actually calculated. This means that repaints using the clip region method will not correctly update regions behind transparent areas of other views. Therefore, the clip region repaint method is used in only a few special cases.

Input events such as mouse clicks are dispatched to individual views in the system using a two-phase search mechanism. In the first phase, working from back to front, each view in the view stack where the event occurred asks the views in front of it to process the event. When there are no more views in front of the current view, the second phase begins with an attempt to consume the event. Working from front to back, each view in the view stack (as determined during the first phase) is given an opportunity to consume or ignore the event. If the view takes no special action, the event is passed to the next view down in the view stack. If the the view intends to consume the event, it does so by performing an action associated with the event such as indicating that a button has been pressed and then marking the event as consumed. This process continues until the event is consumed, or until the DispDriver class is given the event (this class consumes all events).

Other Classes

The visible part of each object in an HP VEE program is maintained by the DevCarrier class. DevCarrier's job is to maintain the visual appearance of all objects in the system by showing the terminal pins, maintaining the various highlights and outlines required by HP VEE, and allowing various editing operations on the user's program such as connecting lines and adjusting the sizes or positions of objects. DevCarrier does not display any object-specific information. That information is displayed by object-specific view classes, which are attached to DevCarrier as subviews.

User objects are specialized objects that are built using a subclass of DevCarrier called SubProg. SubProg maintains a visual subprogram which acts just like a normal object when viewed from the outside, but contains an internal dataflow network of its own that is just like the main program. All of the dataflow networks in HP VEE are maintained by a class called ConView (context view). It is the gray background area behind the lines and boxes in a dataflow network.

The top-level workspace environment class—IPEditor (iconic program editor)—is just a special case of SubProg and is therefore built as a subclass of SubProg. It is attached as the only subview of DispDriver and maintains the top-level editing environment. It is the same as SubProg, except that it must maintain the menu bar, run/stop buttons, and other features specific to the top level.

The context view class (ConView) maintains a list of all objects in the context and the lines connecting them. When the context view is asked to repaint itself, it first paints its background color (gray, by default), and then paints all lines in the line list. Then each HP VEE object in the context is painted according to the stacking order. If an HP VEE object falls partially or completely outside the context view's bounds, then according to the clipping rules, that view will be only partially painted or not painted at all.

The clipping and repaint algorithms allow an HP VEE program to be visually much larger than the screen space allotted to it. By adding navigation controls such as the background scroll capability, a very large dataflow network can be supported even on a comparatively small screen.

Model-View Architecture

HP VEE is organized around a model-view architecture. This is similar to the model-view-controller architecture used in other object-oriented systems except that we chose to merge the functionality of the controller into the view. The fundamental assumption in the model-view architecture is that the internal data and program elements (the models) can operate without any knowledge of or dependence on their visual representations (the views). By separating the system at this natural boundary, both the views and the models can be written more simply without any unnecessary dependencies. One feature of this architecture is that one model can be attached to any of several different views without any special support in the model. For example, a model that contains a real number can be attached to a text field or to a meter. Since the properties of the number do not change based on how it is displayed, no changes are required of the class that holds the number. However, since there are few similarities between a meter view and a text view, they need not be built with one view class.

User panels are one area that really benefit from the split between models and views. When the user selects an HP VEE object such as a slider and asks that it be added to the user panel, several things happen internally to make that happen. First, if a user panel has not been created for this program or user object, one is created. The user panel class is similar in concept to the context view class, but without support for interconnections required for dataflow networks. Next, an instance of the PanelCarrier class is created to hold a copy of the object-specific part of the slider view. This copy is created from the original and attached to the new panel carrier and to the original slider model (which is not copied). The panel carrier is then attached to the user panel view.

One of the most significant architectural impacts of the implementation of user panels is the fact that there can be many independent views attached to the same underlying model at the same time. Because of this architecture, it is easy for panels from user objects to be added as a unit to higher-level panels. This allows the creation of complex panels consisting of grouped controls and displays.

The DispDriver class is designed to be the only place where calls to the underlying window system (such as the X Window System) occur. This allows the display driver to be replaced if appropriate when porting to a new platform. During development, for example, we used a driver written to talk directly to the display card of an HP 9000 Series 300 computer because it ran so much faster than the window systems. Now that very high-performance workstations are available, this is no longer necessary.

Printing is handled simply by replacing DispDriver with the printer driver class, which knows how to perform graphics operations on a printer. The information intended for the printer is just "displayed" on the printer and no special printer support must be developed aside from the printer driver itself. This also allows the print output to match the screen display very nicely.

Acknowledgments

Building an advanced user interface is really not difficult, but it takes a great deal of thought and perseverance. It also requires support from management. We were lucky on the HP VEE team because we had managers who understood the value of a good user interface. They encouraged the team to produce the best product that we were capable of even if the schedule would be put at risk. Of course, the team members themselves were very highly motivated to produce an exciting product. John Bidwell, the HP VEE project manager, provided the leadership and management support required for our success. He was able to resist the temptation to ship the product before it was ready, and kept all of the various team members focused on the goal of a truly easy-to-use product. Sue Wolber, Randy Bailey, and Ken Colasuanno each contributed to the overall usability of the system in each of their respective areas. Jon Pennington performed usability testing and provided most of the usability feedback during development.

Reference

1. P. Brooks, "No Silver Bullet: Essence and Accidents of Software Engineering," *IEEE Computer*, September 1987, pp. 43-57.

HP VEE: A Dataflow Architecture

HP VEE is an object-oriented implementation. Its architecture strictly separates views from the underlying models. There are two types of models: data models and device models. Special devices allow users to construct composite devices.

by Douglas C. Beethe

The HP VEE dataflow programming environment was developed with the specific objective of providing an interface that would allow users to express a problem in block diagram form on the screen and then execute it directly. Dataflow programming was chosen because of its direct correlation to the block diagram models we wished to emulate.

Previous efforts in industry and academia related to dataflow programming had yielded some interesting results, but general applicability had not yet been established. Thus our early research efforts were directed primarily at the question of whether we could solve some of the problems that had plagued earlier attempts and prove general applicability.

The design and construction of HP VEE used object-oriented technology from the beginning. We had enough experience with procedural coding technology to realize that a project like HP VEE would be too complex to achieve with procedural technology. The architecture that evolved from this development is the subject of this article. The design of various elements of the underlying HP VEE architecture will be discussed as will the manner in which they work together to produce the executable block diagram as a dataflow model.

The Model-View Paradigm

One of the characteristics of the HP VEE architecture that is common to most object-oriented implementations is the strict separation between models and views. Most users are familiar with the world of views, and indeed often make no distinction between the view of an object and its underlying model.

From a functional point of view the model is the essence of an object, incorporating both the data (state variables) that gives the object its uniqueness, and the algorithms that operate on that data. In HP VEE, by definition, the model operates independently of the view, and does not even know of the existence of any graphical renderings of itself, except as anonymous dependents that are alerted when the state of the model changes (see Fig. 1).

There are many examples of applications that have views possessing no underlying functional models. Consider the various draw and paint programs, which allow the user to create very sophisticated views that, once created, are incapable of performing any function other than displaying themselves. Likewise, there are numerous examples of applications that support very sophisticated functional models but lack any ability to display a view of those models, be it for passive display of state or for active control.

Most of the scientific visualization software appearing today is used to create views of the data output of functional models that have little or no display capability. Most of the views that are seen by the HP VEE user are actually graphical renderings of the states of underlying models. In the interactive mode, access to the models is by means of these views, which communicate with their respective models to change their state, initiate execution, and so forth. For example, the view of the ForCount iterator has a field in which the user can enter the number of times the iterator should fire at run time. Upon entry, this value is sent to the underlying device model, where it is kept as a state variable. When this state variable is changed, the model sends out a signal to anyone registered as a dependent (e.g., the view) that its state has changed, and the view then queries the model to determine the appropriate state information and display it accordingly (see Fig. 2).

This strict separation between model and view might seem excessive at first, but it results in a partitioning that makes the task of generating the two different kinds of code (very different kinds of code!) much easier from the standpoint of initial development, portability, and long-term code maintenance. It also eases the job of dealing with noninteractive operations in which HP VEE is running without any views at all, either by itself or as the slave of another application. And finally, this separation eases the task of developing applications that must operate in a distributed environment where the models live in one process while the views are

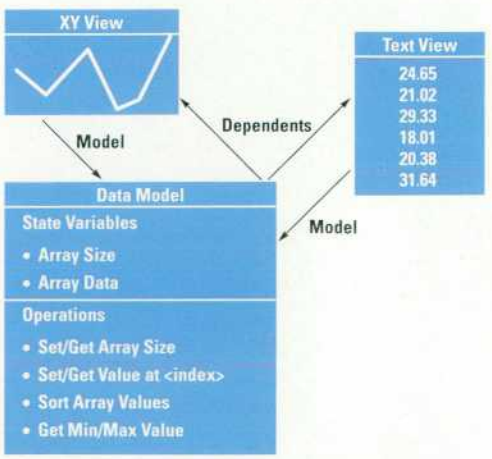

Fig. 1. Two different views of the same underlying model.

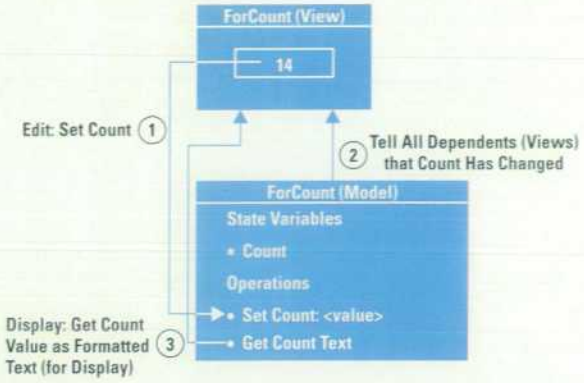

Fig. 2. Interaction of a view and the underlying model at edit time.

displayed by another process, possibly on an entirely different system. This last aspect is becoming more and more important in an application world that is taking increasing advantage of networked systems.

HP VEE itself is composed of two kinds of models. The first is the device model, which acts like a black box having inputs, outputs, and some operational characteristic that transforms the data at the inputs to the result at the outputs. The second is the data model (container), which handles the transport of information along the data lines, which interconnect devices. The data model also provides mathematical functions, which can be invoked to operate on the data, and formatting and deformatting functions, which change the representation of the data when required for display or for communication with some other application that requires the data in a different form. Both types of models will be discussed in some detail.

Data Models

The fundamental abstraction for information in HP VEE is the container object (Fig. 3). Containers can hold data for any of the supported data types: text, enumerated, integer, real, complex, polar complex, coordinate, waveform, spectrum, and record. Both scalars (zero dimensions) and arrays from one to ten dimensions are supported. In addition, the dimensions of array containers can be mapped in either linear or logarithmic fashion from a minimum value at the first cell of a dimension to a maximum value at the last cell of that dimension. This allows an array of values to have some physical or logical relationship associated with the data. For example, a one-dimensional array of eleven measurements can be mapped from 0 to 32 cm to indicate the physical relationship of the values in each position in the array to some real-world phenomenon. The first value would be at 0 cm, the next at 3.2 cm, the next at 6.4 cm, and so on.

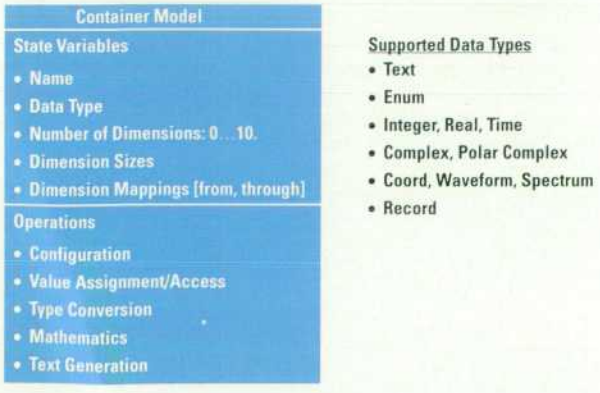

Fig. 3. Container model attributes.

One of the properties of containers that is used extensively in HP VEE is the knowledge of how to transform to another type on demand. The automatic form of this transform is allowed only for transforms that incur no loss of information. This has the effect of allowing most promotions, but disallows any automatic demotions. For example, integer can be promoted to real, and real to complex or polar complex, but complex cannot be demoted automatically to real. To do so would likely cause the loss of information that would not reappear in the promotion of that real value back to complex. An interesting special case of this is the reversible transformation between waveform and spectrum (time and frequency domains). While these data types seem to have the same irreversible relationship to each other as the real and complex types just discussed, it is a well-known fact that a reversible transformation exists between these two special types by means of the Fourier transform. For example, a 256-point waveform is transformed to a 129-point spectrum (ignoring the symmetrical values with negative frequency), and the same spectrum regenerates the original 256-point waveform by means of the inverse Fourier transformation (Fig. 4).

Another powerful property of containers is their inherent knowledge of data structure as it applies to mathematical operations. At first glance, operations such as addition and subtraction seem relatively simple, but only from the standpoint of two scalar operands. For other structural combinations (scalar + array, array + scalar, or array + array) the task requires some form of iteration in typical third-generation languages (3GLs) like C that has always been the responsibility of the user-programmer. Containers encapsulate these well-understood rules so that the user deals with, say, A and B simply as variables independent of structure. When any of the nontrivial combinations is encountered, the containers decide among themselves if there is an appropriate structural match (scalar with any array, or array with conformal array) and execute the appropriate operations to generate the result.

Other more complicated operations with more robust constraints (e.g., matrix multiplication) are handled just as easily since the appropriate structural rules are well-understood and easily encapsulated in the containers. These properties aid the user in two ways. First, the user can express powerful mathematical relationships either in fields that accept

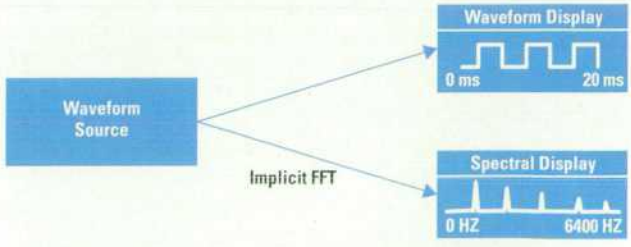

Fig. 4. Automatic transformation of a time-domain waveform (e.g., 256 real values, 0 to 20 ms) to a frequency-domain spectrum (129 complex values, 0 to 6400 Hz).

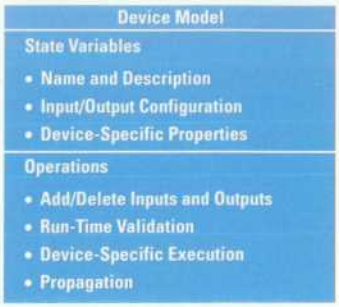

Fig. 5. Attributes of a simple device model.

constant expressions or in any of several delayed-evaluation fields (e.g., Formula, If/Then, ...) without having to deal with the cumbersome iteration syntax of 3GL programming. This by itself has the pleasant side effect of eliminating much if not most of the iteration in many applications, compared to their 3GL equivalents. Second, the interconnection of the various objects that make up a model in HP VEE is much simpler when any of the inputs is constrained to a specific data type. Since the containers know how to respond to most requests for type change, the user is freed from the cumbersome task of explicitly changing (casting) the original type to the required type. For example, the inputs to a spectral display that requires a spectrum input will not disallow connection to a waveform (time-series data) because the output supplying the waveform will transform it to a spectrum on demand at run time. This same capability is used during the evaluation of any mathematical expression, thus allowing the user to intermix types of operands without explicit type casting.

Device Models

Fig. 5 shows the attributes of a simple device model. Each device can have its own inputs and outputs. Many have user-controllable parameters that are accessed as constants through the panel view of the device or as optionally added inputs. In general, the device will execute only when each of the data inputs has been given new data (including nil data). Thus the data inputs to any given device define a system of constraints that control when that device can execute. This turns out to be quite natural for most users since the data relationships that are depicted by the data lines that interconnect devices generally map directly from the block diagram of the system in question, and often are the only form of constraint required for the successful execution of a model.

There are numerous cases, however, where an execution sequence must be specified when no such data dependencies exist. Such cases typically fall into two categories: those where there is some external side effect to consider (communications with the real world outside my process) and those that deal explicitly with real time. To deal with this situation we developed the sequence input and output for each device (on the top and bottom of the device, respectively), as shown in Fig. 6. The sequence output behaves like any other data output by firing after successful execution of the device except that the signal that is propagated to the next device is a always a nil signal. Likewise, the sequence input behaves like any other data input with one exception. When connected it must be updated (any data will do, even nil) along with any other data inputs before the

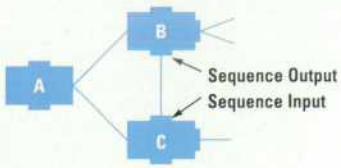

Fig. 6. While B and C both need the data from A, the sequence connection between B and C will cause C to execute after B.

device will be allowed to execute, but unlike other data inputs, connection is not required. Thus any time it is required that A must execute before B where no other data dependencies exist between the two devices, it is sufficient to connect the sequence output of A to the sequence input of B.

For users who have already been introduced to programming in third-generation languages such as Pascal, C, or BASIC this can require a paradigm shift. Experience with such users has shown that they are often preoccupied with sequencing (since 3GLs almost universally use control-flow paradigms) and have a difficult time at first believing that the data constraints represented by the lines that interconnect the devices are sufficient to define a robust sequence of execution. It is only after using the system for a time that they are weaned away from this need to sequence each and every device explicitly and begin to feel comfortable with the dataflow paradigm.

Contexts

Several types of devices are supplied as primitives with HP VEE, including those used for flow control, data entry and display, general data management, mathematical expressions, device, file, and interprocess I/O, virtual signal sources, and others. There is also a mechanism that allows users to construct special devices with their own panels and a specific functional capability. This device is known as a UserObject and is essentially a graphical subprogram.

UserObjects (Fig. 7) encapsulate networks of other devices (including other UserObjects) and have their own input/output pins and custom panel displays. Viewed as a single collective object with its own panel, each UserObject operates under the same rules as any primitive device: all data inputs must be updated before the UserObject will execute its internal subnet. Each UserObject will contain one or more threads, which execute in parallel at run time. In addition, threads in subcontexts (hierarchically nested contexts) may well be

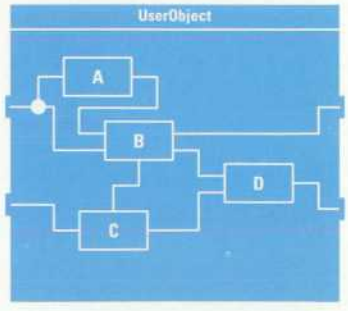

Fig. 7. A UserObject encapsulates a subnetwork of other objects into a single larger object with its own inputs and outputs.

running in parallel with their host threads in their parent contexts.

UserObjects can be secured such that the user of the device can access only the panel and not the internals. In this form the UserObject is almost indistinguishable from any primitive device. This capability allows developers to create arbitrary devices that can be archived in a library for later access by users, who can treat these devices as true primitives in their application.

Threads

Devices that are connected to each other within the same context form a single thread of execution. One of the inherent advantages of dataflow programming is the ability to support multiple independent threads of execution with relative ease (see Fig. 8). This becomes particularly useful when interacting with the rest of the world, since independent monitoring operations ("Has that message arrived yet?") can proceed in parallel with related operations. In typical 3GLs such operations require elaborate schemes for enabling interrupts and related interrupt service routines. Most who have dealt with such code as inline text can attest to the difficulty of maintaining that code because of the difficulty of easily recreating the relationship between parallel operations once the code has been written.

Several devices were developed especially for thread-related activities. One of these is the Exit Thread device, which terminates all execution for devices on that same thread when encountered. Another is the Exit UserObject device, which terminates all execution on all threads within the context in which it is encountered.

Certain devices have the ability to elevate a thread's priority above the base level to guarantee that thread all execution cycles until completion. One such device is the Wait For SRQ device (SRQ = service request), which watches a specified hardware I/O bus in anticipation of a service request. If and when such a request is detected, this device automatically elevates the priority of the subthread attached to its output so that all devices connected to that subthread will execute before devices on any other thread (within this context or any other context) until that subthread completes.

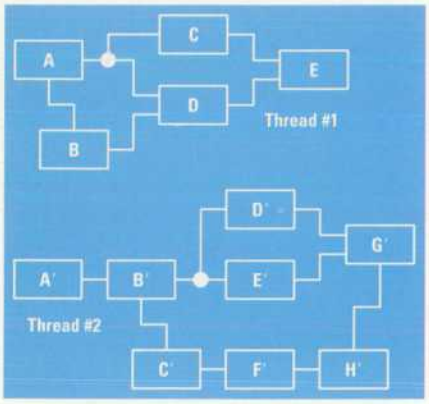

Fig. 8. Any context (e.g., a UserObject) can contain one or more threads, each of which executes independently of all others within that context.

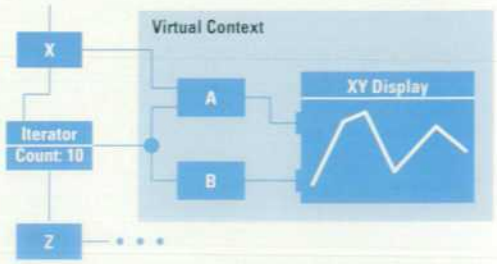

Fig. 9. Objects A and B and the XY display will execute 10 times each at run time as the iterator fires its only data output (right side) 10 times before firing its sequence output (bottom). The data from the output of X is reused for the last 9 of the 10 executions of A (active data rule).

Although it is not specifically thread related, a similar capability exists for exception service. At the time an exception is raised (e.g., an error occurs), all other devices on all other threads are suspended until an exception handler is found (discussed later).

Propagation: Flow of Execution

From an external point of view, the determination of which devices can execute is a simple problem of finding out which devices have had all of their inputs updated. From an internal point of view, the problem is a bit more difficult. To prevent infinite feedback the general rule for dataflow programs is that each device can execute only once per activation of the context in which the device resides. On the other hand, it was felt from our earliest prototypes that having iteration occur within some subgroup of devices in a context was superior to dropping down into a subcontext multiple times to accomplish the same thing, especially for nested iteration.

Thus we were faced with the problem of allowing groups of devices to execute multiple times within a single activation of a context. Identification of these devices could only occur at run time as they appeared on the subthread hosted by the primary output of an iterator. To deal with this we developed the virtual context, which is defined not by the user but by the system (see Fig. 9). At run time, the devices that are executed on the subthread hosted by an iterator are remembered. Then, just before the next firing of the iterator (since an iterator generally fires its output more than once for each execution of that iterator), the devices in this virtual context are selectively deactivated separately from the other devices in the context. This allows them to be re-executed when the iterator fires again by the normal rules of propagation.

One other side effect of such iteration is that any data being supplied to a device within the virtual context by a device that is outside that virtual context is going to be delivered only once to the device within the virtual context. Thus new data is supplied to the inputs as required on the first iteration, but on all subsequent iterations no new data arrives. One could solve this by using a special intermediary Sample&Hold device, but a simple extension to the rules of propagation turned out to be much easier. The extension,

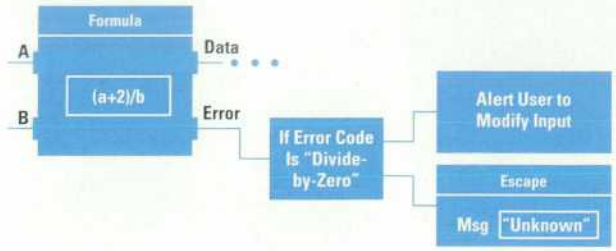

Fig. 10. The special error output will fire in lieu of the data output if any error is encountered while evaluating the formula. The value posted at the error output is the error code number. This allows the user to decide how to handle the situation.

known as the "active data rule," says that data from any active output of a device that is currently active (executed, but not yet deactivated) can be reused. This has essentially the same effect as the Sample&Hold but is much less error-prone.

The goal in all of this is to create a scheme of execution that does not require the user to specify a sequence of execution with explicit device-by-device triggering as is common in the world of digital design. In addition, we wanted execution to proceed as if the entire network were running on a multiprocessor architecture with true parallelism. On a typical uniprocessor machine only one primitive device is actually drawing cycles from the processor at any one instant, but the overall effect is as if all devices both within the same context level and across other levels of the network hierarchy are running in parallel.

Asynchronous Operations

For some devices we found a need to invoke certain operations programmatically that were peripheral to the general operation of the device, such as AutoScale or Clear for an XY graph. While the primary function of the graph is to construct a graph from the data present at the synchronous data inputs, operations such as AutoScale could happen at any time. A different class of inputs that were not incorporated into the general scheme of propagation was needed to initiate these asynchronous operations. Thus we developed the control input, which when updated at run time will perform its assigned function within the associated device regardless of the state of any other input on the device.

Exception Management

Exception (error) management could have been approached from a number of different points of view, but it proved most effective to implement a strategy based on an optional output that fires if and only if an untrapped exception is raised from within the scope of that device (Fig. 10). For primitive devices this allows the user to trap common errors such as division by zero and deal with possibly errant input data accordingly. In each case a number (an error code) is fired from the error pin and can be used by the ensuing devices to determine just which error has occurred. If the decision is not to handle the error locally, the error can be propagated upward with the Escape device, either as the same error that could not be handled locally or as a new user-defined code and message text, which may be more informative to the handler that eventually owns the exception.

Hierarchical exception handling is possible because an error pin can be added to any context object (UserObject) to trap errors that have occurred within its scope and that have not been serviced by any other interior handler. If the exception pops all the way to the root context without being serviced, it generates a dialog box informing the user of the condition and stops execution of the model. To enable the user to locate the exception source, the entire chain of nested devices is highlighted with a red outline from the root context down to the primitive device that last raised the exception.

Acknowledgments

Much of the conceptual framework for HP VEE in the early stages came from lengthy discussions with John Uebbing at HP Labs in Palo Alto. His insights and questions contributed significantly to many elements of the underlying structure which eventually matured into the HP VEE product. John's vision and imagination were invaluable. I would also like to thank several members of the design and test teams whose continued feedback concerning the functional aspects of the product proved equally invaluable: Sue Wolber, Randy Bailey, Ken Colasuonno, Bill Heinzman, John Friemen, and Jerry Schneider. Finally, I would like to thank David Palermo who in his position as lab manager provided the resources and direction to see this project make it from the first conceptual sketches to the real world. No project of this nature can succeed without such a sponsor.