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

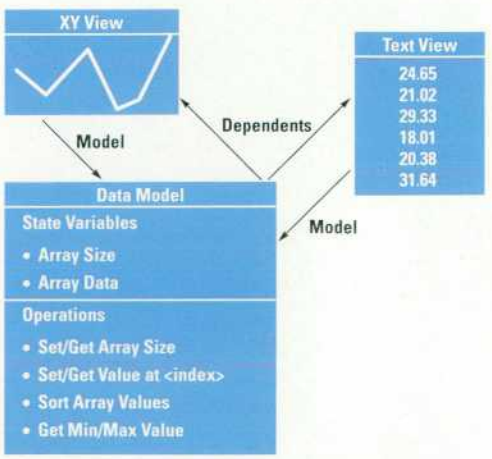

Fig. 1. Two different views of the same underlying model.

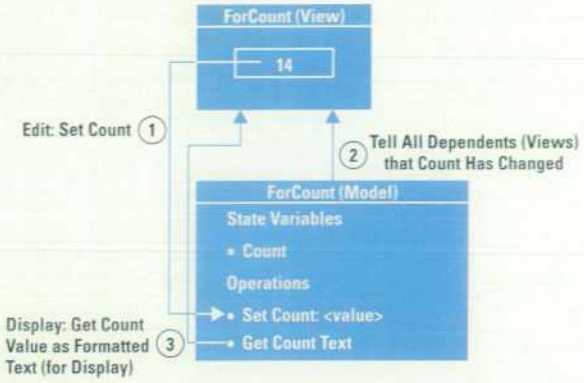

Fig. 2. Interaction of a view and the underlying model at edit time.

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.

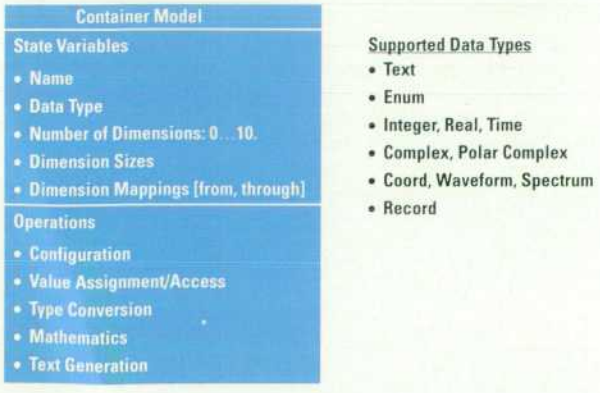

Fig. 3. Container model attributes.

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

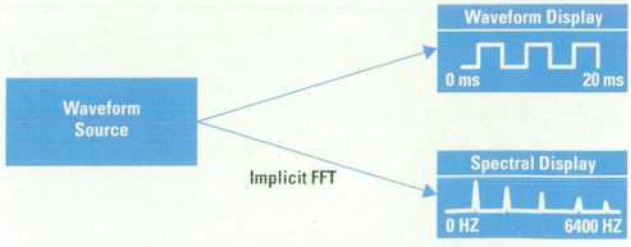

Fig. 4. Automatic transformation of a time-domain waveform (e.g., 256 real values, 0 to 20 ms) to a frequency-domain spectrum (129 complex values, 0 to 6400 Hz).

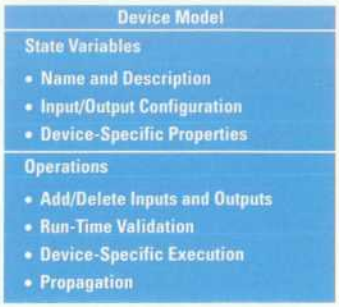

Fig. 5. Attributes of a simple device model.

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

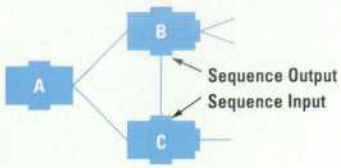

Fig. 6. While B and C both need the data from A, the sequence connection between B and C will cause C to execute after B.

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

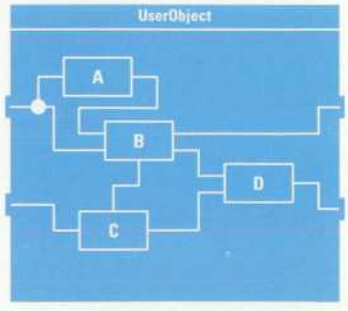

Fig. 7. A UserObject encapsulates a subnetwork of other objects into a single larger object with its own inputs and outputs.

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.

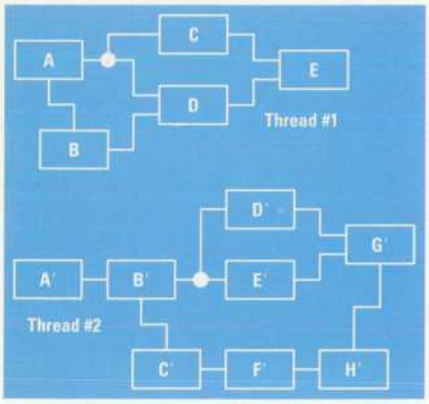

Fig. 8. Any context (e.g., a UserObject) can contain one or more threads, each of which executes independently of all others within that context.

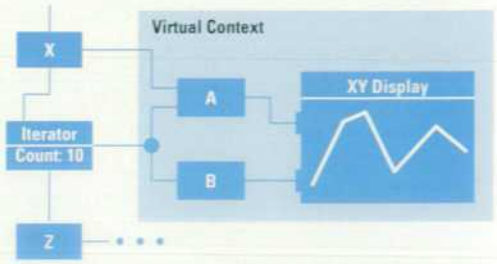

Fig. 9. Objects A and B and the XY display will execute 10 times each at run time as the iterator fires its only data output (right side) 10 times before firing its sequence output (bottom). The data from the output of X is reused for the last 9 of the 10 executions of A (active data rule).

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,

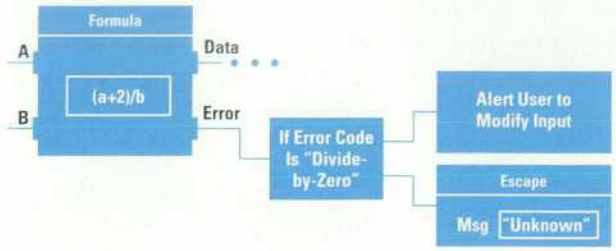

Fig. 10. The special error output will fire in lieu of the data output if any error is encountered while evaluating the formula. The value posted at the error output is the error code number. This allows the user to decide how to handle the situation.

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.