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

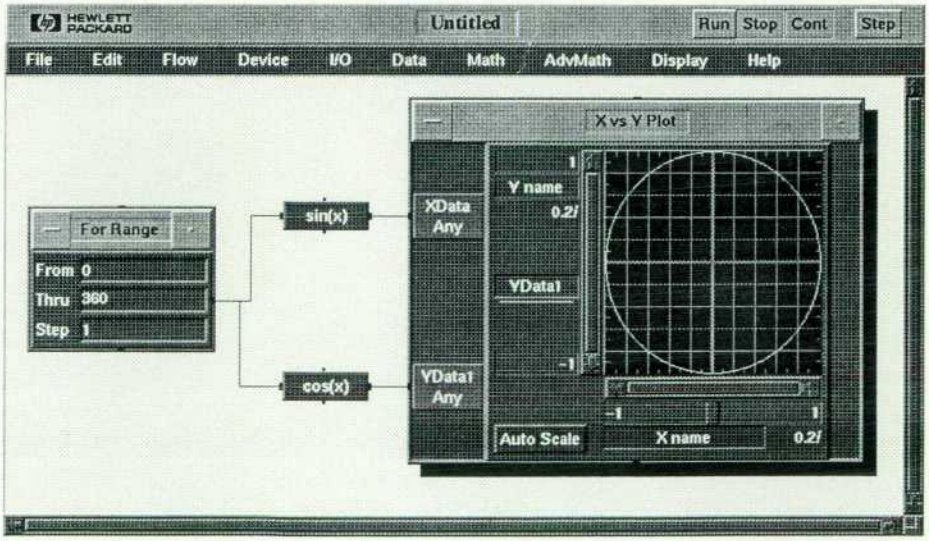

Fig. 1. A simple HP VEE program to draw a circle.

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.

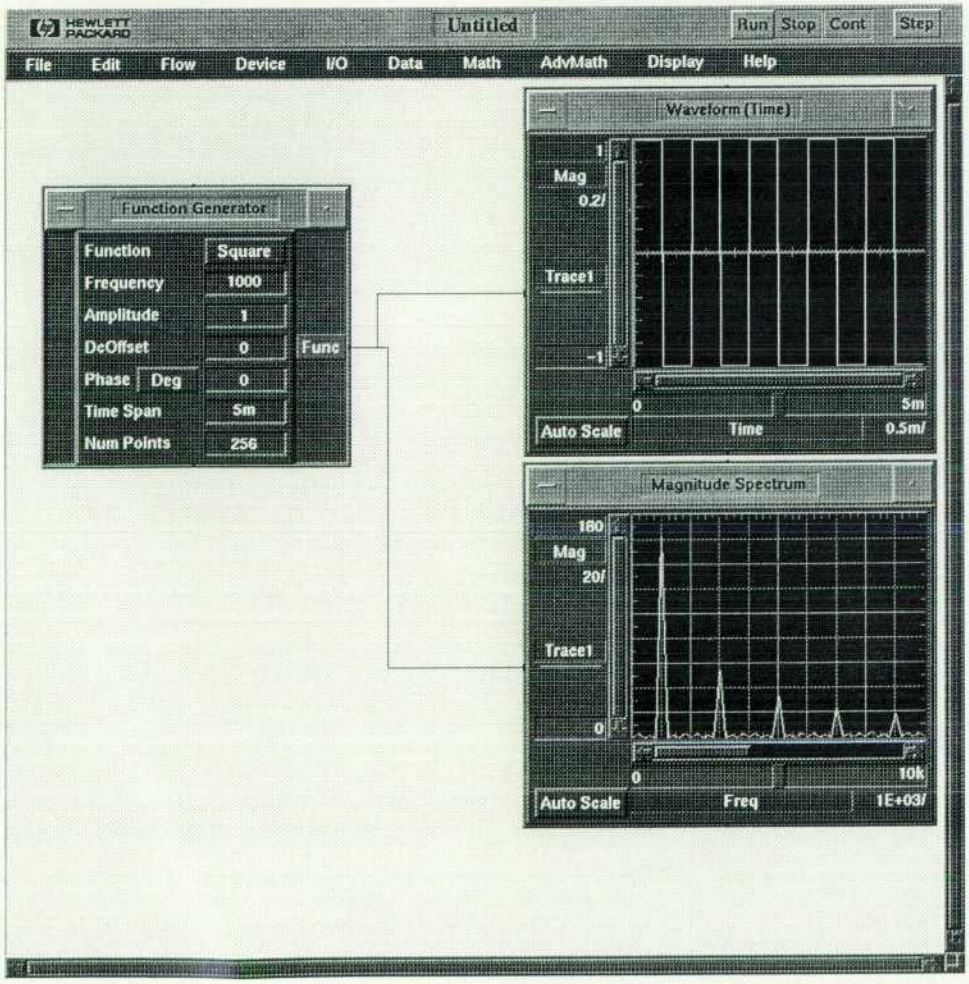

Fig. 2. A waveform displayed in the time and frequency domains.

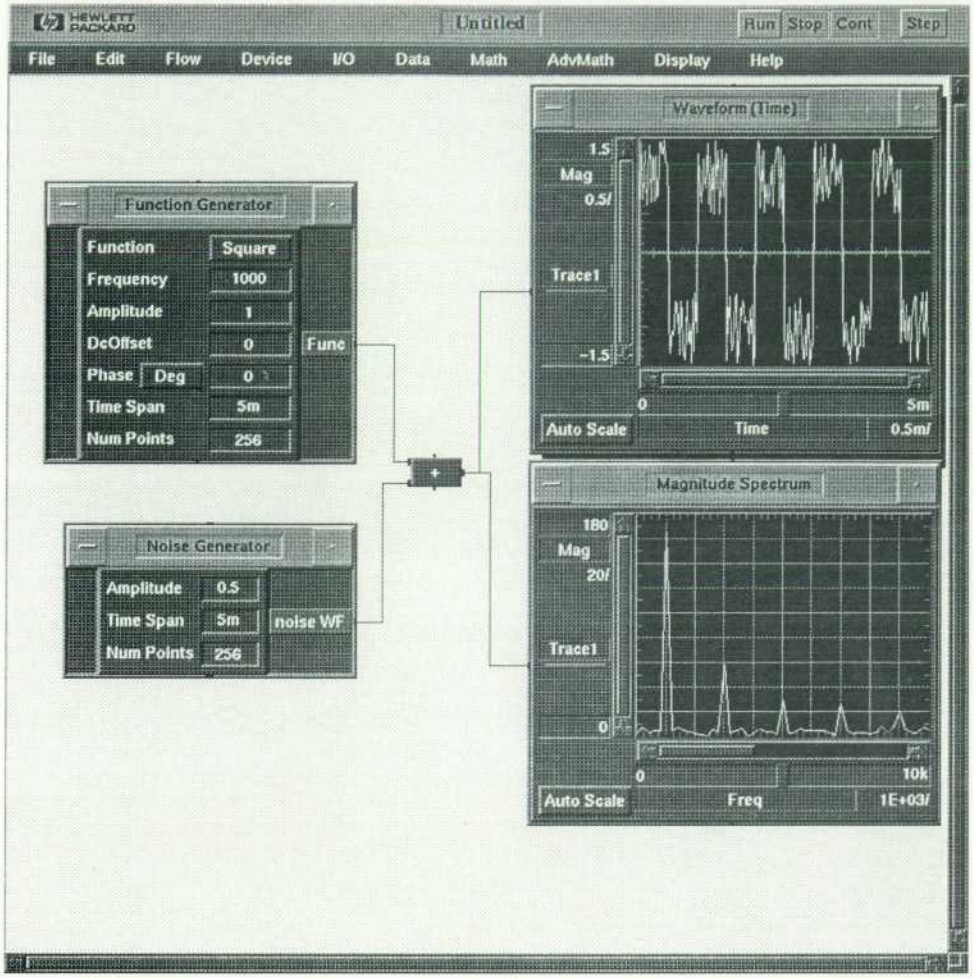

Fig. 3. Noise added to a waveform in the time and frequency domains.

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

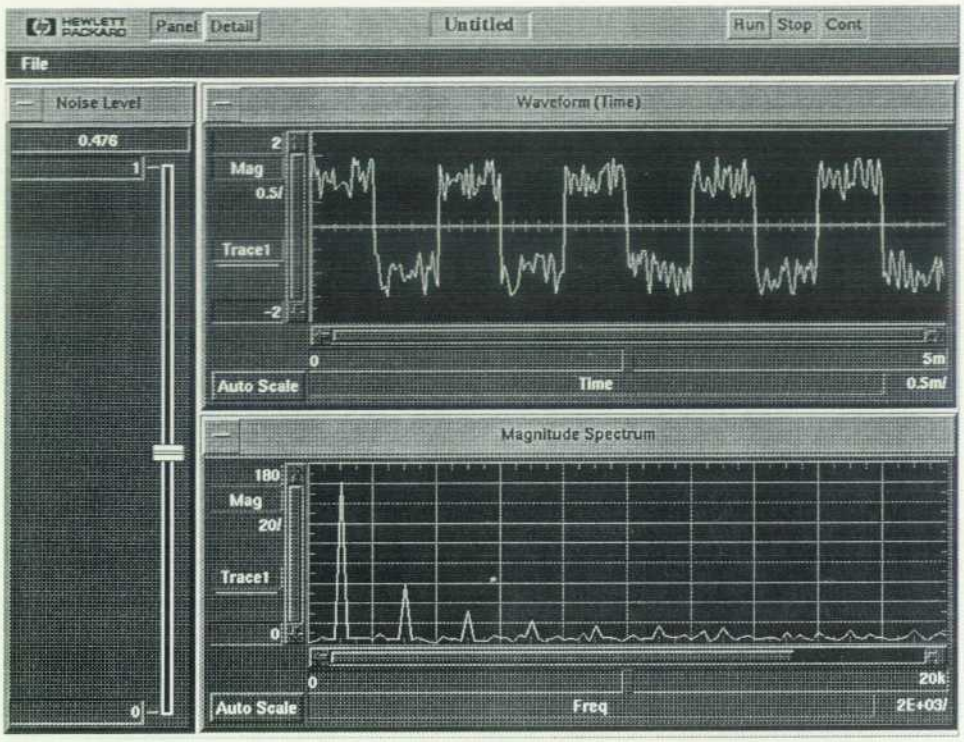

Fig. 4. User panel for waveform plus noise application.

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.