Virtual Instruments: Object-Oriented Program Synthesis

K.S. Bhaskar, J.K. Peckol
John Fluke Mfg. Co., Inc.
J.L. Beug, California
Polytechnic State University

Abstract

Virtual Instruments[1] is an experimental programming environment for developing electronic test and measurement (T&M) applications. Intended users are test engineers, who are not programmers, but computer literate domain specialists. Unlike traditional programming environments, that provide weak support for a broad range of applications, virtual instruments provides strong support for a specific application. The programming paradigm is bottom-up synthesis of layers of virtual machines — called virtual instruments — using human interface models from the application domain, so that software development occurs without writing code. The object-oriented view of the world has proven a natural fit. Implementation was in Berkeley Smalltalk on a SUN workstation.

Overview

This paper reports on an experimental programming environment to support electronic test and measurement (T&M) applications software development. Although T&M software is a major industry, previous investigations of the field have been motivated more by market research for product development than by technical considerations. Since it has not attracted the attention of computer scientists, there has been little, if any, formal research on how T&M software differs from other kinds of software.

Our own work originated in market studies for product development, but branched off into more formal research. We observed that test engineers describe the writing of T&M software as difficult, but, given suitable test instruments, are able to perform the tests manually. We therefore wanted to investigate programming environments based on the manual testing paradigm. We were interested in the differences between T&M software and other software, and why there had not been significant advances in T&M software tools, analogous to the word processors and spreadsheets of other domains.

Our experimental environment, called Virtual Instruments, provides a human interface modeled on T&M instrument front panels: each *virtual instrument* has a functional *virtual front panel*. Instrument front panels are familiar to the test engineer for immediate control of instrument functionality. In addition to this immediate control, our virtual instruments have the ability to emit code corresponding to their virtual front panel settings. This facilitates programmed control of instrument functionality with the same front panels. By also providing ways to encapsulate information, pass parameters, etc., we eliminate the formal coding step. The same uniform human interface extends to debugging as well.

In the following text, we express our views on electronic test and measurement applications, refer to related prior work, describe our contribution, illustrate the use of virtual instruments with an example, outline the implementation in Smalltalk-80, discuss results achieved, and suggest future research.

Electronic Test and Measurement Applications

A typical T&M program is one to test an electronic circuit by evaluating responses to suitable stimuli. A typical configuration consists of a computer (called a *controller*), and a collection of electronic test and measurement instruments, connected to a bus, such as the industry standard IEEE-488 bus.

In the commercial arena, such applications are typically written in variations of BASIC, usually augmented with instrument control primitives. ATLAS and FORTRAN are commonly used in the aerospace / military world. Programming environments are primitive: for example, screen-oriented editors became popular only a few years ago. Previous attempts to study the market in order to provide better environments had concluded that, despite universal complaints about the time to develop applications software,

[1]The material presented in this paper is included in a pending patent.

customers did not want anything more sophisticated! We believe the reasons are:

- Programs are authored by test engineers, who are subject matter specialists with little formal knowledge of computer science. Even though familiarity exists with word processing and spread sheet software, without additional training, better programming environments are more confusing than helpful. BASIC is easy to learn.
- Programs are tedious, but conceptually simple. While programs tens of thousands of lines long are not uncommon, they do not employ complex data or control structures, manipulate symbolic information, execute recursive algorithms, make complex decisions, etc. Whether this is a cause or a symptom is not entirely certain, but it is an observed fact. The structural simplicity of test programs made our work easier.
- Being interpreted, most BASIC implementations provide an immediacy that suits the empirical, "software development by experimentation", techniques that are typically employed. This is important because the correct stimulus is usually experimentally developed. The correct responses have to be "learned" by applying the stimulus to, and measuring the response of a circuit known to be good.
- Software bugs are less likely to result from logic errors than from applying the wrong stimulus or testing against an incorrect expected response. When a bug is encountered, it is desirable to experimentally determine the correct stimulus / response, correct the program and proceed. In this respect, debugging needs are more akin to artificial intelligence expert systems development than to traditional software development.
- Traditional programming environments usually support top-down programming. For test engineers, bottom-up programming is more intuitive.

We also observed that most attempts at developing programming environments attempted to support a broad range of applications, and the requirement for generality appeared to compromise the level of support. We wondered if, by limiting our scope to T&M applications, we could provide stronger support for software development than that available in the more general environments.

Related Prior Work

Within any field are individuals who have acquired expertise through extensive experience. Although these experts may not be trained in formal design techniques, they solve problems effectively using pragmatic methods. To allow such experts to apply familiar skills to the unfamiliar task of software development, an environment is needed that permits the user to concentrate on the problem solution rather than the implementation language.

Traditional attempts to develop more productive programming environments have focused on application breadth [TEI81] [RIC78] [BAR84]. With such techniques, the range of application domains is diverse. Usually, such flexibility is gained only through the sacrifice of problem solving power in specific domains because tools and aids must be general purpose. We took the opposite approach and investigated a restricted domain. This allowed us to provide stronger support with more specialized tools, aids, and methodologies to allow task experts to solve domain specific problems with familiar tools. Expert system development shells are another example of programming environments with strong support for narrow domains [HAY84].

Our work builds on Tannenbaum's [TAN76] interpretion of a computer as a hierarchy of virtual machines (the microprogramming level to the problem oriented language level). In Tannenbaum's model, the language of each successive virtual machine becomes increasingly more powerful and is the machine language for the next machine. One measure of the power of an instruction is the number of resulting actions at the circuit level [BEL71].

With such an interpretation, a program written in the language of the lowest level virtual machine (the hardware or circuit level) can command any set of actions that the hardware is capable of executing. However, writing programs as sequences of binary bits is cumbersome and error prone. At successively higher levels, programs become more task specific, thus, it becomes increasingly easier to accomplish more specialized tasks. Spreadsheet programs are one example, in that only a minuscule subset of possible programs can be executed, but it is easy to accomplish a specific task that fits the paradigm.

Virtual Instruments

Like others before us, we concluded that programming environments for traditional software development were not suited to test and measurement needs. We wanted to demonstrate the feasibility of an environment to dramatically improve software development times for our specific application domain. We were also interested in the possibility of building a specialized programming environment above the Smalltalk-80 environment. We decided that our requirements were:

- Encapsulation and knowledge hiding. Our beliefs told us that these were key ingredients of any programming environment.
- Minimal syntax. We decided that procedure calls or messages were semantically adequate for our needs (in any case, we did not want to invent one more language), but that we would need keyword parameters. (The Smalltalk-80 message syntax was almost ideal.)
- Interactive execution and debugging of code fragments.
- Bottom-up software development. This fits nicely with the previous requirement.
- Graphical human interface models, familiar to the test engineer.

- **Ease of implementation and experimentation.** Since our objective was research, we wanted to minimize implementation effort, and maximize our ability to experiment with different ideas.

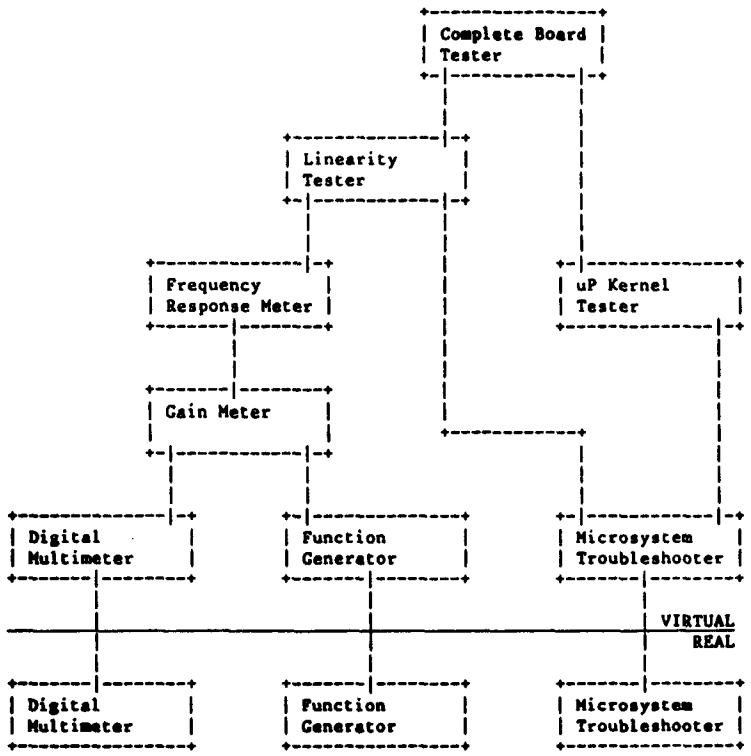
Our contribution lies in the invention of two concepts: *virtual instruments* with *virtual front panels*, and *virtual instrument synthesis*.

Since test engineers are experts at dealing with test instruments, we decided to mimic test instruments on the computer. The test engineer could then manipulate these virtual instruments. Virtual instruments can control real instruments through an interface such as the IEEE-488 interface. A virtual instrument is similar to a device driver, except that it also has a human interface. For example, a virtual instrument disk controller could have a knob to select the track, another to select the cylinder, a display for data, and a Read/Write function selection. This could be operated by setting the cylinder, track and function controls and sending it a "do it" command, via a pop-up menu or a soft key.

Every such instrument is an instance of a corresponding class, and its virtual front panel provides the test engineer with a human interface that s/he is familiar with and can control. The engineer can adjust the "knobs" on the virtual front panel by using a mouse, and command the virtual instrument to "do it": i.e. perform its intended operation and display the results, if any, on "gauges" and other output devices. Virtual front panels are also, therefore, a useful debugging aid for typical test program bugs.

Test program generation can be achieved by sleight of hand. After setting up a virtual instrument to his/her satisfaction, a test engineer can issue a "generate code" command to cause the virtual instrument to emit code corresponding to its current virtual front panel setting. Thus, software generation can, in theory, be accomplished without ever requiring the test engineer to "write code". In fact, there exist many commercial T&M products that support this paradigm in different ways.

However, this basic paradigm is too restrictive and tedious even for test and measurement applications! Our experience with the commercially available systems indicates that they are most useful for demonstration purposes and for the novice. In our virtual instrument synthesis paradigm, instead of using the generated code directly as the test software, we use it to build, or *synthesize* a new virtual



Virtual Instruments in a Board Test Program

Figure 1

instrument class. Once synthesis is complete, instances of the synthesized virtual instrument can, in turn, be used to synthesize higher level virtual instruments. This provides information encapsulation. Furthermore, since external access to a virtual instrument is only through its virtual front panel, hiding is also accomplished.

The virtual instrument synthesis paradigm extends to all levels of abstraction in the test program. For example, consider an amplifier where the gain is controlled by a microprocessor. A complete test program to determine the frequency response would measure the gain of the amplifier through a range of frequencies. By varying the gain and measuring the frequency response at each gain level, the linearity of the amplifier can be tested. By then testing the microprocessor kernel as well (the RAM, ROM, etc.), a complete test program is synthesized. However, at each stage, the human interface paradigm is that of dealing with a test instrument (of course, much detail has been glossed over, but the essence of the technique is simple). Thus, given a digital multimeter and a function generator, the test engineer knows how to measure gain. Given a gain meter and an ability to sweep it over a range of frequencies, s/he knows how to measure frequency response, and so on, to implement a test program for the complete circuit board (see Figure 1). This approach extracts the implied functional modularity that normally would never leave the test engineer's brain, and transfers it to the actual test software. The benefits extend beyond faster software development to software maintainability, portability and code sharing.

An Example

This example illustrates the synthesis of a new virtual instrument called Gainmeter, to measure gain, just as a voltmeter measures voltage. Measuring gain involves applying a stimulus (specified by an amplitude and a frequency), and measuring the response (specified by output voltage). Gain is then computed as

$$20 \log_{10} (\text{output voltage} \div \text{stimulus amplitude})$$

Gain meters do not exist in practice, but making a gain measurement is a meaningful encapsulation of information. In Virtual Instruments, this encapsulation can be instantiated as a virtual instrument with a virtual front panel. This is more meaningful to a test engineer than a traditional gain(amplitude, frequency) function, especially so because this uniform human interface model occurs at every level of abstraction.

The figures in this example are photographs of the screen of a SUN workstation executing Berkeley Smalltalk-80. Smalltalk-80 features such as pop-up menus, mouse button colors, and the distinction between classes and instances are described in [GOL83] and [GOL84].

Figure 2 shows an initial "power-up" screen. In the Generator list window is a list of all virtual instruments classes. The Instrument list window is a list of current virtual machine instances, initially empty.

Typically, the first step is to create required instances of each class. Each class has an inherited method to create

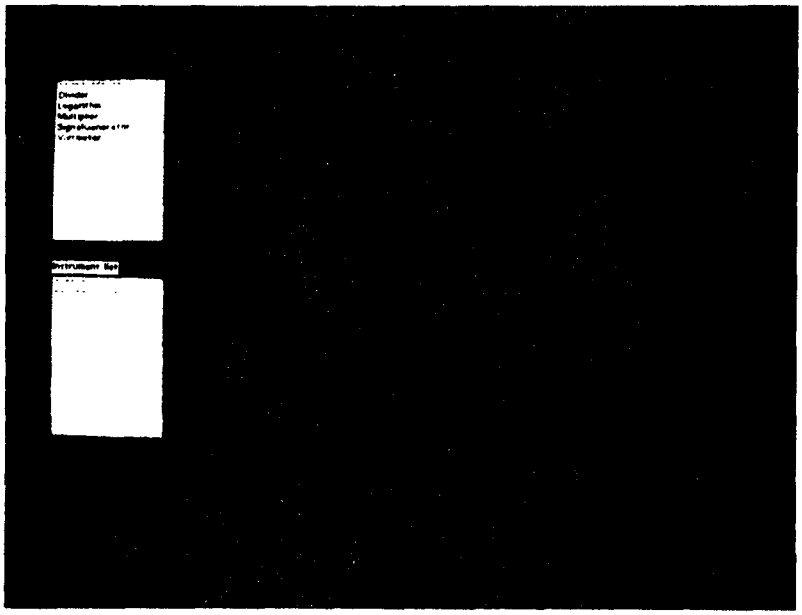


Figure 2

uniquely named instances of itself (each with its own virtual front panel). This feature, like most features, is accessed via a pop-up menu. Figure 3 shows the screen after creating such instances. (Since this was a research project, we decided to explore the limits of the technique. Consequently, even arithmetic operations are performed by virtual instruments: for example, to compute a logarithm, one can set the number and base on the front panel, and "run" the instrument to compute the logarithm.)

Before synthesizing a virtual instrument, a class must be created for it. This skeletal class includes all inherited methods for the overall behavior of an instrument. The subsequent synthesis only requires the definition of the front panel and the generation of steps used to perform its operation, the **runMe** method. Creating a new class is accomplished via a pop-up menu (the user is prompted for a name). This adds the new class to the **Generator** list. The user can then select a pop-up menu option **synthesize**. This creates a **synthesizing Gainmeter** window, as shown in Figure 4. This window is divided vertically into three regions. The three panes of the top region are used to define the front panel of the new virtual instrument class. The middle region displays the code, as it is being generated. The bottom region, in conjunction with the middle region is used to specify arguments for messages.

A front panel definition consists of names of front panel items, a type associated with each name, whether the item is an input, an output, or both, and the type of display (gauge, dial, etc.) to be used for the item. Methods for accessing each item are automatically generated. Figure 5 shows the display after the front panel is specified for class **Gainmeter**. Also in the figure, the value of **amplitude** in the virtual instrument instance **aSignalGenerator15** has been set (this can be done in two ways: the mouse red button can be used to "drag" the pointer, or a new value can be explicitly typed). Experimentation to empirically determine required front panel values can occur at any time, resulting in environment modelessness.

A mouse yellow button menu item, **generate**, inserts the code to invoke **aSignalGenerator15** at run time as a step in the **Gainmeter runMe** method (Figure 6). The generated code selects the signal generator, sets the front panel items, **frequency** and **amplitude**, to their current values, and runs **aSignalGenerator15**. A variable (**q1**) is created to store the value, if any, returned by the signal generator **runMe** method.

Since it is desired to use a **Gainmeter's** own front panel item, **atFreq**, instead of the current value, **1325**, as the **frequency** input to **aSignalGenerator15**, the user then selects **t frequency: 1325** in the middle region, and **self atFreq** in the bottom region. A yellow button menu item is used to effect the change. Similarly, **self inLevel** is associated with **amplitude**. Figure 7 shows the display after making these changes.

This process of generating steps continues. Since the test engineer knows how to make a gain measurement using actual instruments, s/he performs these steps, and the code is generated automatically. Conceptually, the only increase in complexity arises from having to associate previously

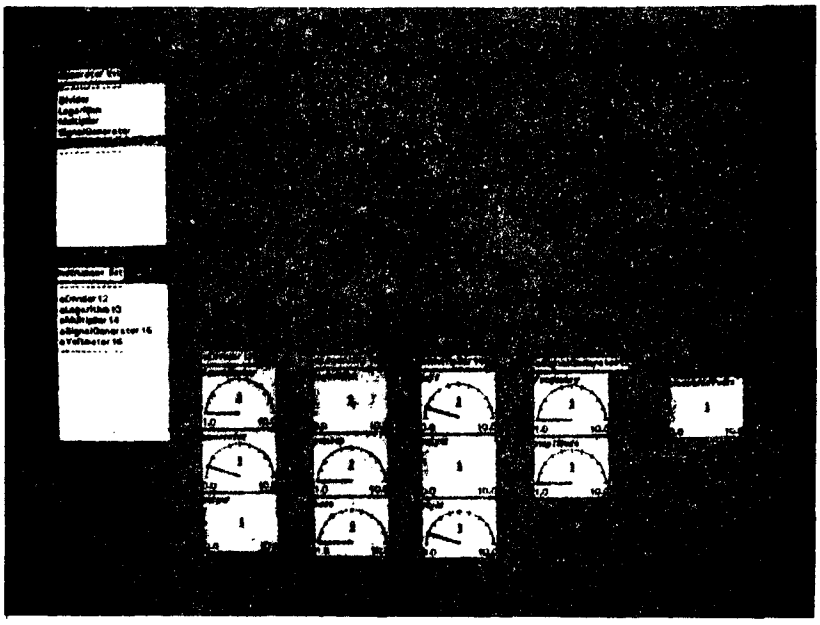


Figure 3

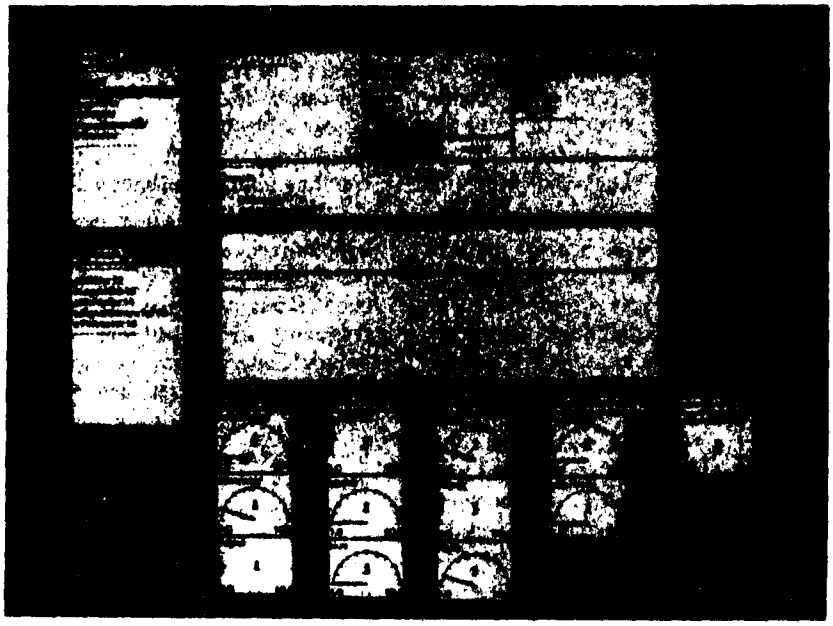


Figure 4

computed results with message arguments, and even this is reasonably intuitive. Figure 8 shows the display after all steps have been generated.

The initial, skeletal code for the GainmeterrunMe method contains a dummy place holder ^ self myValue: self myValue for the value to be returned. A final change to the code is to return q5 (the variable with the value returned by aMultiplier14). The method can then be compiled, and an instance of Gainmeter can be created. This instance, aGainMeter1, can be used just like any other instrument (Figure 9). (An earlier implementation automatically, and transparently, recompiled the method at each stage, but this was too slow, even for experimentation.)

The virtual front panels for the other instruments can now be closed, leaving just the gain meter. This improves execution speed, and makes for a cleaner display. A closed virtual front panel can be re-opened at any time, via a yellow button menu item in the Instrument list window. Opening closed virtual front panels automatically displays the current values. Since values displayed on virtual instrument front panel are the key program variables, and since virtual front panels can also be used to modify these same variables, this extends the uniform human interface model to debugging as well. Debugging is modeless and can happen at any time: there is no need to "compile with the debug flag" or to perform any other ritual.

Design Considerations

The two common design paradigms are top-down and bottom-up (or perhaps, more appropriately, strategy and tactic driven) depending upon whether one moves from the general to the specific or vice versa. A strategy driven methodology is rich in internalized knowledge, knowledge that represents a general understanding of problem and domain. However, such knowledge can be difficult to vocalize. Strategy driven solutions can be characterized as a process of progressive or stepwise refinement [WIR71] [TAU77] (similar to forward chaining) in which a hierarchic decomposition of the problem proceeds through decreasing levels of abstraction. Such methods tend to favor knowledge monotonicity and can be fragile with uncertain knowledge. As a result, some form of truth maintenance or backtracking is required.

In contrast, a tactics driven problem solving approach emphasizes knowledge externalization. Domain experts are usually familiar with detailed information and have an intuitive or heuristic understanding of conceptual interrelationships. Such a methodology encourages incremental development and learning. Solutions tend to be iterative (similar to backward chaining) and robust with uncertain and non-monotonic knowledge.

Virtual Instruments allows a domain expert to work from the pragmatic level to the solution level. It is bottom-up software development by virtual machine construction, applied to the domain of electronic test equipment.

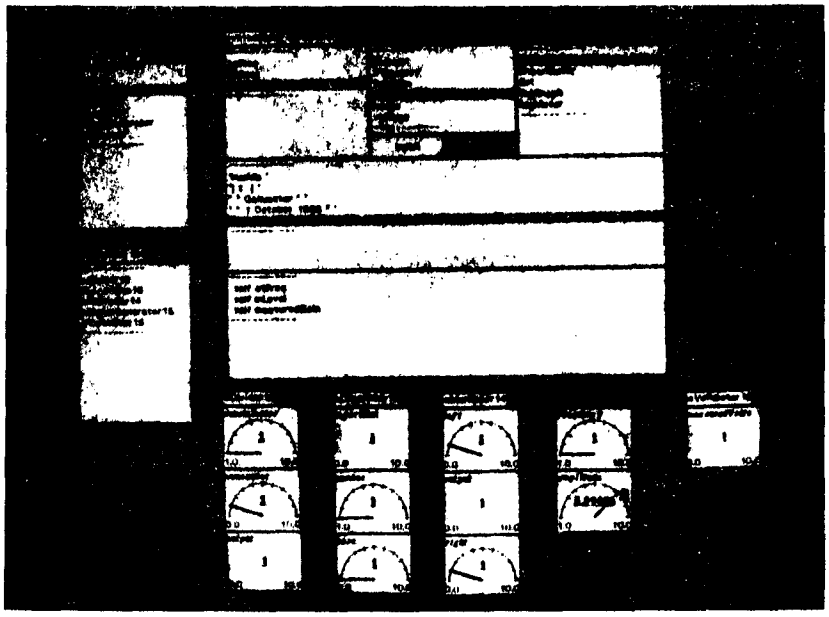


Figure 5

Various programming methodologies or paradigms were considered as possible candidates for implementation. Each is examined briefly.

The pure procedural approach (distinguished from methods using embedded, attached, or hidden procedures) is the most familiar. The basic procedural methodology implies that the problem solver has a broad understanding of the task to be solved. Such a tacit requirement suggests that the method can be particularly fragile at knowledge boundaries [REI80] or under change, thus reducing its power as a tool for incremental development. Further, future modifiers must have an understanding of procedure internals.

A rule based paradigm is more supportive of incremental knowledge and program development than procedural methods. A potentially significant limitation, however, exists in the amount of time spent searching the system knowledge base. The success of the methodology is strongly dependent upon the robustness of the inference engine and conflict set resolution algorithm. As with procedural methods, most present day rule based systems exhibit fragile behavior at knowledge boundaries, performing best in domains characterized by shallow knowledge [PRE85]. Furthermore, the implementor usually requires a domain expert and the assistance of a knowledge engineer (for extracting knowledge from the domain expert).

The technique of using parameter access to trigger procedural invocation has evolved from such languages as Simula [DAH66] and more recently (with the notion of procedural attachment) from some of the frame languages such as KRL [STE86]. Commonly referred to today as access oriented programming, such methods can provide a powerful interface between the user and an underlying program. A demand driven formalism supports incremental knowledge acquisition and program development and can form the basis for "learn-by-watching" software generation. Potentially, program execution speed can also be improved because the control mechanism need not devote time to monitoring for changes in designated variables.

The development environments available for the virtual instruments project were the VAX 11/780 [tm Digital Equipment Corporation] and SUN 2/50, both executing Unix [tm AT&T]. Available language choices for implementation were MAINSAIL [tm XIDAK, Inc.], LISP with Emacs, Icon, and Smalltalk-80 (available only on the SUN). Smalltalk-80 was selected because object oriented techniques in general, and Smalltalk-80 in particular, naturally support many of the criteria established for the development of a virtual instruments programming environment.

- Encapsulation and knowledge hiding are especially well supported through the object discipline. The ability to define an external interface to a virtual instrument independent of the internal implementation is essentially free with object-oriented programming. For example, the definition of an instrument can be improved internally, but higher level instruments need know nothing about it.
- The message syntax of Smalltalk-80 provides a program syntax that is easily understood by the test engineer. We did not have to invent a new language

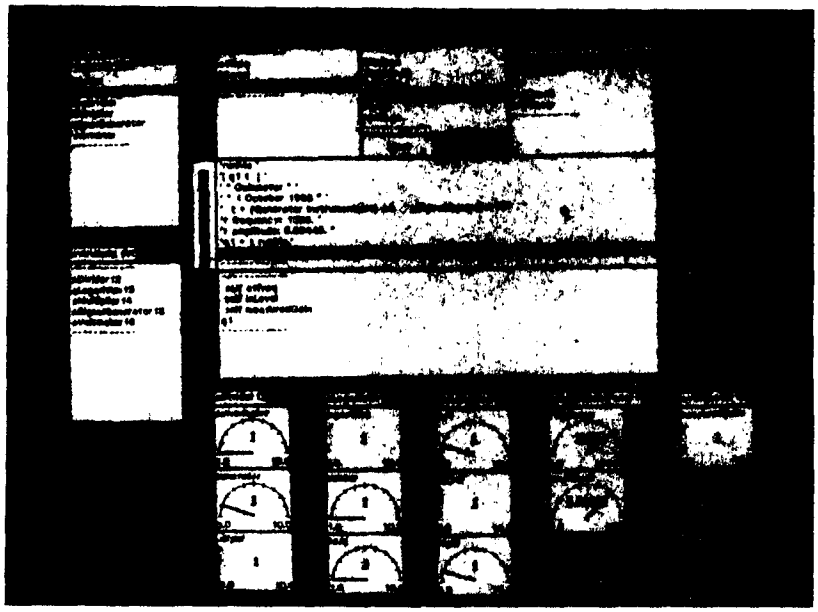


Figure 6

or a new syntax!

- The underlying capability to invoke the Smalltalk-80 compiler dynamically and to create and send messages dynamically makes it easy to interactively execute and debug code fragments. The encapsulation provided by virtual instruments and their front panels provides a convenient and intuitive means for the test engineer to have this capability.
- Bottom-up software development in ordinary environments requires that programmers write "drivers" for lower level modules as they are developed. However, the object management capability and lack of artifacts like linking removes the need for drivers. Virtual instruments can be accessed from higher level virtual instruments, but Smalltalk-80 makes it easy to implement the direct access.
- The classes that are part of the standard Smalltalk-80 distribution provide a rich set of primitives with which to build graphical human interfaces.
- The goal of ease of implementation and experimentation is also naturally satisfied. Abstract superclasses with inheritance provide an ideal mechanism for providing basic behavior for virtual instruments, and making available display devices for virtual front panels (for example, virtual instruments can use new types of display devices as and when these are created).

Implementation in Smalltalk

The software was implemented by Jim Beug. In practice, Smalltalk-80 proved to be a very amenable vehicle for implementing Virtual Instruments. Much of the code was written by studying existing Smalltalk-80 classes, and adapting our software to fit existing classes, or by copying and modifying existing classes to make new classes. For example, SelectionInListView proved widely applicable, and front panel gauges and meters evolved from ClockView.

Each virtual instrument class is a Smalltalk-80 subclass of class VirtualInstrument. Each virtual instrument is an instance of its class. Class methods in class Generator provide the ability to create and delete virtual instruments and classes of virtual instruments (i.e. to administer the Generator list and Instrument list window functions in the example). Instrument synthesis was implemented by instances of Generator, so that, theoretically, synthesis of more than one instrument can occur at any time, although support for this in our experimental implementation is best described as weak. Each instance of Generator manipulates the definition (i.e. the variables and methods) of its virtual instrument class, for example, to create and delete methods and front panel items.

A virtual instrument must respond in three primary ways. First, it must have a means to display and change its settings. This is accomplished via the virtual front panel. Second, there must be a way to use it: to provide a stimulus, measure a response, perform a test, or otherwise

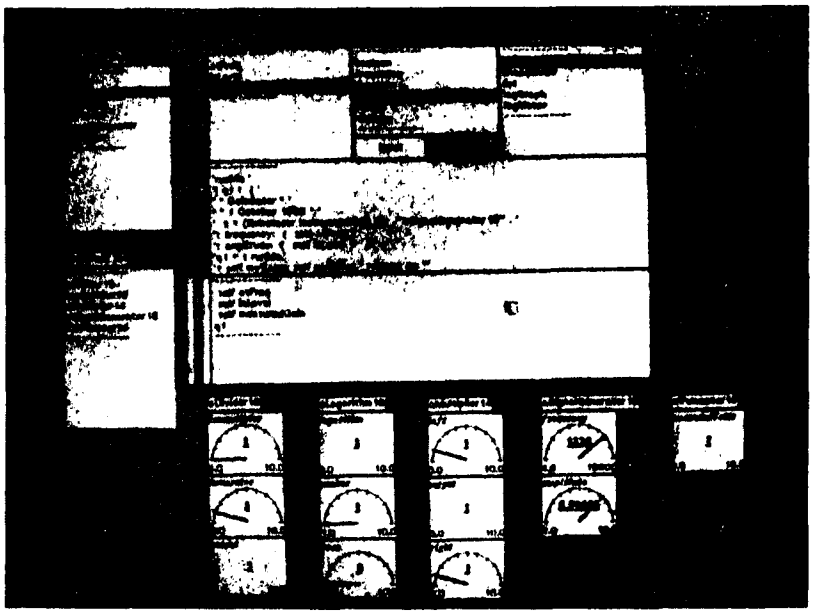


Figure 7

do what it is expected of it. To accomplish this, all virtual instruments respond to a runMe method. Third, a virtual instrument must be able to emit code to set its front panel and then run it. All virtual instruments inherit a generate method to accomplish this.

Each front panel item is an instance of class **Parameter-Blok** [*sic*]. A front panel item is identified by name, by type, whether an input, output, or both, the current value and a view. Each virtual instrument has two methods for each front panel item, one to access and one to modify the item.

Gauges, meters and the like are implemented using the standard Smalltalk Model-View-Controller idiom. Views use virtual instruments as models. Controllers use the red button to modify input parameters, for example, by dragging bars in bar graphs. Yellow button menus are used to control instrument behavior, and are inherited by the instrument from class **VirtualInstrument**. The standard system blue button menu is used for closing, framing, etc.

The inherited generate method emits a line of code to select the virtual instrument, then examines the front panel items, and, for each item, emits a line of code to set it to its current value. Finally, it emits a line of code to send the instrument the runMe message. This code must be inserted into the instrument being synthesized, and appropriate entries made in the third region of the synthesizing window, so that arguments can replace the current front panel values.

Implementation of the software took 10-12 weeks. Smalltalk-80 has a significant learning curve, and an estimated one third of that time was spent becoming proficient. Once this barrier was overcome, however, programming was easy, and, dare we say it, even fun?

Results

The software proved too slow for use in any real application. On a SUN 2/50 with 4M bytes of memory and no local disk, a gain "measurement" took 1-2 seconds. To be completely fair to the Smalltalk-80 implementation, it must be said that the software was written for ease of modification, rather than efficiency. Even with 4M of memory, there was considerable paging (it occasionally ran out of virtual memory!), and not having a local disk probably slowed it down further. Re-implemented for efficiency on faster hardware with a better Smalltalk-80 implementation, it would have been more usable. We believe at least an order of magnitude speed-up to be essential for it to be usable. We decided therefore, not to attempt to drive hardware with this implementation. Thus, in the **Gainmeter** example, the **SignalGenerator** and **Voltmeter** were merely code stubs.

However, the implementation provided sufficient functionality for successful demonstrations of the concept to engineers and managers alike. Most of them believed that virtual instrument synthesis was indeed a viable way to develop test programs and that it had flexibility and extensibility appropriate to the application domain. Apart from

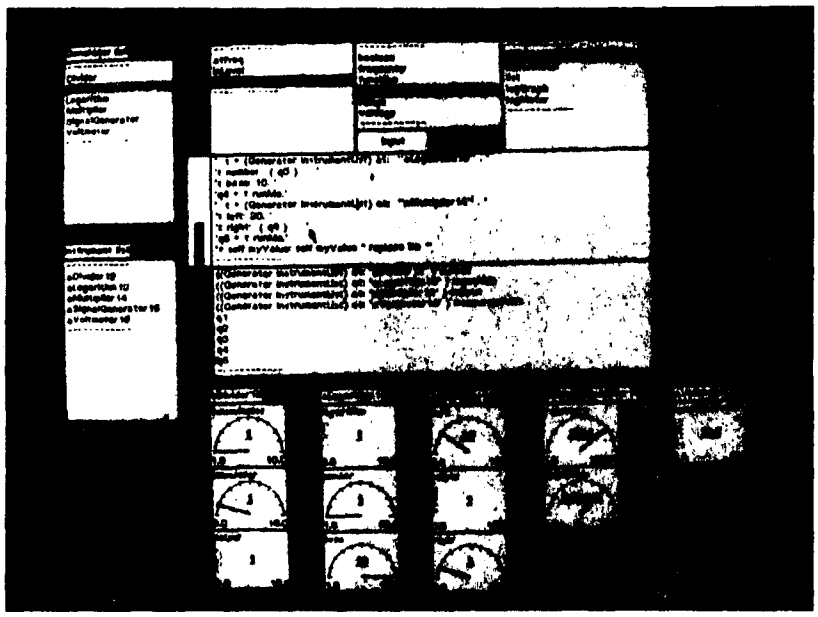


Figure 8

execution speed, the utility of the present implementation is limited by the availability of front panel primitives and base-level virtual instruments, and insufficient robustness. It should be noted that nothing precludes making the present implementation complete. However, since the slow speed of execution is a major obstacle, we felt that our energies would be better directed towards providing full functionality in a future, faster, implementation.

The only fully developed display devices for the front panel are various species of bar graphs and panel meters. A fully functional software package must provide a full complement of such devices. For example, a frequency response meter should display a plot of the gain as a function of frequency. Use of a presently available device, such as a bar graph, would diminish the utility of the frequency response measurement, because the user needs the plot, rather than a series of instantaneous measurements.

A more complete set of base level virtual instruments is required. One aspect of providing a more complete set of base level instruments is to provide support for switching, frequency measurement, etc. The lowest level of virtual instruments (see Figure 1) limits the functionality that can be achieved by a test program: instrument synthesis does not add new stimulus / response capabilities to the system. Hence, the available set of base level instruments limits utility. The other aspect of a more complete set is support for control structures like looping. No control structures are created during virtual instrument synthesis; in fact, few control structures are required in T&M applications. However, some elementary structures are needed, for example, to make a frequency response meter sweep a range of frequencies. When a virtual machine class is created, it can be created by copying another virtual machine class. Therefore, the recommended way to create a frequency response meter class is by cloning a dummy class that implements a loop without a body. Since instrumentation applications require exotic variants of basic control structures (such as loops with geometric steps, instead of arithmetic steps), the available set of dummy classes limits the utility of our implementation.

Owing to bugs in our software and in Berkeley Smalltalk-80, the implementation was less robust than desired. Bugs in software are inevitable, but there were bugs in Berkeley Smalltalk that interfered with our ability to find and fix bugs in our code.

Shortcomings notwithstanding, it is our opinion that the utility of virtual instrument synthesis for T&M software has been demonstrated. More philosophically, it demonstrates the viability of the alternative approach to making computers more usable: that of strong support in a narrow domain, rather than the traditional weaker support in a broader application domain.

Suggestions for Future Work

There are three major directions for future work: extending the domain, enhancing capabilities, and building more efficient implementations. Domain extensions examine how the philosophy can be applied to other areas. Continued research on capabilities involves making the system more

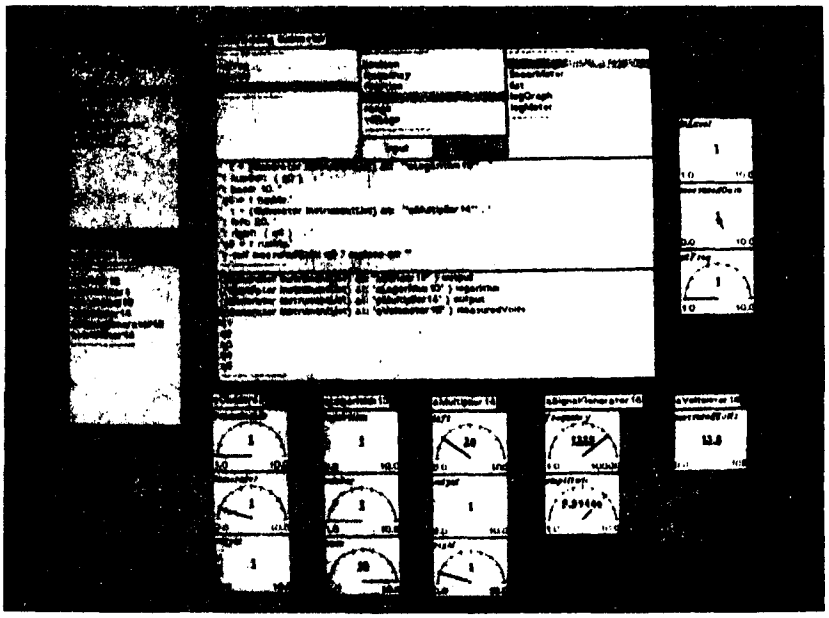


Figure 9

powerful in the domain of electronic test equipment. Work to improve efficiency considers the modifications necessary for production use. The following discussion examines each alternative, with possible domain extensions considered first.

Electronic test equipment provided us with an appropriate domain for formulating and investigating issues in programming philosophy. Our ideas are, however, not intrinsically linked to the domain, and extend to virtual applications generation in several interesting ways.

Areas such as electronic manufacturers and grocery stores can, at first, appear to be unrelated, but, a deeper understanding of the domains often reveals common elements in both problem statement and methods of solution. People in the two example businesses are faced with the problem of inventory control. Such overlap occurs both with the need to ensure adequate supply of raw material (such as electronic components or fresh vegetables) and the need to make certain that the finished product (fabricated systems or bagged groceries) is delivered promptly. Experts in both domains should be able to evolve functional models of their processes using familiar methods and jargon yet ultimately produce similar underlying code.

Virtual generation of applications extends naturally to domains in which there may be no prior art. It should be possible to expand the approach to allow designers to incrementally formulate logical extensions to existing physical machines or to create powerful new ones. Such logical machines could serve as interactive artificial laboratories for experiments that may otherwise be too volatile, hazardous, or expensive.

A strong isomorphism exists between software and hardware. By extending the notion of object oriented programming to hardware, the virtual instruments (or virtual applications) development environment can provide a powerful mechanism for research into dynamic computer architectures. By mapping each virtual applications object onto its own processor in a multiprocessing network it should be possible to configure a loosely coupled MIMD processing system. The system would be coarse grained at the sub-problem level. Such a system could be dynamically reconfigurable and implemented to allow demand driven construction (and dismantling) of constituent virtual machines.

In virtual instruments, the user "teaches" the computer. One extension of the capabilities of virtual instruments is to learn autonomously without teaching. With the present implementation, each new instrument must be explicitly created and configured by the user. By designing daemons to monitor applications activity and to autonomously generate new instruments, it should be possible to create a system whose organization improves with experience. Such a system would entail the autonomous creation of classes and methods and thereby support both generalization and specialization of instruments.

Vertical inheritance, as reflected in generalization and specialization, describes one set of virtual instrument interrelationships. A second kind of inheritance can be seen in horizontal or lateral relationships. Lateral inheritance defines the inheritance of properties between objects usually

considered to be unrelated, i.e. recognition of common properties without full inheritance. By permitting such inheritance, knowledge gained in one domain can be applied to similar problems in a different domain. Thus, for example, a subset of the methods evolved in an adaptive process control system may be applicable to the problem of speech understanding.

By developing specialized virtual instruments, it should be possible to create communities of experts that co-operate to solve problems. Such experts can be created statically or allowed to evolve or learn with experience. Problem solution can proceed in several ways. The process of negotiation through the interchange of messages among instrument objects can lead to an optimized (according to a selected metric) problem solution. Alternatively, a similar exchange of messages can result in shared knowledge. Such synergy of effort can result in the interchange of methods and lead, over time, to the development of specialized collections of knowledge or skill pools.

The present virtual instruments implementation is designed as a research vehicle. When the methodology is extended for casual use, a number of modifications should be incorporated. Currently, the system is "expert friendly", performance in the presence of errors and faults is less than optimal. Extending the existing rules for monitoring and raising faults and including sophisticated fault handlers will contribute to making the human interface more robust.

The execution speed of the current implementation is marginally adequate for limited research. In a practical application environment, such performance will be unacceptable. Several possibilities for such improvement exist. One is to optimize the underlying code for the existing (or a faster) hardware vehicle. Several other possibilities are to execute the method either on a device such as the Berkeley SOAR [UNG84] processor, or the Tektronics 4406, or to implement directly in silicon.

Finally, research should be conducted into incorporating various other programming paradigms into the virtual instruments environment. Among these are included both the rule based and access oriented methods.

Bibliography

[BAR84] Barstow, David R., Shrobe, Howard E., Sandewall, Erik, *Interactive Programming Environments* McGraw-Hill Book Company, 1984.

[BEL71] Bell, C. Gordon and Newell, Allen, *Computer Structures: Readings and Examples* McGraw-Hill Book Company, 1971.

[DAH66] Dahl, O.J., and Nygaard, K. SIMULA - An ALGOL-Based Simulation Language, *Communications of the ACM*, Vol. 9, pp. 671-678, 1966.

[GOL83] Goldberg, Adele and Robson, David, *Smalltalk-80 The Language and its Implementation* Addison-Wesley Publishing Company, 1983.

[GOL84] Goldberg, Adele, *Smalltalk-80 The Interactive Programming Environment* Addison-Wesley Publishing Company, 1984.

[HAY84] Hayes-Roth, Frederick, Waterman, Donald A., and Lenat, Douglas B., *Building Expert Systems* Addison-Wesley Publishing Company, 1984.

[PRE85] Prerau, David S., Selection of an Appropriate Domain for an Expert System, *AI Magazine*, Vol. vii, No. 2, pp. 26-30, Summer 1985.

[REI80] Reiter, R., A Logic for Default Reasoning, *Artificial Intelligence*, Vol. 13, pp. 81-132, 1980.

[RIC78] Rich, Charles and Shrobe, Howard E., Initial Report on a LISP Programmer's Apprentice, *IEEE Transactions on Software Engineering*, Vol. SE-4, No. 6, pp. 456-467, November 1978.

[STE86] Stefik, Mark J., Bobrow, Daniel G., and Kahn, Kenneth M., Integrating Access Oriented Programming into a Multiparadigm Environment, *IEEE Software*, pp. 10-18, January 1986.

[TAN76] Tanenbaum, Andrew S., *Structured Computer Organization* Prentice-Hall, Englewood Cliffs, New Jersey, 1976.

[TAU77] Tausworth, Robert C., *Standardized Development of Computer Software* Prentice-Hall, Inc., Rnglewood Cliffs, New Jersey, 1977.

[TEI81] Teitelman, Warren and Masinter, Larry, The Interlisp Programming Environment, *IEEE Computer*, Vol. 14, No. 4, pp. 25-34, April 1981.

[UNG84] Ungar, David; Blau, Ricki; Foley, Peter; Samples, Dain; and Patterson, David, Architecture of SOAR: Smalltalk on a RISC, *11th Annual Symposium on Computer Architecture*, June 4-7, 1984, Ann Arbor, Michigan.

[WIR71] Wirth, N. Program Development by Stepwise Refinement, *Communications of the ACM*, Vol. 14, No. 4, pp. 221-227, 1971.