Scaling Up Visual Programming Languages

Margaret M. Burnett, Marla J. Baker, Carisa Bohus, Paul Carlson, Sherry Yang, and Pieter van Zee
Oregon State University

The directness, immediacy, and simplicity of visual programming languages are appealing. The question is, can VPLs be effectively applied to large-scale programming problems while retaining these characteristics?

Visual programming languages let the programmer sketch, point at, or demonstrate data relationships or transformations, rather than translate them into sequences of commands, pointers, and abstract symbols. These simplifications seem to promise to make programming easier, more reliable, and more accessible. However, making visual programming languages suitable for solving large programming problems often seems to require the very complexities VPLs try to remove or simplify. This is called the *scaling-up problem.*

Productive research that does not compromise the attractive qualities of VPLs depends on an understanding of the problem, its components, and their interrelationships. In this article we profile the scaling-up problem through nine of its subproblems. Some present formidable obstacles, some have been virtually solved, but others have scarcely been recognized. They all share two characteristics: (1) their solutions are important to solving the scaling-up problem, and (2) their attempted solutions outside the context of the scaling-up problem will have little impact in moving VPLs closer to the goal of scaling up.

WHAT ARE VISUAL PROGRAMMING LANGUAGES?

Shu defines VPLs as languages that use "some visual representations (in addition to or in place of words and numbers) to accomplish what would otherwise have to be written in a traditional one-dimensional programming language."[1] As Shu also states, "to be considered a visual programming language, *the language itself* must employ some meaningful . . . visual expressions as a means of programming." (In keeping with this definition, visual environments supporting *textual* programming languages are outside the scope of this article.)

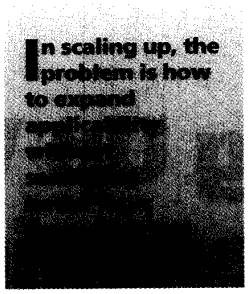
VPL characteristics

The goals of VPL designers are to improve the programmer's ability to express program logic and to understand how the program works. These goals, however, cannot be realized just by switching from text to pictures. Instead, newer VPLs employ visual techniques to achieve one or more of the following characteristics:

1. *Fewer concepts required to program.* For example, in many VPLs the programmer does not deal with pointers, storage allocation, declarations, scope, or variables.
2. *Concrete programming process.* In many VPLs, the programmer can see, explore, and change specific data values or even sample executions.
3. *Explicit depiction of relationships.* Constraint and dataflow diagrams are example techniques.
4. *Immediate visual feedback.* After a program edit, in many VPLs, imme-

0018-9162/95/$4.00 @ 1995 IEEE

diate display of updated results helps the programmer find errors sooner.



Opportunities and obstacles in scaling up are especially great in the class of VPLs with a high degree of *liveness*, Tanimoto's term for the amount and immediacy of visual feedback provided to the programmer.[2] We call this class *responsive VPLs*. In responsive VPLs, a programmer action—such as an edit to the program or data—immediately executes the change and redisplays the affected displayed values. There is no separate compilation step or explicit "run" command by the programmer. In this article, we emphasize the issues faced by responsive VPLs.

The scaling-up problem

In scaling up, the problem is how to expand applicability without sacrificing the goals of better logic expression and understanding. From a size standpoint, scaling up refers to the programmer's ability to apply VPLs in larger programs. Such programs range from those requiring several days' work by a single programmer to programs requiring months of work, large programming teams, and large data structures. From a problem-domain standpoint, scaling up refers to suitability for many kinds of problems. These range from visual application domains—such as user-interface design or scientific visualization—to general-purpose programming in such diverse areas as financial planning, simulations, and real-time applications with explicit timing requirements.

To illustrate the scaling-up problem, we discuss nine major subproblems and describe emerging solutions from existing VPL systems. First, we examine representation issues, including static representation, screen real estate, and documentation. Next, we examine programming language issues—procedural abstraction, interactive visual data abstraction, type checking, persistence, and efficiency. Finally, we look at issues beyond the coding process.

REPRESENTATION ISSUES

In VPLs, visual representations offer the flexibility to communicate more information than text alone. Despite this advantage, numerous problems arise in devising effective visual representations for programming. Three of the most widely-recognized representation problems are static representation, effective use of screen real estate, and documentation.

Static representation

The appearance of a visual program "at rest," such as on a screen snapshot, is called its static representation. Static representation is the same as syntax in any language whose syntax is static and two-dimensional (or less). Examples of such syntaxes include text, dataflow diagrams, and state-transition diagrams. For VPLs without this kind of syntax, however, static representations are difficult to devise. For these languages, lack of a complete static representation is a barrier to program review, analysis, and explanation.

Problematic VPLs in this area include those with dynamic syntax, such as programming-by-demonstration languages. For example, to create a program that left-aligns any two rectangles, the programmer might insert a line along the left edge of a sample rectangle, drag a second rectangle against the line, and finally delete the line. In this example, the programmer's dynamic manipulations are the syntactic tokens for the program's operations.

Even when language syntax is entirely static, some VPL attributes can be difficult to represent statically. VPLs that require nontextual references to data—by pointing at unlabeled data or icons, for example—are often difficult to represent in a way that is static, precise, and two-dimensional. Values that involve a time dimension or more than two spatial dimensions, such as animations and multidimensional data structures, are also difficult to represent statically.

For a VPL's static representation to be useful, it must be consistent with the VPL's goals. For example, although we can imagine the programmer painstakingly creating program screen dumps in various states and manually pasting them together to obtain a complete printout, such a cumbersome approach is hardly consistent with the goal of making programming easier. For the same reason, translation to a standard textual language such as C for static representation purposes is unsuitable for VPLs that are intended to allow programming without reverting to traditional languages.

One measure of a static representation's usefulness is editability. To edit a static representation (if editing is supported), the programmer must use two syntaxes—dynamic and static—which can increase programming difficulty. One solution is to devise a static syntax with strong similarities to the dynamic syntax. The sidebar "Static representation in Chimera" presents such an approach.

Another measure of a static representation's usefulness is the capability to hide excessive visual details with levels of abstraction. For abstraction over time, the goal is to avoid lengthy screen sequences that differ from one to the next by a single pixel. The sidebar shows how Chimera employs context-sensitive evaluation of multiple actions to hide excessive spatial and temporal details.

Effective use of screen real estate

Visual interaction is the VPL's primary means of communication, but adequate, timely display of screen information is difficult for large programs. The problem is limited screen size and effective use of the available space. Techniques are needed to organize and access large, diverse quantities of program information. The capabilities of the menus, icons, and scrollable windows commonly found in user interfaces are not powerful enough to fill this need.

Although most work relating to screen real estate has been done independent of the scaling-up problem, screen real estate is closely tied to other scaling-up subproblems. For example, screen real estate strategies can have a strong impact on overall system efficiency. If display and navigation techniques are inefficient, the impact on a responsive

VPL could require a retreat to a lower level of liveness.

Screen real estate also has close ties to static representation. For example, if improved static representation leads to effective VPL printing capabilities, the static printed page could supplement screen real estate. But the shortage of screen real estate makes it more difficult to devise a static representation usable for on-screen display. The Chimera system integrates work on VPL static representation with work on effective use of screen real estate. It incorporates rule-based techniques to ascertain the elements of current interest, eliminating extraneous details. The programmer has navigation options, such as exploding one panel into a strip of panels to show more details and coalescing a strip into a panel to show fewer details.

Some relevant work has not yet been applied to VPL screen real estate. Data display and navigation research from the human/computer interaction and graphics communities might be adaptable to VPLs (see the sidebar "Displaying more in less space"). Also, VPL research related to graphical reasoning might eventually offer navigation power comparable to that now provided by search capabilities for word patterns in textual language editors. In addition, VPL documentation research shows that, because many VPLs' environments intimately understand language syntax and semantics, semantic information can control the way program information is mapped to screen space.

Static representation in Chimera

Chimera[1] is a system for creating and editing user interfaces, graphics, and text, whereby macros are programmed by demonstration. The programmer demonstrates the desired operations, then the system statically represents the operations using a comic-strip metaphor. Figure A shows a statically represented sequence that results in one box pointing (with an arrow) to another. The programmer selects part of the sequence, identifies its parameters, and instructs the system to convert the selected sequence into a reusable macro.

Chimera coalesces physical operations into logical ones. In Figure A, the name of each logical operation appears, with the number of physical operations it represents, above each panel. A box is created, and its attributes are set in four logical operations (panels two through five); the line, arrowhead, and second box are created in the next three logical operations; and the arrow is aligned between the two boxes in the last three logical operations. (The programmer can interactively expand any panel into a more detailed sequence.)

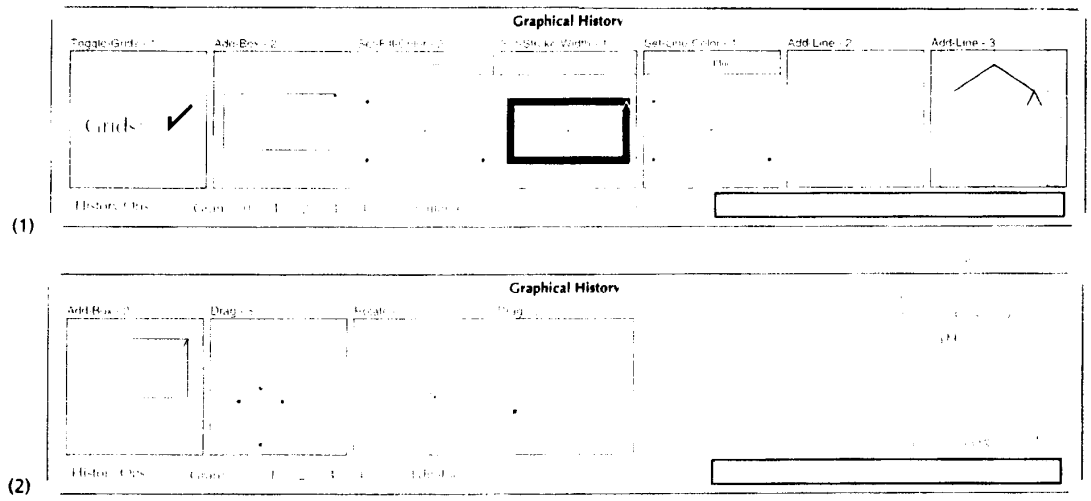
A given panel shows only those objects that participate in the logical operation, plus some landmark objects for screen context (landmark objects have a faded appearance). Panels can be edited using the same "by-demonstration" style with which they were created.

The example shown demonstrates Chimera's key strategies for static representation, which are to

- display logical rather than physical operations;
- permit interactive expansion of high-level panels into low-level ones and coalescence of low-level panels into high-level ones;
- show, in each panel, only those objects that participate in the panel's logical operations (with landmark objects also shown for context);
- render objects in a style according to their role in the logical operation; and
- let the programmer edit the static representation.

Reference

1. D. Kurlander, "Chimera: Example-Based Graphical Editing," in *Watch What I Do—Programming by Demonstration*, A. Cyper, ed., MIT Press, Cambridge, Mass, 1993, pp. 271-290.



(1)

(2)

Figure A. Static representation of Chimera operations. (Reprinted with permission from *Watch What I Do—Programming by Demonstration*, A. Cyper, ed., MIT Press, Cambridge, Mass. © 1993.)

Documentation

Code-level documentation lets the programmer supply additional information that cannot be conveyed in the source code. Although VPL documentation is not well developed, the work to date demonstrates that it is more an opportunity than a problem because of its potential—through VPLs' visual and dynamic characteristics—to move beyond textual-language documentation.

DYNAMIC TEXTUAL DOCUMENTATION. Many VPLs support textual documentation, functionally equivalent to the in-line comments found in textual languages, by means of textual comments that can be placed in the program. For example, the object-oriented dataflow language Prograph[3] allows the programmer to annotate the dataflow graphs.

The documentation takes space, which exacerbates the problem of screen real estate. Prograph addresses this conflict by allowing the user to control when documentation is displayed. Fabrik[4] extends the Prograph approach with selectable levels of displayed documentation detail and by displaying the documentation of whatever is under the mouse pointer in a reserved documentation box, thus conserving space while providing easily accessible, on-demand information. The drag-and-drop mechanism in

Displaying more in less space: Context versus detail

Many visual representations of programs take a great deal of space. Examples include diagrams representing large procedures or data structures. There are two traditional solutions: make the drawing smaller and look at the whole, or partition the drawing into pieces and look at each piece separately. While a smaller drawing preserves the context, the reduction may make the details unrecognizable. Partitioning a drawing yields the opposite effect: details at the expense of context. The challenge is to develop display techniques that provide both detail and context, in conjunction with navigation techniques that let the programmer access information without being distracted by the task of navigating.

Recent approaches developed for data display allow viewing and navigating through detailed information, using contextual cues. These approaches may be extensible for program display. Cone trees,[1] perspective walls,[2] and fisheye views[3] demonstrate some of the possibilities in Figure B. Cone trees employ 3D techniques to fit more information from 2D hierarchies on the screen. A perspective wall b ıds a 2D linear structure that would not completely fit or he screen with traditional methods, using 3D depth cue In fisheye views, information is selected for display base on its relevance. A numeric threshold function determ ıes whether an item will be displayed based on how clo: to the area of interest the piece of information is, and its gl bal (contextual) importance.

References

1. G. Robertson, J. Mackinlay, and S. Card, "Cone Trees: ıni- mated 3D Visualizations of Hierarchical Information," oc. *CHI 91 Conf.*, ACM, New York, pp. 189-194.
2. J. Mackinlay, G. Robertson, and S. Card, "The Perspective 'all: Detail and Context Smoothly Integrated," *Proc. CHI 91 C nf.*, ACM, New York, pp. 173- 179.
3. G. Furnas, "Generalized Fisheye Views," *Proc. CHI 86 C nf.*, ACM, New York, pp. 16-23.

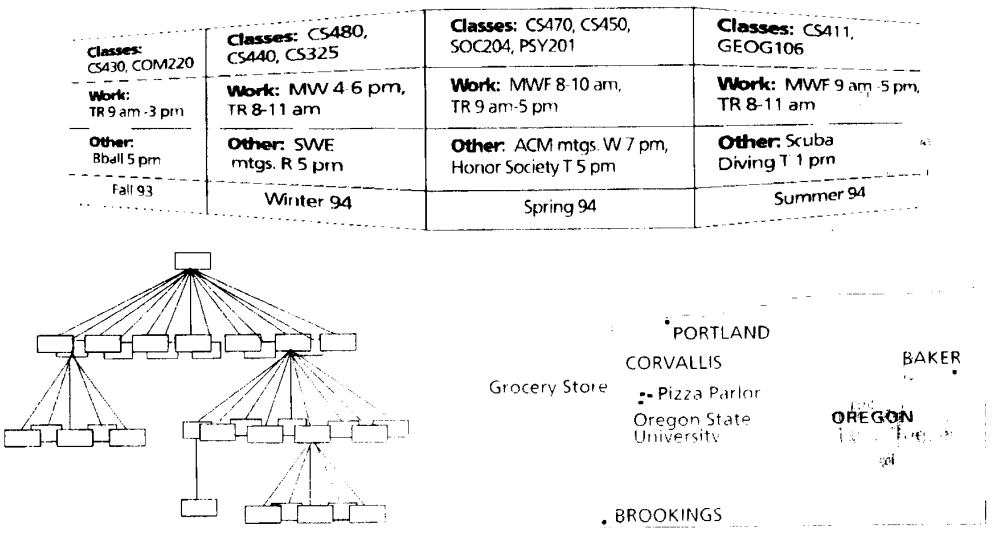


Figure B. Top: A perspective wall. Bottom left: Cone trees display 2D trees using 3D cones. Bottom right: A fishey view.

"VPL"[5] provides similar documentation support. In this approach, objects can be selected and dragged to other windows with different capabilities, one of which displays the documentation for dropped-in objects.

STATIC GRAPHICS AND AD HOC ANIMATIONS. VPLs can support documentation in media other than text. For example, Forms/3[6] supports graphical documentation. The programmer can incorporate color, boxes and lines, sketches of data structures, and even programmed combinations of these items predicated on the program's data values.

Because the primary reason for program documentation is to describe program operation, inaccurate documentation is worse than none at all. Inaccuracies—a pervasive problem, historically—tend to arise in any documentation that is independent of the program itself.

Accurate documentation is an area in which responsive VPLs can excel. Whenever the programmer enters or changes a responsive VPL program, the VPL computes new answers, often displaying a rudimentary animation of the computation and all intermediate values. To learn how a particular program segment works, a programmer can provide values and watch an instant animation. This capability is exploited further in VPLs—such as Forms/3 and Garnet's C32[7]—that support preexisting sample values, enabling the programmer to see an ad hoc animation without having to think of sample values to start the animation.

The ability to watch any portion of the program on demand fulfills the documentation goals of helping the programmer understand the program, while guaranteeing consistency between this information and the actual workings of the code. Thus, because programs with test values can provide responsive documentation and because documentation can include programming, the separation between source code and documentation begins to disappear.

PROGRAMMING LANGUAGE ISSUES

The general programming-language community has long had successful approaches for handling data abstraction, types, and other issues. Simple visual implementation of traditional approaches, however, is seldom suitable for VPLs. Adding functionality is not enough; the solutions must also be consistent with characteristics that are used to achieve VPL design goals such as simplicity, concreteness, explicitness, or responsiveness.

Procedural abstraction

An important advance in the early days of programming was procedural abstraction, the ability to create procedures or subroutines that encapsulate subtask details. Procedural abstraction is such an important building block in today's programming projects that it's hard to imagine programming without it. Unlike the other issues discussed, procedural abstraction has been solved in today's VPLs. The most important attribute in the solutions is consistency with other programming in the same VPL. Representative solutions include

- allowing the programmer to select and iconify a section of a dataflow graph, which adds a node (representing the subgraph) to a library of function nodes;
- using a form as a grouping mechanism for calculations; and
- recording and generalizing a sequence of manipulations (as in the Chimera sidebar).

Studying earlier VPL procedural abstraction solutions is instructive in understanding how subproblem solutions, when formulated in isolation, can actually detract from a VPL's scalability. Some early solutions required procedures to be created in traditional programming languages such as C. These solutions did not contribute to the scalability of *visual* programming languages because the programmer had to program in a traditional *textual* language to achieve scalability. This cast VPLs in the role of supporting textual programming languages, rather than being scalable programming languages themselves. Other early solutions restricted the use of procedures to prepackaged library functions—an approach that was at odds with the generality needed for scalability. Thus, while these early solutions provided procedural abstraction, they did so at the expense of attributes needed for scaling up VPLs.

Interactive visual data abstraction

The encapsulation of details that follows from procedural abstraction is equally important in the realm of data types. Data abstraction encapsulates user-defined abstract data types by allowing access only through programmer-defined operations.

For VPLs, data abstraction raises issues such as maintaining concreteness, visibility, and interactivity in the presence of abstract types and information hiding. Such issues have prevented many VPLs from supporting data abstraction, particularly in responsive VPLs. Recently, however, approaches have been developed that address the issues.

INTERACTIVE + VISUAL + DATA ABSTRACTION. By definition, a VPL that supports data abstraction supports a visual process to define a new data type, and it also results in a visual program. A responsive VPL adds interactivity to the process. We use the term *interactive visual data abstraction* to emphasize these aspects. The earliest attempts at data abstraction in VPLs did not actually support visual programming; they either did not support a visual process or did not result in visual programs. Visual code generators, for example, produced textual—not visual—programs. The opposite approach, textual creation of data types that could later be used visually, did not support visual programming because programming was done textually. These approaches could not support responsiveness, since the programming process was separated from the code-evaluation process.

Many applications today are graphical and interactive. So that a VPL can provide these characteristics in a manner consistent with data abstraction, support is needed for user-defined visual appearance and user-defined interactive behavior. ObjectWorld[8] and Forms/3[6] are two VPLs that incorporate a type's appearance into its definition. Forms/3 also supports specifications about a type's behavior under user interaction (see the sidebar "Interactive visual data abstraction").

Forms/3 is a VPL based on the spreadsheet-like notion of cells with formulas. As soon as a formula is provided for a cell on the screen, the system calculates and displays its value, providing immediate feedback. The programmer groups related cells on forms, which provides the basic mechanism for abstraction.

Suppose the programmer wishes to create an abstract data type ImageMover, a new event-sensitive type that an application end-user can move around a window. First, the programmer clicks on a palette to instantiate a new form and names the new form ImageMover. Next, the programmer places cells on the form that specify the components of the new data type. As the section marked "composition" of Figure C shows, an ImageMover will be composed of a width, height, shape (the person eating ice cream), and an event receptor. An event receptor is a primitive type that detects interactive events.

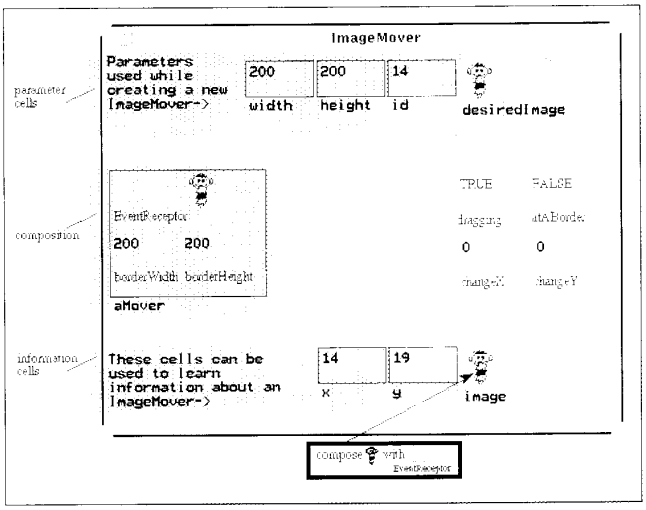


Figure C. This form defines the data type ImageMover. Since it includes an event receptor, it is an interactive type. Information about events is available in the cells on the same copy of the EventReceptor form that is referenced by cell EventReceptor, shown in Figure D.

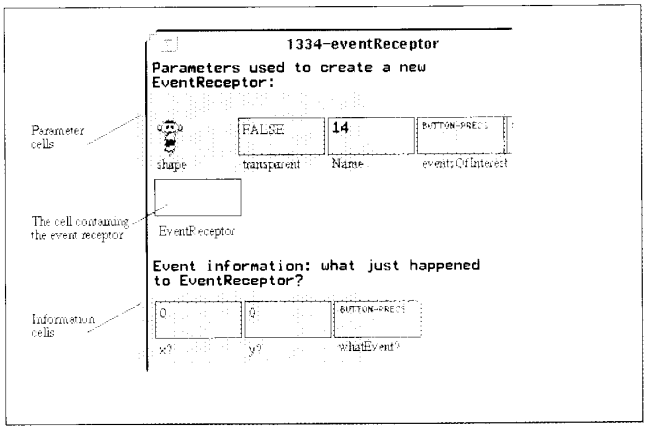


Figure D. A portion of the form defining the event receptor built-in data type.

Each cell has a formula. The formula (not shown) for EventReceptor is simply a reference to cell EventReceptor on the built-in form shown in Figure D. The visual appearance of all ImageMovers is calculated using the formula in the cell image (bottom of Figure C). The programmer defines some formulas as (sample) data values, such as "200" for cell width, to obtain concrete feedback during development.

Logical information-hiding is accomplished through physical information-hiding. For example, the cells with dotted borders on the middle right of Figure C have been marked "hidden" by the programmer. All cells that define the composition of a type are automatically marked hidden by the system. Hidden cells are usually visible only when the programmer is defining a formula for a cell on the same form. Accessibility of a hidden cell is determined by visibility: If it isn't visible, it cannot be referenced in a formula.

Figure D shows the instance of the built-in data type EventReceptor that was referenced in one of the components of the ImageMover. In Forms/3, the cell formula actually defines a sequence of values along a logical time dimension rather than an atomic value, and this provides the foundation of the approach to events. For example, the displayed value for cell whatEvent? reflects the most recent event in the sequence. Any cell whose formula refers to this cell will, by its definition, incorporate the newest value in its own newest value. This is the functional equivalent of the traditional notion of events from imperative programming, which says, "whenever an event of interest occurs, the parts of the program interested in that event take the appropriate actions."

This example shows some of Forms/3's strategies for interactive visual data abstraction:

- programming of interactive abstract data types is in the same style as other programming;
- information hiding is controlled and communicated through physical visibility;
- every data type is visual, and its appearance is included in its definition;
- interactivity is supported for user-defined data types; and
- sample values are incorporated throughout the programming process, allowing immediate concrete feedback.

EVENT-HANDLING TO SUPPORT INTERACTIVE DATA TYPES. Adding interactivity to visual data types requires detection and handling of interactive events such as mouse movements. Surprisingly, generalized event-handling has been slow in coming to VPLs. Early VPLs supported event-handling only through application-specific built-in buttons and windows. Progress in this area may have been delayed by adherence to the traditional view of event-handling, which separates events from data and thus discourages incorporating event detection into types.

For VPLs that are declarative, one way to achieve compatibility between events and declarativeness is to view events as a vector of values along a time dimension. Forms/3 uses this approach. To enhance scalability, Forms/3 also supports combining events and data into new, higher level events. This event abstraction—the ability to define and respond to higher level, user-defined events—avoids over-proliferation of low-level event details.

Type checking

Type checking is used in all modern programming languages, including VPLs, to detect errors that arise from operation/operand incompatibility. There are three ways to perform type checking in any programming language: statically with types explicitly declared by the programmer, dynamically, or statically without explicit declarations. VPLs can use these traditional approaches in nontraditional ways. For some VPLs, this is in fact necessary to achieve goals such as simplicity and feedback.

STATIC TYPE CHECKING WITH EXPLICIT TYPES. Static typing with explicit type declarations has three advantages: (1) detection of type-incorrect programs at program translation time allows early feedback to the programmer; (2) use of type information by the translator improves executable code efficiency; and (3) explicit type declarations serve as helpful documentation.

Few VPLs use this approach, however, because it requires dedicating significant programmer time and program space to type declarations. It also presupposes an understanding of types. Further, this approach is not very amenable to polymorphism (although a few explicitly typed languages such as C++ and Ada partially deal with this problem). Without polymorphism, the programmer may need to create several nearly identical versions of the same code to accommodate the type system's inflexibility.

DYNAMIC TYPE CHECKING. Most VPLs use dynamic typing (without explicit type declarations) to achieve simplicity and flexibility. Traditionally, dynamic typing's biggest disadvantage is lack of feedback—runtime type errors may not be discovered until months after the program was entered, perhaps by someone other than the original programmer. This lack of timely feedback is especially at odds with responsive VPLs, whose objective is to provide immediate feedback.

Fortunately, however, some evaluation strategies for responsive VPLs bring runtime so close to translation time and program-entry time that dynamic type checking *can* produce feedback as soon as a type error is introduced. For example, in spreadsheets, concrete, immediate feedback about type errors is provided by evaluating a formula as soon as it is entered and displaying a special value if errors occur. This approach features simplicity and immediate visual feedback, although it cannot detect all type errors. For example, if cell *A* had the value "true," the error in the formula "if cell *A* then 3+4 else cell *A*+4" would not be detected.

STATIC TYPE CHECKING WITH IMPLICIT TYPES. Some VPLs are starting to use static type checking with implicit types, an approach borrowed from modern functional languages. The idea is for the system to make inferences from the type information implicit in a program. For example, if the programmer defines *X* to be 3, the system infers that *X* is an integer. If the programmer later defines *Y* to be equal to *X*, then the system infers that *Y* must also be an integer.

This approach can improve VPL simplicity and feedback capabilities. Ideally, it supports simplicity because a programmer does not enter type declarations and can program in a seemingly type-free environment, just as with dynamic type checking. Yet, because the approach is static, it produces immediate feedback about type errors at program entry time. The dataflow language Fabrik[4] was the first VPL to incorporate this approach. In Fabrik, each node contains input and output "pins" for attaching connecting wires to other nodes. Types are checked whenever the programmer attempts to connect two pins. A type mismatch displays a message and prevents connection. The type system fits seamlessly into Fabrik's programming environment—the programmer does not need to think abstractly about types, learn new concepts, or use new constructs.

There are obstacles to seamlessly incorporating implicit types into VPLs. Good error reporting requires display of the error-causing types; however, these types can be so complex that the error messages become incomprehensible. Also, most implicit typing approaches impose type-related language restrictions and thus miss the complete flexibility of dynamic type checking. Forms/3's implicit type system[9] partially addresses these obstacles by simplifying the types and eliminating some language restrictions.

Persistence

Data persistence—extending data lifetime beyond a single program execution—is vital to scaling up. Without it, many applications are impossible. A VPL can obtain data persistence in at least four ways:

1. provide a way for the programmer to explicitly handle file I/O;
2. incorporate database language capabilities into the VPL;
3. use a semitransparent approach, in which the programmer explicitly identifies instances of persistent data types but does not explicitly save or retrieve the data from the external store; or
4. use an entirely transparent approach, in which all data is persistent and is automatically saved and retrieved as needed.

Of these four possibilities, the first is rarely used with VPLs. Perhaps the reason is that burdening the program-

mer with explicit I/O responsibilities is inconsistent with making programming easier and clearer.

The second possibility is to draw on the extensive work for visual database access. However, few reports of this work extend its application to VPLs.

Semitransparent approaches to data persistence can enhance VPL simplicity by eliminating mechanisms like files and accesses. For example, in the Prograph interpreter, the values of data objects declared as persistent are automatically saved whenever the program file is saved; they are then available during subsequent executions.

Commercial spreadsheets use a fully transparent approach to persistent data. In a spreadsheet, all data is persistent—the programmer doesn't even have to think about it. In this simple approach, data can be changed in place by modifying the "program" (spreadsheet). Although the notion of total data persistence may seem inefficient, storage optimizations are possible since values based entirely on other values can always be recomputed instead of stored.

For the VPLs that are declarative, the extent to which data persistence support requires state modification is unresolved. Data persistence tends to go hand-in-hand with the notion of state change over time, a concept that can create problems with declarative languages' side effect-free nature. The spreadsheet approach addresses this issue in a declarative way by encouraging the programmer to build data histories through formulas in new columns that depend on earlier columns. Other possibilities lie in explicit approaches to time or change, such as those now being developed in the functional programming community.

Efficiency

Any programming language would like to claim efficiency as one of its attributes, and VPLs are no exception. However, the importance of immediate feedback makes efficiency critical when scaling up VPLs that are responsive. To maintain responsiveness, these VPLs need language translation and program execution efficient enough to provide immediate feedback on the validity and results of newly edited program fragments. They also need graphical display capabilities efficient enough to support direct manipulation. The challenge lies in developing techniques that can maintain these efficiencies regardless of program size.

EFFICIENT LANGUAGE TRANSLATION AND PROGRAM EXECUTION IN RESPONSIVE VPLS. To provide immediate feedback, responsive VPLs use an interpreter or incremental compiler for language translation. Efficient incremental translation techniques, long actively researched

Coding and beyond in Vista

Vista[1] is a responsive VPL for software engineers. It promotes evolutionary prototypical development of object-oriented software systems, seamlessly integrating design and implementation capabilities in a single visual programming system.

A key element of Vista's strategy in supporting design as well as programming is its multiparadigm approach, providing notations and models that follow basic software engineering principles. Object orientation supports reuse through inheritance, information hiding, modularity, and weak coupling. Dataflow is used to model data transformation and has an established visual representation. Control flow, represented in Vista by event passing, models sequences of events and actions.

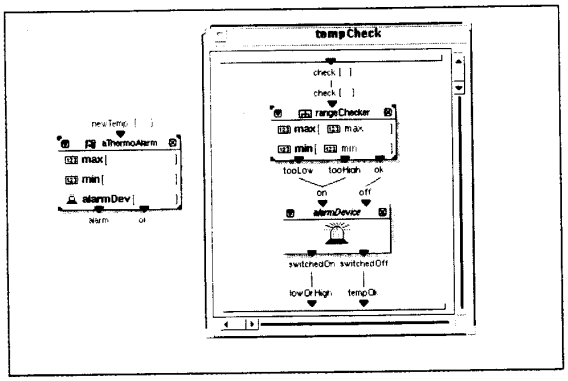


Figure E. Design reuse in Vista. (Reprinted with permission from *Visual Object-Oriented Programming: Concepts and Environments*, Prentice Hall, Englewood Cliffs, N.J. © 1995.)

A design component in Vista is a set of subcomponents that are wired together to specify dataflow and/or control flow. Top-down design is supported—the programmer can create and interconnect design components using only interface descriptions. As implementations of the design components are provided, the program comes alive incrementally. In addition to the input and output ports, the programmer can specify replaceable subcomponents as part of the component's interface. The replaceable subcomponents abstract the component's implementation into a design that can be customized by substituting new subcomponents.

Figure E shows a portion of a temperature-monitoring program. The component aThermoAlarm has input and output event ports (used for control flow via events) above and below its frame to receive and send events. Within its frame are the replaceable subcomponents: min, max, and alarmDev. When using aThermoAlarm, the programmer specifies the desired subcomponents. For example, alarmDev might be FlashWarningLight in one case and TurnOffPower in another. The tempCheck component shows a customized implementation of aThermoAlarm whose replaceable components have been instantiated with alarmDevice and same-named functions max and min.

Reference

1. S. Schiffer and J. Fröhlich, in "Visual Programming and Software Engineering with Vista," *Visual Object-Oriented Programming: Concepts and Environments*, M. Burnett, A. Goldberg, and T. Lewis, eds., Prentice Hall, Englewood Cliffs, N.J., 1995.

in the wider programming-language community, are about the same for VPLs as for other programming languages. VPLs that require parsing, however, are exceptions because parsing multidimensional combinations of graphical and textual tokens is not well matched with traditional grammars and techniques. Efficiently parsing programs entered through incremental freehand sketching and gestures is a growing research area, especially since the advent of pen-based interfaces.[10]

VPL researchers have developed several techniques to improve incremental execution efficiency. Prioritizing execution tasks is one such technique. The language "VPL"[5] is a dataflow VPL designed specifically for the computationally intensive domain of image processing in an interactive environment. It accords user interface performance the highest priority, handling it separately from the rest of the system, to obtain responsiveness even when intensive image-related computations are also in progress. "VPL" also improves responsiveness by distributing the workload to multiple computers, if available, and by canceling scheduled computations that have since become unnecessary.

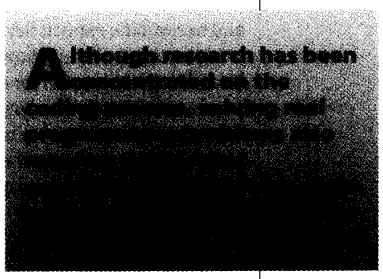
ThingLab II,[11] a constraint-based VPL, gains efficiency through an incremental constraint-satisfaction algorithm. The approach exploits the fact that many VPL constraints evolve gradually, one interaction at a time. The idea is to maintain sufficient information at each variable to inexpensively predict the cost of satisfying each constraint by examining only the immediate neighbors in the constraint graph. The cost information is then used to decide which constraint to satisfy next.

Several responsive VPLs use *lazy evaluation* to improve efficiency. In lazy evaluation, no expression is evaluated until its value is needed. Lazy evaluation also avoids duplicating some computations, and this duplication avoidance can be extended by saving prior results in a table or cache. (This is called *memoization*). In VPLs, some prior results are already saved for display purposes, which means they can be easily reused for computational purposes. Lazy evaluation and memoization are widely used in VPLs. Examples of VPLs that use one or both of these techniques include the Garnet, "VPL," and Forms/3 systems described in this article.

EFFICIENCY OF GRAPHICAL DISPLAY. Responsive VPLs must redisplay whenever computational events or user interactions affect objects on the screen. Possibilities for improving graphical display efficiency include displaying fewer items, displaying them less often, or displaying them less thoroughly. Some of the earlier described display approaches take the first option by displaying fewer details.

Garnet,[7] a constraint-based system for user interface development that incorporates many aspects of visual programming, avoids redisplaying objects. It incorporates a number of optimizations to minimize erasing and redrawing objects, particularly objects being changed. Garnet also uses one-way constraint satisfaction for greater efficiency, instead of full two-way constraint satisfaction.

Viva,[2] a responsive dataflow system for image processing, improves efficiency through a technique involving both spatial resolution of images and update rate. The spatial resolution is automatically adjusted according to the original image quality, and the update rate is automatically adjusted according to the rate at which data arrives from the camera; both can be adjusted downward by the programmer. These factors affect both display efficiency and execution efficiency for calculations that use the stored image data such as edge detection.



Viva also incorporates an approximate approach for some displays. Tonouchi et al.[12] extend this approach as an optimization technique. For example, approximations of display algorithms are used when the programmer is interacting with the system, then the accurate algorithm completes the display during a break in user interaction.

BEYOND CODING

Most VPL research has been concentrated on the coding process. However, solving real programming problems also requires support for noncoding programming tasks such as design, testing, and debugging. In devising solutions for these noncoding tasks, VPL researchers have the opportunity to seamlessly integrate support for noncoding aspects into the coding environment (see the sidebar "Coding and beyond in Vista").

Through immediate visual feedback, many responsive VPLs inherently support testing and debugging, resulting in uniformity with the programming process for these tasks. The dataflow language "VPL"[5] is a prime example: The "VPL" programmer does not differentiate among programming, testing, and debugging. For example, instead of setting breakpoints before executing, the programmer might watch data flow through the "probe" nodes in a dataflow graph. Based on this observation, the programmer may notice an error and temporarily disconnect an arc to halt execution at the erroneous node. When the programmer fixes the bug by connecting new or changed nodes, execution continues in the changed dataflow graph.

Many noncoding aspects are needed to bring a program to production status. Version control, separate compilation, conditional compilation, library generation, portability, extensibility, cross-language interoperability, namespace partitioning, and delivery tools form a partial list. Such pragmatic features are necessary for realistic VPL software development but, since most VPLs are still in the research stage, few have been implemented.

THE ESSENCE OF THE SCALING-UP PROBLEM is that simplicity, concreteness, explicitness, and feedback—the characteristics commonly used to achieve VPL design goals—are not normally associated with scalability. Yet, the examples presented show that they can be compatible. These examples demonstrate that static representations are possible for dynamic languages and that concrete sample values can be used with abstraction mechanisms. They illustrate ways to maintain the efficiency needed for immediate feedback

and to combine explicit details with context and in limited screen space. And they show emerging techniques to seamlessly integrate support for documentation, type checking, persistence, and noncoding aspects of software development. As these advances show, solutions to the scaling-up problem lie not in compromising the distinctive qualities of VPLs, but in devising ways to capitalize on them. ∎

Acknowledgments

We thank John Atwood, Sunanda Mishra, and the reviewers for their helpful comments, and Chih Lai, Chung-Cheng Luo, and Richard Wodtli for their contributions to this research. This work was partially supported by the National Science Foundation under an NSF Young Investigator Award and grants CCR-9215030/9396134 and CCR-9308649, and by Oregon State University's Oregon Space Grant Program under NASA grant NGT 40033.

References

1. N. Shu, *Visual Programming*, Van Nostrand Reinhold, New York, 1988.
2. S. Tanimoto, "Viva: A Visual Language for Image Processing," *J. Visual Languages and Computing*, Vol. 1, No. 2, June 1990, pp. 127-139.
3. P. Cox, F. Giles, and T. Pietrzykowski, "Prograph: A Step Towards Liberating Programming from Textual Conditioning," *1989 IEEE Workshop on Visual Languages*, IEEE CS Press, Los Alamitos, Calif., pp. 150-156.
4. D. Ingalls et al., "Fabrik, A Visual Programming Environment," *OOPSLA '88 Proceedings, ACM Sigplan Notices*, ACM, New York, Vol. 23, No. 11, Nov. 1988, pp. 176-190.
5. D. Lau-Kee et al., "VPL: An Active, Declarative Visual Programming System," *1991 IEEE Workshop Visual Languages*, IEEE CS Press, Los Alamitos, Calif., pp. 40-46.
6. M. Burnett and A. Ambler, "Interactive Visual Data Abstraction in a Declarative Visual Programming Language," *J. Visual Languages and Computing*, Vol. 5, No. 1, Mar. 1994, pp. 29-60.
7. B. Myers et al., "Garnet: Comprehensive Support for Graphical, Highly Interactive User Interfaces," *Computer*, Vol. 23, No. 11, Nov. 1990, pp. 71-85.
8. F. Penz, "Visual Programming in the Object World," *J. Visual Languages and Computing*, Vol. 2, No. 1, Mar. 1991, pp. 17-41.
9. M. Burnett, "Types and Type Inference in a Visual Programming Language," *1993 IEEE Symp. Visual Languages*, IEEE CS Press, Los Alamitos, Calif., Order No. 3970, pp. 238-243.
10. A. Apte and T. Kimura, "A Comparison Study of the Pen and the Mouse in Editing Graphic Diagrams," *1993 IEEE Symp. Visual Languages*, IEEE CS Press, Los Alamitos, Calif., Order No. 3970, pp. 352-357.
11. J. Maloney, A. Borning, and B. Freeman-Benson, "Constraint Technology for User-Interface Construction in ThingLab II," *Proc. OOPSLA '89, ACM Sigplan Notices*, ACM, New York, Vol. 24, No. 10, Oct. 1989, pp. 381-388.
12. T. Tonouchi et al., "Creating Visual Objects by Direct Manipulation," *1992 IEEE Workshop Visual Languages*, IEEE CS Press, Los Alamitos, Calif., Order No. 3090, pp. 95-101.

Margaret M. Burnett *is coguest editor of this special issue. Her biography appears following the guest editors' introduction, p. 16.*

Marla J. Baker *is a PhD student at the University of Washington, where she is supported by a National Science Foundation fellowship. Her research interests include programming languages, human-computer interaction, computer graphics, and software visualization. She received a BS degree in computer science from Oregon State University, where she was an NSF REU student researcher. She was a summer research intern at AT&T Bell Laboratories, and she is a student member of ACM.*

Carisa Bohus *is vice president of engineering for Solution Logic, a Portland, Oregon, software consulting firm. She is completing a master's degree in computer science at Oregon State University. Her main area of research is computer networking, especially asynchronous transfer mode. Bohus earned a BA in computer science from Portland State University in 1986.*

Paul Carlson *is a graduate student in computer science at Oregon State University. His current research interest is visual programming languages. He is also interested in applying new advances in computers and automation to the field of transportation. Carlson received a BS in computer science from the University of Minnesota in 1989. He was a NASA Space Grant Fellow at Oregon State in 1993-94. He is a student member of the IEEE Computer Society and the ACM.*

Sherry Yang *is a PhD student in computer science at Oregon State University. Her research interests include visual programming languages, user interfaces, and software engineering. She has been a software testing engineer at Hewlett-Packard and was lead programmer of CoCoPro, a software development cost estimation tool marketed by ICONIX Software Engineering. Yang received a BS with high scholarship and an MS in computer science from Oregon State University. She is a member of Phi Kappa Phi, ACM, IEEE, the IEEE Computer Society, and the Association of Women in Mathematics.*

Pieter van Zee *is a software design engineer for Hewlett-Packard. His research and professional interests include human/computer interfaces and user-centered design, information presentation and retrieval, workgroup computing, and visual programming languages. He worked from 1987 to 1992 for Siemens AG in Munich, Germany, building an object-based environment for software reuse and rapid prototyping. He received the BSE degree in electrical engineering and computer science from Princeton University in 1987 and is completing an MS degree in computer science at Oregon State University.*

Readers can contact any of the authors through Burnett at Oregon State University, Dearborn Hall 303, Corvallis, OR 97331-3202; e-mail, burnett@research.cs.orst.edu.