Influence of Visual Technology on the Evolution of Language Environments

Allen L. Ambler and Margaret M. Burnett
University of Kansas

With the availability of graphic workstations has come the increasing influence of visual technology on language environments. In this article we trace an evolution that began with the relatively straightforward translation of textual techniques into corresponding visual techniques and has progressed to uses of visual techniques that have no natural parallel using purely textual techniques. In short, the availability of visual technology is leading to the development of new approaches that are inherently visual.

Since the advent of language environments, use of visual technology has evolved from visualization of existing textual approaches to inherently visual new approaches. We survey this evolution here.

Terminology. In the seventies, much of the research on software development technology concentrated on the development of loosely integrated tools for supporting various phases of the software development and maintenance process. The Unix development environment is an example of such a *software support environment.*

Subsets of software support environments directly relate to the programming process of a single programmer. These subsets, or *programming environments*, distinguish facilities for designing, coding, editing, documenting, and debugging individual programming tasks from facilities required for planning, tracking, and managing entire software projects. Programming environments may encourage particular programming methodologies and particular languages. Some, referred to as *language environments*, are tightly integrated around a single language. Such tightly integrated, language-specific environments are the focus of this article.

One of the first language environments was Interlisp.[1] Developed in the early seventies, Interlisp provided the basic functionality we associate with a language environment: a fully integrated, language-specific environment with its own user interface, editor, interpreter, and symbolic debugger. More recent language environments have adopted and further refined many of Interlisp's features. In some cases, Interlisp's original ideas still represent the state of the art.

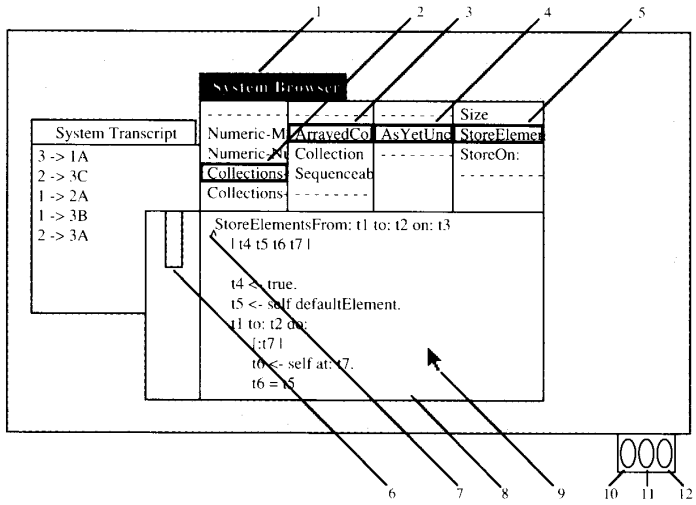
At the time language environments were first developed, computer technology was in a more primitive state. Bit-mapped graphics and even CRTs were less common than the standard Teletype terminal. In such an environment, the "friendliest" user interface was the briefest user interface. Therefore, language environments for many years were strictly command-driven. But the advent of visual technology brought dramatic changes in language environment user interfaces.

0018-9162/89/1000-0009$01.00 © 1989 IEEE

Layout and operation of the Smalltalk user interface

The Smalltalk environment assumes a high-resolution bitmap display, a keyboard, and a mouse with three buttons. The three mouse buttons are indicated by the appendage to the screen labelled (10, 11, 12).

We should note here that, in Smalltalk terminology, a window is a virtual screen large enough to contain an entire object. To accommodate physical screens, a window is viewed through a potentially smaller rectangular area that can be moved about on the surface of the window. Only this rectangular area, called a view, is actually displayed on the physical screen. By moving the view (a process called scrolling), you can view any portion of the window. In common usage, this distinction between a window and a view into it is often lost, with the term window used to refer to the physical window rather than the virtual window. We use "window" to refer to the physical window.



The System Browser window, identified by the tab (1), consists of the body (8) of a method "StoreElementsFrom:to:on:" (5) of object "AsYetUnclassified" (4) of object "ArrayedCollection" (3) of object "Collec-

Visual user interfaces

Multiple windows. Smalltalk[2] introduced not only a new language extending the object-oriented approach of Simula 67, but also a new and highly visual user interface. Alan Kay pioneered the research leading to Smalltalk-76's programming environment. He devised a user-interface paradigm he called overlapping windows. Kay's paradigm allowed for arbitrarily large virtual windows with modeless switching between windows and therefore between functions. (An interface with modes interprets commands with differing results depending upon the current mode. Typically, getting from one mode to another requires some definitive action, such as entering a key sequence.)

The fundamental aspects of Kay's paradigm were

- displays associated with several user tasks could be viewed simultaneously;
- switching between tasks would be done with the press of a button;
- no information would be lost in the process of switching; and
- screen space would be used economically.

The paradigm would serve as the basis for what Kay called an "integrated environment," in which the distinction between the operating system and an application would fade until every capability of any piece of software would apply to any piece of information. His paradigm contains the foundations for current user-interface paradigms not only for language environments, but also for system shells, including that of the now familiar Apple Macintosh.

See the sidebar on Smalltalk for a detailed example of the Smalltalk user interface.

An alternative windowing paradigm was introduced in Cedar,[3] a language synthesizing many of the same language environment concepts to produce an environment for an Algol-family language. This tiling paradigm is functionally similar to that of Smalltalk, differing primarily in detail and philosophy. See the sidebar on the sample Cedar screen for a description.

Multiple views. With Smalltalk came the concept of multiple windows; with Pecan[4] came the concept of multiple views. The distinction is that multiple views share a common internal data representation of the same data. Whenever any aspect of that data changes, then all views change simultaneously to reflect that change. The idea is that by representing data simultaneously in several ways, an individual user can choose those views that are most useful at a particular time.

In Pecan an internal abstract syntax tree supports concurrent views. The user never views this internal abstract syntax tree directly, but its information is available through various views. For example, the editor provides a program listing view; the declaration editor provides a separate view of the program's declarations; and the structured flow graph view provides a view of the program as a structured flow graph. Whenever any modification is made to the abstract syntax tree, via any view, all active views are notified of the change. The incremental compiler is treated as an undisplayed view; it receives notifications of any updates and recompiles the appropriate part of the code. The same approach is taken with the execution environment. The sidebar on Pecan shows several sample views.

Software through Pictures[5] is another graphically oriented language environment with a family of graphic editors and a centralized database for information about the system under development. Each editor supports a particular methodology of design or analysis developed in recent years. The major editors in Software through Pictures are the Dataflow Diagram Editor, which supports structured systems analysis; the Entity-Relationship Editor, which supports the entity-relationship data modeling approach; the Transition Diagram Editor, which supports the user software engineering (USE) methodology; the Data Structure Editor, which supports data structure definition; and the Structure

tions-Abstract" (2). The bold-lined boxes in each case indicate the selected item. To select a different item within the class hierarchy, you would use the pointer (9) to point to the new selection and click the left button (10). Given the hierarchical ordering (from left to right), selecting a new item in any of panes (2, 3, 4) deselects the selections for all panes to the right of the new selection. You can scroll each selection pane should there be too many entries to fit.

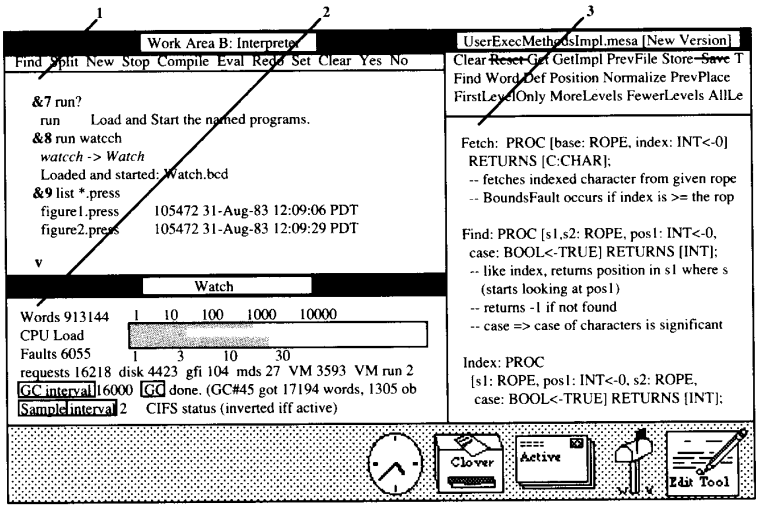
Whenever the pointer (9) is within the System Browser window and within one of the five panes (2, 3, 4, 5, 8), a scroll box (6) appears along the left side of the containing pane. The scroll box lets you scroll the text of the corresponding pane by moving the pointer into the scroll box and either grabbing and dragging the gray slider up or down within the scroll box to indicate the relative position desired within the corresponding pane or by moving the pointer within the scroll box slightly to the left or right of the slider and clicking it to move down (left of the slider) or up (right of the slider) one pane of information.

The caret marker (7) indicates the current position within the text where, if you typed, characters would be inserted. You can move this marker arbitrarily by pointing and clicking with the left button. In addition, you can select sections of text by pointing at one end of a desired selection, depressing and holding the left button while moving the pointer to the other end, and releasing the left button. While you make the selection and until you deselect it (by the next click of the left button), the selected text appears in reverse video. You can copy or delete selected text by choosing copy or cut from a pop-up menu associated with the center button (11). You can subsequently paste (insert) it by moving the pointer to a new location and choosing paste from the same pop-up menu.

A pop-up menu appears whenever you depress the center or right buttons. It contains a vertical list of possible operations applicable in the current context. You select an operation by using the pointer and clicking the left button when the appropriate operation appears in reverse video. After you select an operation, the pop-up menu disappears. The list of possible operations associated with a pop-up menu changes from one pane to another, making the choice of operations sensitive to context.

The right button (12) can also be used for pop-up menus. The convention in Smalltalk is that the center button invokes operations appropriate to the current pane, while the right button invokes commands relating to the window in relationship to all other windows. Typical right-button commands allow closing, reframing (changing a frame's size, shape, and screen position), or overlaying windows. As you can see in the figure, more than one window can be open at any time, with the active window indicated by having its tab (1) distinguished (by a reverse video title).

Sample Cedar screen



In Cedar, windowing features, although similar to those in Smalltalk, are enhanced and have several added features, including icons and buttons. A physical window is called a *viewer*, and viewers are tiled, rather than overlapped as in Smalltalk. Tiling is a method of placing viewers adjacent to one another to cover the entire screen without covering any viewer, partially or totally, with any other viewer.

Tiling versus overlapping has generated considerable discussion. With tiled viewers the system automatically handles sizing and placement of the viewers, but the viewers are therefore not likely to conform to their contents in a way that maximizes the visibility of those contents. With overlapping viewers, the opposite is true: you must handle sizing and placement manually. This allows you to make each viewer conform to its contents.

An open viewer occupies screen space. When a viewer is closed, it yields its screen space and appears only as an icon found at the bottom of the screen. When you reopen the iconic representation of a viewer by selecting and clicking its icon, the viewer is reallocated space on the screen. While a viewer is closed, it is suspended, but not terminated. Resizing a viewer has the effect of resizing at least one other viewer unless it was the only viewer, in which case it cannot be resized.

Within a viewer, other special viewers are possible. In particular, buttons are a special group of viewers used only for invoking procedures. Buttons are represented as text, possibly surrounded by a small box, or as icons. Selecting a button has the effect of invoking a procedure. This procedure typically performs some action on the associated viewer. Often buttons are arranged in menus and displayed just below the caption bar, but they may be placed arbitrarily within another viewer.

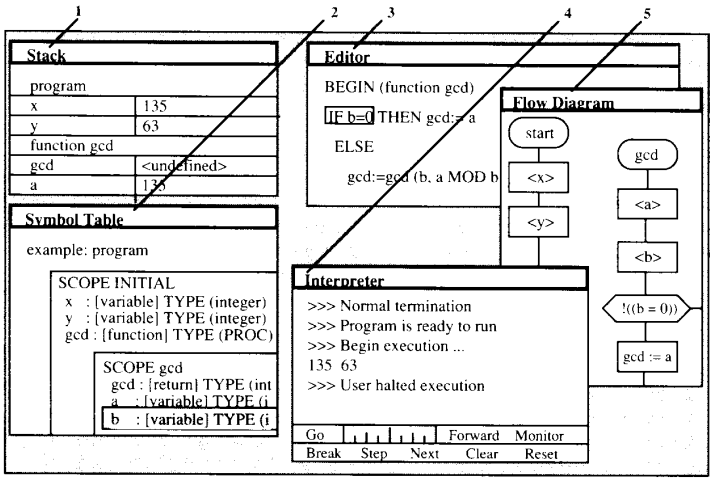
Unique to Cedar is a *guarded* button, displayed as a single line drawn through the normal button representation. It indicates a command with a potentially destructive effect. To invoke a guarded button, you must click it twice within a short time interval.

The Cedar interpreter viewer shown here shows an instance in which the DWIM (Do What I Mean) facility has corrected a command error (1). This screen also has the performance measurement tool active (2) and shows the on-line documentation (3). In the documentation viewer, the buttons Reset and Save are examples of the guarded button feature.

We adapted this figure from information provided in W. Teitelman's "A Tour Through Cedar," published in the April 1984 issue of *IEEE Software*.

Sample views in Pecan

The stack view (1) shows the values of the variables defined at the current point of execution. The two variables *x* and *y* are part of the INITIAL block, which includes the function gcd. The symbol table view (2) illustrates the scope of each symbol, including *x*, *y*, and gcd, as well as gcd's formal parameters *a* and *b*. The editor view (3) shows the section of code currently being executed, namely the if *b*=0 statement within the gcd function. The interpreter view (4) displays the execution status and user inputs and outputs. The bottom portion of the interpreter view contains the ruler bar and several command selections (Go, Break, Step, etc.) that allow you to control the speed and manner of execution. The flow diagram view (5) shows the diagrammatic version of the section of code currently being executed (the gcd function), with the current statement (if *b*=0) highlighted.



We adapted this figure from Reiss' article, "Graphical Program Development with Pecan Program Development Systems," published in the May 1984 issue of *ACM SIGPlan Notices*.

Chart Editor, which supports structured design. Each editor can automatically add information to the centralized database as it is generated and modified.

Beyond the visualization of textual languages. From experience with Pecan and generating systems to provide graphical views of otherwise textual languages, Reiss observed that users were limited by the inherent one-dimensionality associated with the underlying textual languages.[6] He concluded that effective use of the two-dimensional capabilities of graphical views required working with languages whose natural expression was graphical, not textual. Reiss responded by developing the Garden[6] system to accept descriptions of visual as well as textual programming objects. Others, making this same observation, have worked on new approaches to visual languages (see "Visual languages" below).

Like Pecan, Garden uses a common internal representation model, in this case an object-oriented environment complete with inheritance. New conceptual models are described by defining new objects that represent the fundamental objects of the new conceptual model, then by defining the interpretation of manipulations to these new objects. Each of these new objects can be given a visual representation as well as a textual representation. The interpretation of visual manipulations can be described as well. To simplify the process, Garden provides an extensive library of database, process management, user-interface, dynamic display, debugging, and editing facilities usable in defining and testing new conceptual models.

See the sidebar on Garden for a description of the process of defining a new conceptual model.

Visual editing

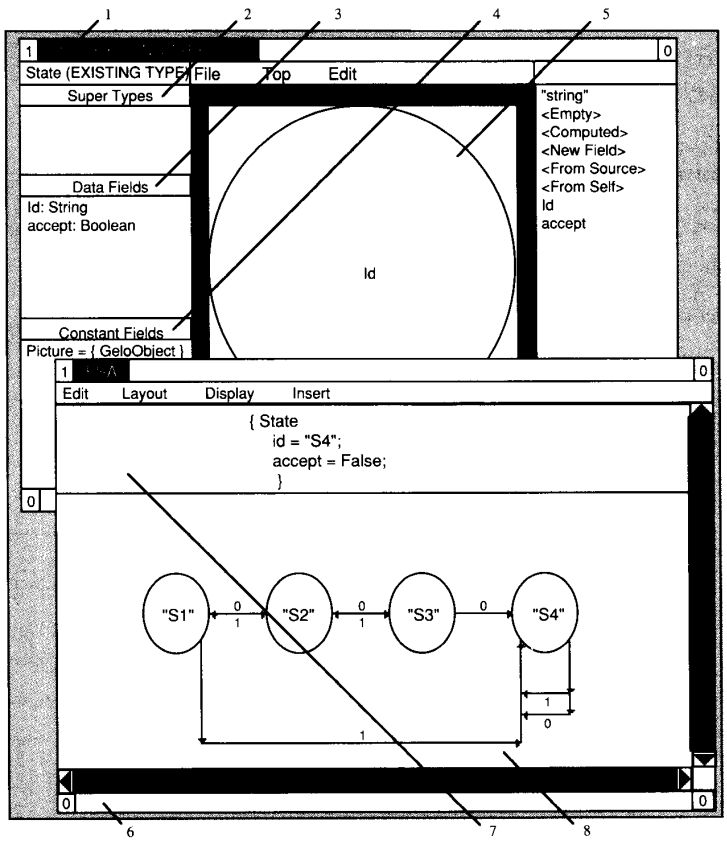
Syntax-directed editing. One of the more prevalent visual editing techniques to appear in language environments is syntax-directed editing. Here, the editor is aware of the language's syntax and can thus either give the programmer immediate feedback whenever a syntax error is typed in or prevent such errors completely by forcing the programmer to use a template based upon the language syntax.

As with windowing systems above, whole papers discuss syntax-directed editors in detail. We will just touch on a few of the more prominent features as they have appeared in language environments. The two syntax-directed editors discussed, the Cornell Program Synthesizer editor and the Aloe editor used in Gandalf, follow the philosophy that if a portion of the program is created based on some template, then the structure created by that template must be preserved during editing. This philosophy allows the use of visual hierarchical traversal techniques based on a program's structure. The examples shown in the accompanying sidebars on the Cornell Program Synthesizer and the Aloe editor illustrate the visual techniques used in traversing and editing programs using the two editors.

The Cornell Program Synthesizer[7] is a language environment for a subset of PL/1 called PL/CS. PL/CS programs are constructed through a syntax-directed editor that uses program-structure-based editing. Structural aspects of PL/CS programs, such as blocks and statements, are entered and edited through a special set of language-specific commands. These language-specific commands generate templates that outline the language's syntactic structure and provide a preformatted, properly indented, parentheses-matched form with place-holders to be filled in by the programmer. Non-structure-based phrases, such as expressions, comments, and identifier lists, are entered using conventional text-based editing commands, which are only available for such nonstructural phrases.

The programmer is similarly restrained from modifying a statement. Text entered to replace a place-holder must continue to match the syntactic part specified by the

Defining a new conceptual model in Garden



In Garden, a new conceptual model can be described in three steps. The type structure is specified, the semantics for each type is specified, and the syntax (textual and/or visual) is specified. Three types are needed for the finite state automaton in the figure: one describing a state, one describing a transition, and one describing the complete automaton.

The type editor window (1) shows the type definition and visual syntax for a state. It has no Super Types (2), includes the data fields id and accept (3), and includes the constant field Picture (4), which maps a state to a basic visual object (5) used to display the id field. All fields of a data structure need not be displayed with the visual syntax. Also, a single data structure can have mappings to more than one visual representation.

In addition to basic visual objects, composition objects are available. For example, a layout object can be used to display more complex or variable data structures than simple fields or records. The complete automaton has been mapped to a layout object composed of the basic object representation for states and several arc objects visually representing the transitions.

After the types are defined, the programmer defines their semantics (not shown in this figure) by writing functions using the textual Lisp-like language currently provided by Garden and other graphical or textual metaphors that have already been defined.

In (6), the programmer is experimenting with an instance of the automaton type (8). Using this editor, the programmer has created the states using a series of selections, insert commands, and dialogue boxes. Similarly, the transitions have been created by selecting the from and to states, using the connect command, and filling in the values requested in the dialogue boxes. Finally, after several sample evaluations (not shown), the current state has become "S4" (7).

We adapted this figure from Reiss' "Garden Tools: Support for Graphical Programming," published in *Advanced Programming Environments*, Lecture Notes in Computer Science #244, from Springer-Verlag, New York.

Syntax-directed editing using the Cornell Program Synthesizer

In the Cornell Program Synthesizer, you enter language-specific commands as text, then depress a special function key.

The editor command

```
.i
```

would produce the following template:

```
IF (⟦c⟧ondition )
   THEN statement
   ELSE statement
```

where "condition" and "statement" are place-holders. The box around the letter "c," ⟦c⟧, shows the placement of the cursor; it denotes that "condition" is to be replaced by user-entered text. When the programmer enters text to replace "condition," the synthesizer will check that the entered text is a Boolean phrase. Should the entered text not match the syntactic part designated by the place-holder, the programmer will be so advised. Similarly, the programmer replaces the two occurrences of place-holder "statement" by further language-specific commands, introducing yet more templates.

The use of language-specific commands to generate the more complex syntax of the language PL/CS and the syntactic checking of place-holder replacement text prevents the programmer from entering syntactically incorrect programs. Further, some static semantic checking prevents errors such as referencing an undeclared identifier.

Syntax-directed editing using the Aloe editor

The language-specific command "wh" might be used to cause the place-holder

```
%stmt
```

to be replaced by

```
while %cond do
        %stmt
od
```

This template provides the structural parts of a while statement: the required syntactic tokens (while, do, and od) with proper indentation, and place-holders for a conditional expression (%cond) and the statement body (%stmt). The cursor is placed automatically at the first place-holder (%cond). The user may then type in an appropriate Boolean expression, replacing the %cond place-holder or, depending upon the abstract syntax, the user might invoke another constructive command expanding and replacing %cond. For instance, replacing %cond with the template for an and expression yields

```
while ( %expr and %expr ) do
        %stmt
od
```

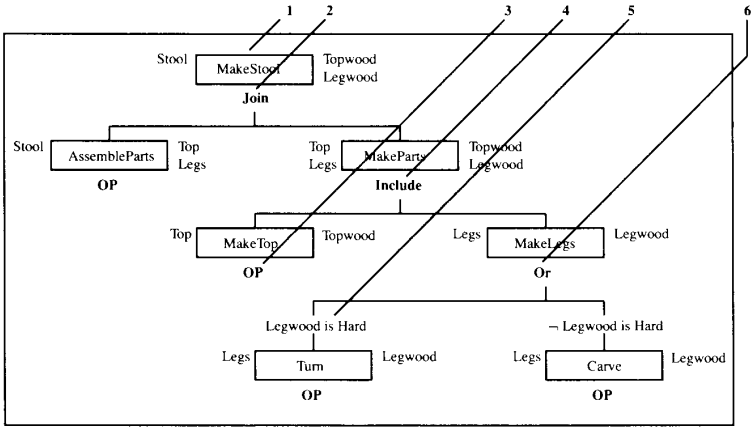
The abstract syntax can be specified such that all expressions are composed entirely by using constructive commands or by entering a simple variable name or value. When specified in this way, editing, which is required to follow the parse tree, can become very rigid. For instance, to edit $((a + b) + c)$ to be $(a + (b + c))$ requires deleting the syntactic part $(a + b)$ and replacing it with a, followed by replacing c with the syntactic part (%expr + %expr), and finally replacing the two occurrences of %expr by b and c. The alternative is an abstract syntax that stops short of generating expressions, that is, there are no syntactic parts specified as replacements for the place-holder %expr. Thus, expressions would be typed and edited as single nodes of the parse tree. Action routines might still be used to check syntactic correctness.

Up and down movement (cursor-previous and cursor-next) corresponds to lateral movement within the parse tree. In and out movement (cursor-out and cursor-in) corresponds to vertical movement within the tree. For instance, in the code shown here, the structural part is shown on the left and the cursor position is identified by a box. If the cursor is initially positioned as (cursor), then up, down, in, or out causes cursor movement as shown.

if %cond then
%stmt
else
%stmt
	if eof(input) then
	done := true;
	else
	read(ch);
		if eof(input) then
		done := true;
		else
		read(ch);
			if eof(input) then
			done := true;
			else
			read(ch);
				if eof(input) then
				done := true;
				else
				read(ch);
					if eof(input) then
					done := true;
					else
					read(ch);

Structural Part	(cursor)	(up)	(down)	(in)	(out)

Graphical representation in Use.It



The binary tree depicts the decomposition of a process for making wooden stools. For each node, represented by a box (1), the input objects are listed on the right of the box and the output objects are listed on the left side of the box. The join control structure (2) decomposes MakeStool into two nodes, MakeParts and AssembleParts. Use of join requires that the right node execute before the left node. The right node has the same inputs as its parent, but cannot produce any of the outputs of its parent, only intermediate values. The left node has as inputs only the intermediate values of the right node and must produce as outputs all of the outputs of the parent node.

Next, the include control structure (4) is used to further decompose MakeParts into MakeLegs and MakeTop. Use of the include allows its subnodes to execute in any order or concurrently. The inputs and outputs of each node are disjoint, and the union of the inputs and outputs is exactly the inputs and outputs of the parent node.

Finally, the or control structure (6) is used to decompose MakeLegs into Carve and Turn. Use of the or control structure requires that exactly one of the subnodes be executed depending upon the evaluation of a condition (5). Both nodes have all inputs of the parent node and produce all outputs of the parent node.

We adapted this figure from J. Martin's *System Design from Provably Correct Constructs*, published by Prentice Hall, Englewood Cliffs, N.J., 1985.

place-holder. A syntactic structure entered through language-specific commands must be edited as a structural unit through language-specific commands. The programmer can clip, delete, and insert entire structural units. When a structural unit is clipped or deleted, it is replaced by its place-holder. Likewise, inserted structural units must replace corresponding syntactic parts; they will be reindented and rechecked for syntactic correctness.

The syntactic structure affects cursor movement as well. When the cursor is moved, it will skip over all but text and structural units, permitting itself to be positioned only where modifications are allowed.

A program can be run at any stage of development. Each structural unit is transformed into interpretable code as it is input and checked. Execution is thus immediate. Once begun, execution is suspended when the interpreter encounters either an error or an unexpanded place-holder. Upon detecting an error, the interpreter indicates the error and passes control back to the user, who then may correct the problem or restart. Similarly, when the interpreter encounters an unexpanded place-holder, it will stop, allow the user to insert the required code, and then continue.

The Cornell Program Synthesizer also has a visual tracing capability, a forerunner of program animation. (Program animation refers to the process of displaying the operation of a program through visual representations that dynamically change as the program executes. Selected variables or even entire data structures are displayed, usually graphically, with their contents changing dynamically as the program alters their values. For example, for a program that maintains a binary search tree of customers, the tree of customer names might be displayed graphically. Whenever a node is added or deleted, the display of the tree would be updated on the screen.)

In the Cornell Program Synthesizer, a cursor follows the flow of control through the program during execution. In addition, the synthesizer provides additional windows for monitoring variables and displaying results. The variable monitoring feature dynamically displays variable values using a least-recently modified approach to displaying large numbers of variables in limited screen space.

See the accompanying sidebar for an example of syntax-directed editing using the Cornell Program Synthesizer.

The Gandalf system[8] uses an editor generated by Medina-Mora and Notkin's Aloe editor-generator. Such an editor has a common kernel of editing commands that are language-independent and a set of constructive commands that are language-dependent. Editing commands are used for generic operations such as manipulating parse trees. Typical editing commands delete a construct or move the cursor. Constructive commands use abstract and concrete syntax descriptions to generate templates for each structural part. Cursor movement in building or editing a program parallels movement in the parse tree. In particular, the cursor is moved up, down, in, or out to the appropriate place-holder.

See the accompanying sidebar for an example of syntax-directed editing using the Aloe editor.

Specification-directed editing. The concept of structure-based editing can be extended further by imposing additional rules on the structure of programs and enforcing these rules through the editor. These rules can be used for purposes such as ensuring that only programs provably in accordance with a prescribed set of specifications may be entered. This section discusses two graphically oriented language environments that use such structure-based editors.

Higher Order Software's Use.It[9] takes a formal graphically oriented approach to program construction. Use.It generates code, in a variety of languages, directly from formal specifications, which are entered and edited using a structure-based editor that enforces decomposition based on provably correct design axioms that limit the interactions between modules.

Use.It applications are represented as binary trees known as control maps. Each node of the tree represents a function having a number of input and output objects. In a graphical representation, input objects are listed to the right of the node and output objects are listed to the left. Leaf nodes are typically irreducible low-level primitive functions. Non-leaf nodes are decomposed into subfunctions using the control structures join, include, and or, each of which is consistent with the six axioms on which the methodology relies.

In Use.It, the decomposition of functions into primitive subfunctions is formally specified and rigidly enforced by the graphical editing process. The preciseness of this decomposition ensures the internal consistency of the code generated by Use.It. In particular, Use.It ensures that any decomposition is logically consistent with the six basic axioms describing the structural interfaces between pieces of code. These axioms formally define a reliable system for structured coding.

See the accompanying sidebar for an example of graphical representation in Use.It.

A similar graphical editing technique is employed by PegaSys,[10] a system that uses graphical images to formally represent program design specifications. The emphasis in PegaSys is on the use of formal graphical specifications as documentation and as a means to verify that (user-written) program code meets specifications.

Graphical specifications are referred to as formal dependency diagrams. FDDs are manipulated graphically using editing techniques subject to system-imposed syntactic and semantic constraints. The constraints ensure that certain properties are preserved during the process of designing a program by successive refinement.

This concept of a rigorously controlled decomposition based on a mathematical logic is similar to Use.It, although the rules used and the properties preserved differ for each system.

Once the graphically edited design specifications are complete, they are manually mapped onto the implementation code, developed using PegaSys's structure-oriented Ada editor. Finally, the system formally verifies that the program is consistent with its design specifications.

The sidebar on PegaSys shows two levels in a formal dependency diagram hierarchy.

From visual editing to visually transformed programming. Each of the above visual editing systems is to some degree template-oriented. That is, some portion of a program is identified for replacement, a desired replacement template is selected, and then the replacement is performed as specified by the template, possibly subject to certain rigorously controlled rules. Clearly, this template-oriented approach can be carried to such an extreme that the entire program is constructed by visual editing techniques alone.

The earliest approaches to visual programming consisted of visual editors for traditional imperative textual languages. Often the control flow was given a pictorial or diagrammatic representation, such as a flowchart or Nassi-Schneiderman diagram. Pecan included such views to represent Pascal programs. Later approaches discarded the textual version completely, using the diagram version as

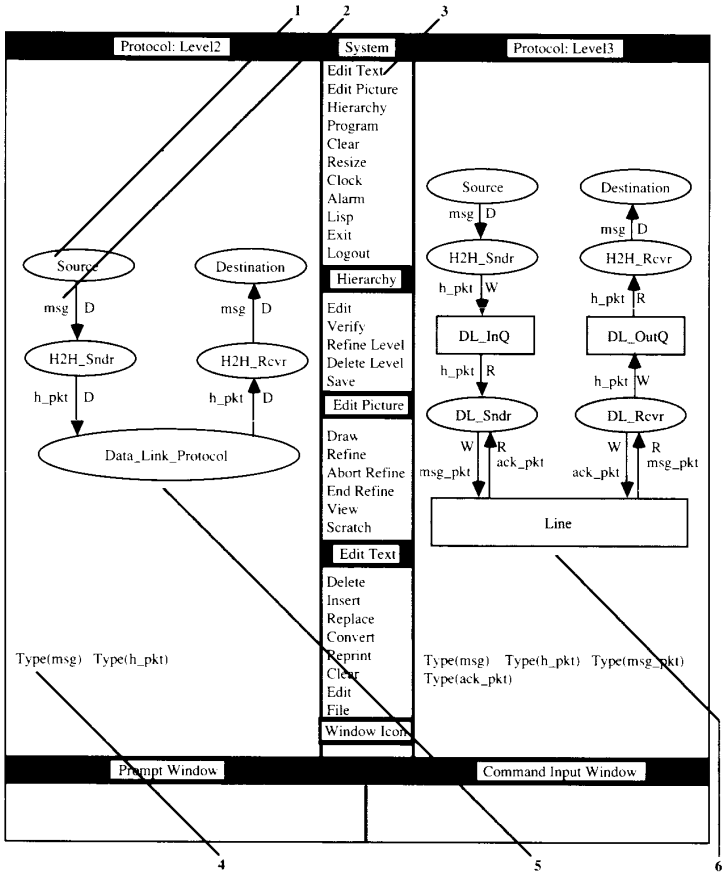
Formal dependency diagram hierarchy using PegaSys

The accompanying figure shows two levels in a formal dependency diagram, or FDD, representing a networking scheme designed using PegaSys. In Level 2 of the figure, the processes Source (1), Destination, H2H_Sndr, H2H_Rcvr, and Data_Link_Protocol are all denoted by ellipses. Dataflow relations are denoted D on the labeled arcs msg (2) and h_pkt, which denote the types of data being passed. The active types msg and h_pkt are also listed at the bottom of the screen (4).

Level 3 is a refinement of Level 2, developed by the user selecting the entity to be refined (5), then, using the appropriate menu commands (3), constructing the replacement entity (6). Note that the dataflow relations D for this entity have also been refined into read relations, denoted R, and write relations, denoted W.

To augment the FDD, PegaSys also includes a facility to associate text with any icon.

We based this figure on Moriconi and Hare's "PegaSys: A System for Graphical Explanation of Program Designs," included in the July 1985 *Proceedings of the ACM SIGPlan 85 Symposium on Language Issues in Programming Environments.*



the only representation of the program and graphical editing of the diagram for construction and editing of the program.

A good example of such a system, Pict/D,[11] uses a conventional imperative language paradigm that replaces all keywords with iconic representations. The resulting language creates and manipulates flowcharts with the added ability to create new icons to represent subcharts. Thus, the language semantics are conventional, programming requires conventional concepts and thought processes, and the resulting programs are of equivalent complexity to the corresponding textual programs. Pict/D concentrates on using visual images to improve our ability to comprehend and edit programs.

Reiss's observation about editable views (see "Beyond the visualization of textual languages" above) applies to visual editing as well. Visual editing of an otherwise textual language can severely limit the power and usefulness of visual technology. Clearly, enforcing structural decomposition rules still has value, as in Use.It and PegaSys. However, some of the early approaches to visual programming that used graphical technology merely to replace corresponding typing have drawn the criticism that they do little more than force a programmer to use menus and other graphical techniques for operations that can often be typed textually faster. Unfortunately, this criticism has unjustifiably been extended to visual programming in general.

Visual languages

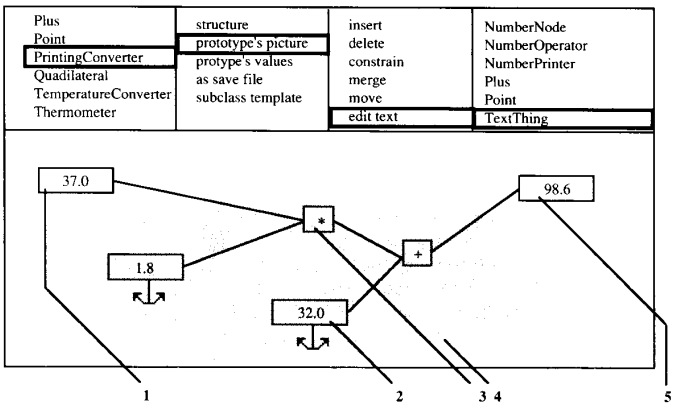
Visual programming uses visual, rather than textual, technology. The development of visual programming languages represents a further step in the evolution toward more visual language environments. Visual languages are generally subdivided into two categories. The first category, *visually transformed languages*, includes those visual languages that are inherently nonvisual but have superimposed visual representations. These are the visually edited traditional languages discussed in the prior section. They emphasize facilitating our ability to specify and comprehend programs using existing language paradigms.

The second category, *naturally visual languages*, attempts to develop new programming paradigms whose inherent natural expression is visual and for which there may be no strictly textual equivalent. In this section, we survey several divergent naturally visual languages and language environments.

By the very nature of the concept of visual programming, it is often difficult to separate a visual language from its language environment. It is this high degree of integration between the language and its environment that makes naturally visual language technology such an influential

A Celsius-to-Fahrenheit temperature converter constructed in ThingLab

Inserting several instances of previously defined classes into the window (Constant (2), Times (3), and Plus) and entering the constants' values of 1.8 and 32.0 creates the prototype of the TempConverter (4). Connecting two instances of NumberPrinters (1,5) to display the Celsius and Fahrenheit temperatures results in the PrintingConverter as shown in the figure. It works because the constraints placed on Plus and Times force adjustment of one of the Celsius or Fahrenheit temperatures. Whenever the Celsius temperature (1) is edited, the Fahrenheit temperature (5) will be adjusted. Also, because of the multi-way nature of the constraints, editing of the Fahrenheit temperature will result in adjustment of the Celsius temperature.



We based this figure on Borning's "The Programming Language Aspects of ThingLab," published in the October 1981 issue of *ACM Transactions on Programming Languages and Systems.*

area in language environment research.

Dataflow paradigm. Visual languages based on the dataflow paradigm might be considered visually transformed languages. However, it can be argued that the dataflow paradigm is based on dataflow diagrams, in which a program is composed of functional modules, with connecting paths between inputs and outputs. In textual languages based upon this paradigm, dataflow diagrams are normally drawn first as part of the program design process and then translated into textual syntax. Visual languages based on this paradigm simply omit the translation to text.

Constraint-based paradigms. Many constraint-based programming languages are well suited to a visual representation. The advantage of a visual representation for this paradigm lies in the multi-way nature of constraints. A constraint may affect many variables, which in turn may affect many more; such complicated interrelationships are often easier to see in a diagrammatic representation than in a textual representation.

ThingLab[12] is an experiment in constraint-based programming. Given a set of constraints (rules) describing the invariant properties and relationships of all objects in a particular problem space, then the set of solutions is the set of values that simultaneously satisfy all constraints. With sufficient constraints, the set of solutions can be made arbitrarily small, thus effecting a solution.

This approach in many respects resembles logic programming, where constraints are analogous to rules and relations and finding a set of values that satisfies all given constraints is analogous to resolving a query. ThingLab is a constraint-oriented simulation language environment that supports the construction of simulation environments using constraints and constraint-inheritance mechanisms. It incorporates inheritance (from the object-oriented paradigm of its underlying implementation language) and allows inheritance of the constraints themselves. This approach has the disadvantage that the inherited constraints may conflict when multiple inheritance is allowed. Still, many recent constraint languages incorporate various forms of inheritance.

Based on Smalltalk and heavily influenced by Sutherland's SketchPad, an early constraint-based graphical communication system that allows the user to directly draw and edit pictures on the screen using a light pen, ThingLab incorporates elements of graphical programming-by-demonstration. Its influence shows in later visual language environments, most notably ThinkPad and Rehearsal World, both described below.

See the sidebar for an example of a Celsius-to-Fahrenheit temperature converter constructed in ThingLab.

Programming-by-demonstration. This is a naturally visual process for which there is no textual equivalent. Programming is done by graphically manipulating the data on the screen, demonstrating to the computer what the program should do. The advantage to this approach is obvious — it is easier for a programmer to perform a process than to describe textually how to perform the process.

ThinkPad is a declarative, graphical, programming-by-demonstration language and environment.[13] To perform programming-by-demonstration, the user graphically manipulates a diagrammatic representation of a data structure to demonstrate the operations on the data. An automatic mapping of the user's manipulations to Prolog code implements the program.

In ThinkPad, the user defines a data structure by drawing its graphical properties. In fact, a single data type can have multiple diagrammatic representations ("forms"), all specified graphically. The

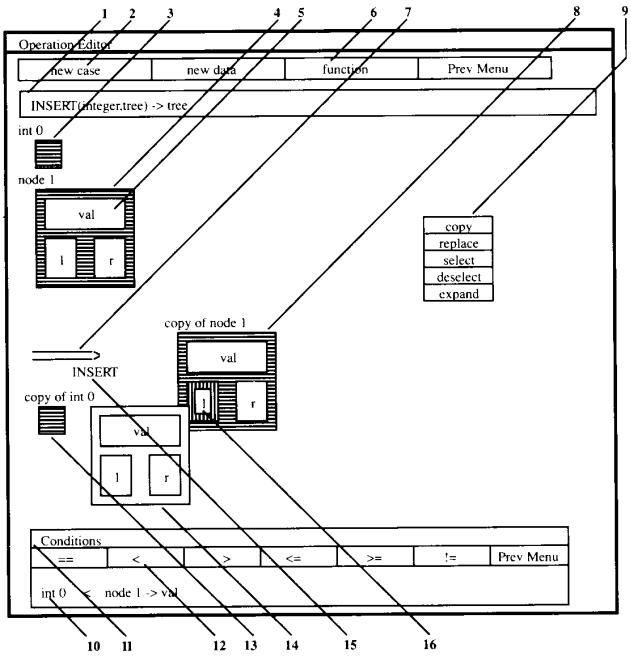
Defining an operation in ThinkPad

A simple example will illustrate the use of ThinkPad. The problem is the insertion of nodes into a binary tree. The first step (not shown) is to define a binary tree, using ThinkPad's data editor. This is done by selecting a shape from the Shapes menu to represent a node (in this example, a rectangle) and resizing and repositioning it to suit. The fields within the node are similarly selected, resized, and repositioned.

The user began by naming the operation INSERT and specifying the type of all parameters and results (1). The system knows from the parameters that two graphical forms, an integer and a tree node, will be needed for the operation, which it automatically displays (3,4).

Next, the function is described by identifying a series of cases and for each case demonstrating the corresponding operation. Each case is distinguished by a unique condition (11). In this case the condition, int 0 < node 1 -> val (10), is entered by sequentially selecting int 0 (3), the less-than symbol (12), and val of node 1 (5).

The user now specifies the results (7) of the INSERT operation by manipulating the forms on the screen to demonstrate the operation for this case. First, the tree node is copied, by depressing the copy button (9) and then selecting node 1 (4) to create a new node (8).



Next, the function button (6) is pressed and INSERT is selected (15) to indicate a function recursive call. Then, the arguments are indicated by copying int 0 (13) and expanding (9) l (16) of the copy of node 1 (8), giving the structure of the left node (14).

The user has now completely defined operation INSERT for one case and can continue in the same way for each remaining case (2).

We adapted this figure from "ThinkPad: A Graphical System for Programming by Demonstration," by Rubin, Golin, and Reiss in the March 1985 issue of *IEEE Software*.

multiple forms capability allows ThinkPad to support multiple views of a single data type. The user can define operations on the data structure by manipulating the representation on the screen. In addition, the user may specify type constraints that the operations must preserve. Pointing and clicking selects the desired conditions, provided by the condition editor.

Internally, the data structure is represented by a set of Prolog assertions about the data. Relationships in the data structure are also mapped into Prolog assertions, and type constraints are implemented as predicates. All the assertions pertaining to one data structure are grouped into a separate set of Prolog clauses. Graphical specifications are stored in another library, with cross links to the set of Prolog assertions.

Because ThinkPad internally defines operations as transformations from one arrangement of the data structure to another, manipulating the data structures graphically is the equivalent of programming. However, while there is a direct mapping from the graphical manipulation of the data structures to a program (in this case implemented in Prolog), there is no mapping from the program to the graphical representation of the data structures. Execution and debugging revert to the text-based Prolog code, rather than to the visual interface that exists during the creation of the code.

See the accompanying sidebar for an example of defining an operation in ThinkPad.

Rehearsal World,[14] a visual programming language environment devised for nonprogrammers, is one of the earliest environments to fully support visual programming. Rehearsal World uses a theater metaphor. The basic components, called performers, interact with each other on a stage (the screen) by sending cues. The screen is the stage upon which performers (objects) perform the actions the user has taught them for a particular production (program).

Rehearsal World includes several predefined primitive performers, each of which understands a large predefined set of cues. Any existing performer can be copied; thus, each performer acts as a prototype from which other performers can be generated. The use of predefined performers and cues is, in essence, the integration of predefined code segments into the language environment itself.

The actual code generated by Rehearsal World is Smalltalk, but the programming process normally occurs strictly through graphical or visually oriented manipulation; hence, the user does not have to know Smalltalk to program in Rehearsal World. Likewise, code is normally debugged visually, by observing the performers' behavior during rehearsals, although additional lower-level debugging facilities are available.

For a closer look, see the sidebar "The Rehearsal World theater."

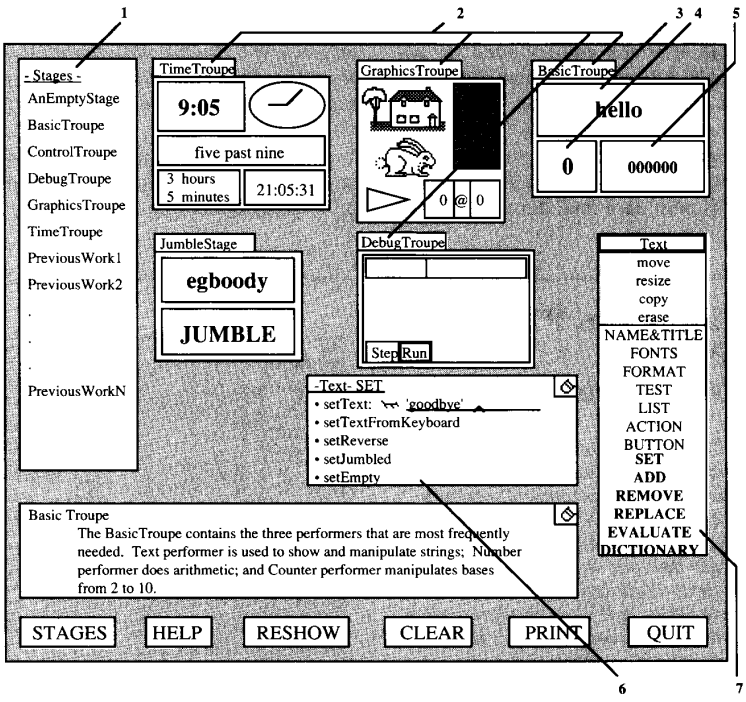
In PT, or Pictorial Transformations,[15] programming takes the form of first describing visual data representations, then

The Rehearsal World theater

The user starts by selecting from the menus of available stages (1) and troupes (2). Each troupe contains a set of performers represented as icons. For example, the BasicTroupe consists of a Text performer (3), a Counter performer (4), and a Number performer (5). For each performer, a category menu (7) is available as a pop-up display via a mouse button. This category menu contains certain commonly used cues (in lowercase) and categories of other cues (in uppercase). Most categories are common to all performers; a few (in bold) are unique to an individual performer. Selecting a category gets a cue sheet (6), which lists the cues available and related to the selected category.

The user can audition a performer by selecting a cue and observing its action. For example, a Text performer offers a variety of functions associated with a text editor. The Text performer performs these various operations when sent appropriate cues (by the user or another performer). For example, setText 'goodbye' will cause 'goodbye' to be displayed by the Text performer.

A performer learns by demonstration to send another performer a particular cue. The user initiates this by sending a performer a cue indicating that an action is to be defined and telling the system to "watch" the demonstration. A tiny "eye" icon, such as the one in window (6), opens to indicate the system is watching. The user then demonstrates the cues to be sent to other performers by simply selecting those cues from their cue sheets.



We based this figure on Finzer and Gould's "Programming by Rehearsal," in the June 1984 issue of *Byte*.

graphically manipulating them to develop program algorithms. Objects in PT, also called *dynamic icons*, consist of tuples of attribute-value pairs (much like association lists in Lisp) and an iconic display function that creates an object's display image as a function of its attribute-value pairs. For instance, its attributes may determine whether or not an object is shaded, where it is located, or how big it is. The value of an attribute may be another object, and thus objects can be hierarchically structured. A graphical object editor allows construction of new object types.

A procedural language, PT uses a film-making metaphor. Programming requires designing graphical objects and using such objects to demonstrate the workings of algorithms. A *picture* is a collection of graphical objects; a *film* (analogous to a procedure or program) is a sequence of manipulations performed on a picture. A programmer first selects a starting picture, then works through pictorial transformations on that picture. The process is recorded as one or more films.

By collecting object icons into a picture and filming a sequence of manipulations on objects of this picture, the programmer obtains a visual representation of a program's execution that results from defining the program itself. Thus, the way the programmer selects and modifies objects, plus the dynamic representation of an object based on its attribute values, aids the programmer in designing a program's animation simultaneously with its algorithms.

See the sidebar "Filming in PT."

Form-based paradigms. You can think of form-based programming as a generalization of spreadsheet programming. Even though it uses text, in a spreadsheet the relationships between the data and the expression of the computations are represented as part of the form itself, not described textually. Hence, the spreadsheet is naturally visual. It would be hard to imagine a purely textual version of a spreadsheet as natural to use, partly because the visual and interactive aspects of spreadsheets play an important role in allowing nonprocedural programming.

The visual representation of a cell matrix allows the omission of the concepts of variables, declarations, and output formatting. In addition, it contributes to the visual image of a large cell matrix wherein each value is normally computed just once per evaluation, with the order of evaluation derived, not specified. The visual interface with its various operational areas allows a modeless operation. Hence, being visual contributes significantly to the success of spreadsheet languages.

Forms[16] extends the spreadsheet paradigm over a larger problem domain. It relies on a visual representation generalized from the spreadsheet representation to minimize required language concepts. The basic "sheet" in Forms is a form, corresponding to a piece of paper on which you can place cell matrices, called *objects*. A cell expression can reference any cell (or cells) in any object within the containing

form or within other forms, subject only to the restriction that the resulting derived evaluation must not be made to be circular.

Cell matrices are bounded or unbounded. A bounded cell matrix is one of fixed, known dimensions, whereas an unbounded cell matrix has at least one dimension that is unknown during the specification of the form. However, all objects must have their dimensions fixed prior to evaluation. Values for unbounded objects are specified by generic cell specifications stated in terms of the *ij*th cell, combined with specific cell specifications for specific fixed cells. A subform is similar in content to a form, but certain objects will inherit their values as parameters. These objects map onto other objects during evaluation. In addition, the value of one or more objects may be returned.

Forms is a declarative language, in that there is no concept of "state." For each evaluation of a form or subform, each cell is evaluated only once. Cyclic evaluation is not allowed. However, iteration and recursion can be accomplished via repeated invocations of a subform, each creating a new instance of the subform. Hence, the set of all forms and subforms used for a given computation provides a

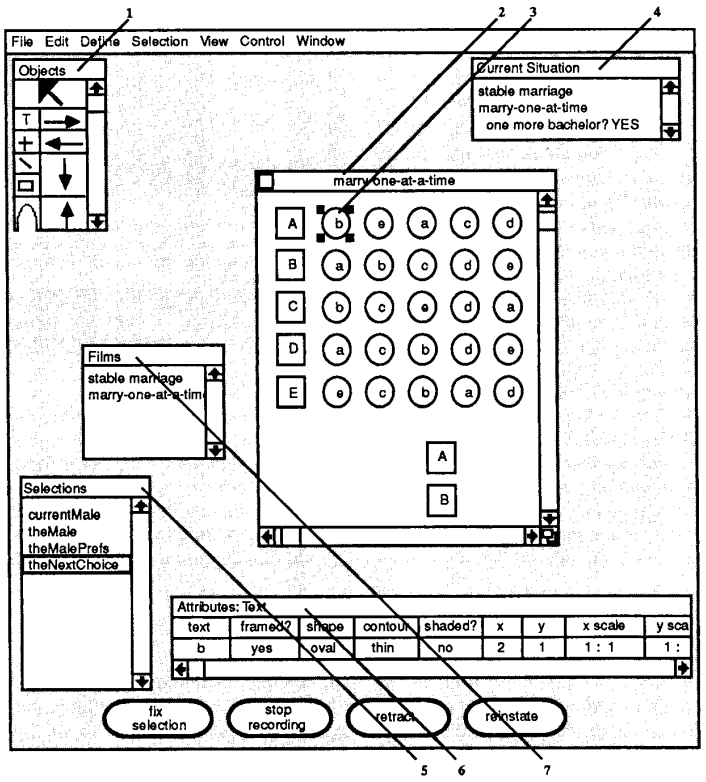
Filming in PT (Pictorial Transformations)

The screen here shows the Pictorial Transformations programming environment during the process of programming a solution to the stable marriage problem. (This problem attempts to match men and women in marriage based on their stated preferences for each other.) In PT, an object is a tuple of attribute-value pairs together with an iconic display function that describes how to display an object based on its attribute-value pairs.

The collection of objects available for use in constructing new objects is displayed in the window at the upper left (1). In this example, several structured objects have already been built. The 6×5 matrix in the center window (2) is a construction of a column of objects, each a row of objects, each a text object. The attribute-value tuple associated with the selected subobject at (3) is displayed in the Attributes window (6).

In this example, an oval shape indicates a female, and a thin contour indicates unmarried. In PT, attributes like contour and shape not only convey information about the visual representation of the algorithm, but also can be tested and used directly in the program solution. Alternatively, these attributes could be named sex or married and might have values such as female and single. An icon has both a logical part and a physical part; hence, the logical values need not be the same as the physical shape.

The iconic display function (not shown) then uses the current set of attribute values to display the object. For instance, the object at (3) is shown to be an unmarried female. When the attribute values of an object change, the object is redisplayed. This has the effect of dynamically animating the execution of a program.



To develop a program, you draw a picture by selecting and placing all objects required in the solution. Series of manipulations to the objects are then recorded as films (analagous to procedures). Current films are listed in the Films window (7). Once initiated, filming proceeds by recording a sequence of selections and actions until terminated.

A selection is an expression that discriminates one or more objects or subobjects of the picture. When completed (fixed), selections can be named for subsequent reuse. The Selection window (5) lists current selections.

Actions transform the values of selected objects. When an action is conditional on the value of a selection, one or more new situations result. During filming, situations are recorded one at a time. Stacked situations are displayed in the Current situation window (4). Since actions may modify an object's attributes, potentially actions might require one or more objects to be redisplayed, thus altering the visual image of the picture. In this way, the picture is animated to follow the execution of the film.

We based this screen on Hsia and Ambler's "Programming through Pictorial Transformations," published in *Proceedings of the 1988 IEEE International Conference on Computer Languages.*

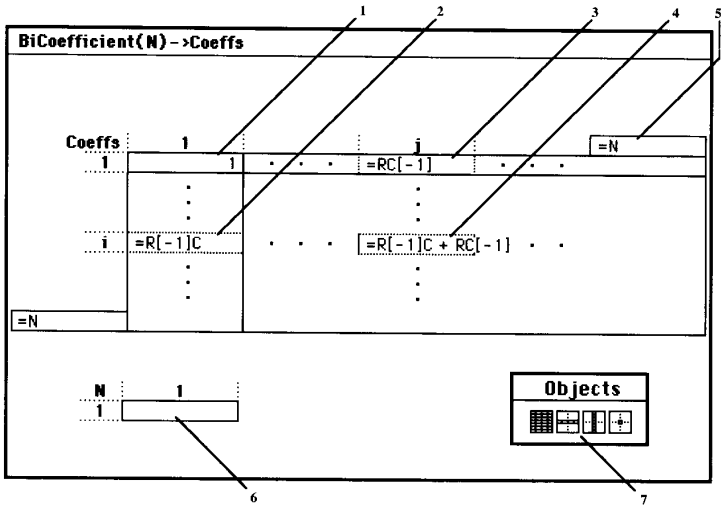
Construction of a subform in Forms

This subform calculates the binomial coefficients for an order *N*-1 equation. Starting from an initially blank form, two objects Coeffs and N are created by selecting icons from the Objects menu (7), dragging them into place, and resizing them as desired. The main cell matrix, named Coeffs, is an unbounded matrix. Both dimensions are unknown and are specified (5) to take the value of the single-cell matrix named N (6) at runtime. Thus, the evaluation of Coeffs will depend on the evaluation of N. This will force N to be evaluated prior to any evaluation of Coeffs.

In this example, the value of N is a parameter, supplied whenever BiCoefficient is instantiated. The computation of BiCoefficient is specified by four expressions (1, 2, 3, 4) that cover the four regions R1C1, R1C*j*, R*i*C1, and R*i*C*j* (for $i,j > 1$). Once the bounds of the matrix are fixed, each cell in each region will be computed using the expression for that region. Cell dependencies will ensure that the matrix will compute correctly starting from R1C1.

When BiCoefficient is to be invoked from some other form, a new instance of BiCoefficient is created when you select it from a list of subforms. For this new instance, the object N, which is expected to inherit its value and was previously blank, is now specified to be the value of a cell in the "calling" form from which we are to get the value of N, the matrix order. This has the effect of fixing the dimensions of BiCoefficient.

The result values are bound similarly by selecting the matrix within the "calling" form that is to receive the result values and specifying that their values are to be the contents of the BiCoefficient matrix.



We based this figure on Ambler's "Forms: Expanding the Visualness of Sheet Languages," published in *Proceedings of the 1987 IEEE Workshop on Visual Languages.*

complete history of the computation. Consequently, it provides a complete trace of the computation and a naturally visual means of debugging.

The sidebar on Forms shows the construction of a subform.

Trend toward naturally visual. It is too early to say which of these visual language approaches will succeed, but clearly they are moving away from the idea of applying visual transformations to textual approaches and toward the idea of naturally visual approaches. This trend shows general agreement with the observation discussed earlier that visual techniques applied to otherwise textual approaches are limited.

In the nearly twenty years since the development of Interlisp, the virtues of a visual, highly integrated language environment have become well accepted. In this article we have looked specifically at the influence of visual technology on three elements of language environments: user interfaces, editors, and programming languages. For each element, we have seen the transition from a strictly textual representation, through relatively straightforward visual representations of otherwise textual technology, and toward new investigations into more naturally visual uses of visual technology. We assert that this trend is also true of other elements associated with language environments, such as debuggers, interpreters, and on-line documentation tools. Perhaps most significantly, visual technology seems to be moving to a convergence between the language itself and the language environment, a convergence that goes beyond the visualization of existing textual approaches, a convergence that is naturally visual.□

Acknowledgments

The authors would like to acknowledge the important contributions made by Phillip G. Bradford, Yen-Teh Hsia, Mike Robinson, and James A. Shelton in the research that led to this article.

References

1. W. Teitelman and L. Masinter, "The Interlisp Programming Environment," *Computer*, Vol. 14, No. 4, Apr. 1981, pp. 25-33.

2. A. Goldberg, *Smalltalk-80: The Interactive Programming Environment*, Addison-Wesley, Reading, Mass., 1984.

3. D.C. Swinehart et al., "A Structural View of the Cedar Programming Environment," *ACM Trans. Programming Languages and Systems*, Vol. 8, No. 4, Oct. 1986, pp. 419-490.

4. S.P. Reiss, "Pecan: Program Development Systems that Support Multiple Views," *IEEE Trans. Software Engineering*, Vol. SE-11, No. 3, Mar. 1985, pp. 276-285.

5. A. Wasserman and P. Pircher, "A Graphical, Extensible Integrated Environment for Software Development," *SIGPlan Notices*, Vol. 22, No. 1, Jan. 1987, pp. 131-142.

6. S.P. Reiss, "Garden Tools: Support for Graphical Programming," in *Advanced Programming Environments*, Lecture Notes in Computer Science #244, R. Conradi, T. Didriksen, and D. Wanvik, eds., Springer-Verlag, N.Y., 1986, pp. 59-72.

7. T. Teitelbaum and T. Reps, "The Cornell Program Synthesizer: A Syntax-Directed Programming Environment," *Comm. ACM*, Vol. 24, No. 9, Sept. 1981, pp. 563-573.

8. A.N. Habermann and D. Notkin, "Gandalf: Software Development Environment," *IEEE Trans. Software Engineering*, Vol. SE-12, No. 12, Dec. 1986, pp. 1,117-1,127.

9. M. Hamilton and S. Zeldin, "Higher Order Software — Methodology for Defining Software," *IEEE Trans. Software Engineering*, Vol. SE-2, No. 1, Mar. 1976, pp. 9-32.

10. M. Moriconi and D.F. Hare, "The PegaSys System: Pictures as Formal Documentation of Large Programs," *ACM Trans. Programming Languages and Systems*, Vol. 8, No. 4, Oct. 1986, pp. 524-546.

11. E.P. Glinert and S.L. Tanimoto, "Pict: An Interactive Graphical Programming Environment," *Computer*, Vol. 17, No. 11, Nov. 1984, pp. 7-25.

12. A. Borning, "Defining Constraints Graphically," *Proc. CHI 86*, Conf. Human Factors in Computing Systems, Apr. 1986, ACM, pp. 137-143.

13. R.V. Rubin, E.J. Golin, and S.P. Reiss, "ThinkPad: A Graphical System for Programming by Demonstration," *IEEE Software*, Vol. 2, No. 2, Mar. 1985, pp. 73-79.

14. W. Finzer and L. Gould, "Programming by Rehearsal," *Byte*, Vol. 9, No. 6, June 1984, pp. 187-210.

15. Y.-T. Hsia and A. Ambler, "Programming through Pictorial Transformations," *Proc. 1988 IEEE Int'l Conf. Computer Languages*, Oct. 1988, CS Press, Los Alamitos, Calif., Order No. FJ874, pp. 10-16.

16. A.L. Ambler, "Forms: Expanding the Visualness of Sheet Languages," *Proc. 1987 Workshop on Visual Languages*, Tryck-Center, Linkoping, Sweden, Aug. 1987, pp. 105-117.

Supplemental readings on visual programming

Due to space limitations, we could not discuss many important visual languages. The following reading list provides sources for additional information.

General surveys

Ambler, A.L., et al., "Integrated Programming Environments: A Survey of Milestones," Univ. of Kansas Computer Science Tech. Report 88-6, Lawrence, Kan., 1988.

Raeder, G., "A Survey of Current Graphical Programming Techniques," *Computer*, Vol. 18, No. 8, Aug. 1985, pp. 11-25.

Shu, N.C., *Visual Programming*, Van Nostrand Reinhold Co., N.Y., 1988.

Visual environments and languages

Halbert, D.C., *Programming by Example*, PhD thesis, Computer Science Div., Dept. of EE and CS, University of California at Berkeley, 1984.

Hirakawa, M., et al., "A Framework for Construction of Icon Systems," *Proc. 1988 IEEE Workshop on Visual Languages*, Oct. 1988, CS Press, Los Alamitos, Calif., Order No. FX876, pp. 70-77.

Kimura, T.D., J.W. Choi, and J.M. Mack, "Show and Tell: A Visual Programming Language," to appear in *Visual Computing Environments*, E.P. Glinert, ed., CS Press, Washington, D.C., 1989.

Kozen, D., et al., "Alex — An Alexical Programming Language," *Proc. 1987 Workshop on Visual Languages*, Tryck-Center, Linkoping, Sweden, Aug. 1987, pp. 315-329.

Ludolph, F., et al., "The Fabrick Programming Environment," *Proc. 1988 IEEE Workshop on Visual Languages*, CS Press, Los Alamitos, Calif., Order No. FX876, Oct. 1988, pp. 222-230.

Moshell, J., et al., "A Spreadsheet-Based Visual Language for Freehand Sketching of Complex Motions," *Proc. 1987 Workshop on Visual Languages*, Tryck-Center, Linkoping, Sweden, Aug. 1987, pp. 94-104.

Pong, M.C., and N. Ng, "PIGS — A System for Programming with Interactive Graphical Support," *Software — Practice and Experience*, Vol. 13, No. 9, Sept. 1983, pp. 847-855.

Smith, D.C., "Pygmalion: A Computer Program to Model and Stimulate Creative Thought," PhD dissertation, Stanford University, Stanford, Calif., 1975.

Smith, D.N., "Visual Programming in the Interface Construction Set," *Proc. 1988 IEEE Workshop on Visual Languages*, CS Press, Los Alamitos, Calif., Order No. FX876, Oct. 1988, pp. 109-120.

Smith, R.B., "The Alternate Reality Kit: An Environment for Creating Interactive Simulations," *Proc. 1986 IEEE Workshop on Visual Languages*, CS Press, Los Alamitos, Calif., Order No. FX722, June 1986, pp. 99-106.

Zloof, M.M., "QBE/OBE: A Language for Office and Business Automation," *Computer*, Vol. 14, No. 5, May 1981, pp. 13-22.

Program animation

Brown, G., et al., "Program Visualization: Graphical Support for Software Development," *Computer*, Vol. 18, No. 8, Aug. 1985, pp. 27-35.

Brown, M., *Algorithm Animation*, published as an ACM distinguished dissertation, MIT Press, Cambridge, Mass., 1987.

Hyrskykari, A., and K. Raiha, "Animation of Algorithms without Programming," *Proc. 1987 Workshop on Visual Languages*, Tryck-Center, Linkoping, Sweden, Aug. 1987, pp. 40-54.

Allen L. Ambler is an associate professor in the Department of Computer Science at the University of Kansas. His research interests include visual programming languages, programming language design, software development environments, and functionally distributed software architectures. He has been a senior architect for Amdahl Corp. and vice president of software development for Dialogic Systems Corp.

Ambler is a reviewer for *IEEE Software* and has served as Executive Committee member-at-large of ACM SIGPlan and as a reviewer for *ACM Computing Reviews*. He is a member of IEEE and ACM.

Ambler received his undergraduate degree in mathematics from the University of Kansas and his MA and PhD in computer science from the University of Wisconsin at Madison.

Margaret M. Burnett is a PhD candidate in computer science at the University of Kansas. Her research interests include visual programming, software development environments, and computer-human interaction. She has been a lecturer for the University of Kansas, a consultant and systems analyst, and a systems engineer for Procter and Gamble.

Burnett holds an MS in computer science from the University of Kansas and a BA in mathematics from Miami University of Ohio.

Readers may contact the authors at 110 Strong Hall, University of Kansas, Lawrence, KS 66045.