AN ABSTRACT OF THE THESIS OF

Sherry Yang for the degree of Doctor of Philosophy in Computer Science presented on November 4, 1996. Title: Generalizing Abstractions in Form-Based Visual Programming Languages: From Direct Manipulation to Static Representation.

Abstract approved: Redacted for Privacy

Margaret M. Burnett

We believe concreteness, direct manipulation and responsiveness in a visual programming language increase its usefulness. However, these characteristics present a challenge in generalizing programs for reuse, especially when concrete examples are used as one way of achieving concreteness. In this thesis, we present a technique to solve this problem by deriving generality automatically through the analysis of logical relationships among concrete program entities from the perspective of a particular computational goal. Use of this technique allows a fully general form-based program with reusable abstractions to be derived from one that was specified in terms of concrete examples and direct manipulation.

Also addressed in this thesis is how to statically represent the generalized programs. In general, we address how to design better static representations. A weakness of many interactive visual programming languages is their static representations. Lack of an adequate static representation places a heavy cognitive burden on a VPL's programmers, because they must remember potentially long dynamic sequences of screen displays in order to understand a previously-written program. However, although this problem is widely acknowledged, research on how to design better static representations for interactive VPLs is still in its infancy.

Building upon the cognitive dimensions developed for programming languages by cognitive psychologists Green and others, we have developed a set of concrete benchmarks for VPL designers to use when designing new static representations. These benchmarks provide design-time information that can be used to improve a VPL's static representation.

© Copyright by Sherry Yang
November 4, 1996
All Rights Reserved

Generalizing Abstractions in Form-Based Visual Programming Languages:

From Direct Manipulation to Static Representation

by

Sherry Yang

A THESIS

submitted to

Oregon State University

in partial fulfillment of
the requirements for the
degree of

Doctor of Philosophy

Completed November 4, 1996
Commencement June 1997

Doctor of Philosophy thesis of Sherry Yang presented on November 4, 1996

APPROVED:

Redacted for Privacy

Major Professor, representing Computer Science

Redacted for Privacy

Head of Department of Computer Science

Redacted for Privacy

Dean of Graduate School

I understand that my thesis will become part of the permanent collection of Oregon State University libraries. My signature below authorizes release of my thesis to any reader upon request.

Redacted for Privacy

Sherry Yang, Author

ACKNOWLEDGMENTS

First of all, I would like to express my gratitude to my advisor, Dr. Margaret Burnett, whose guidance and influence during this project was invaluable. I have benefited greatly from her expertise and experience. I especially appreciate her concern for her students. I am thankful for all the time and energy that she devoted toward improving my research and writing skills, not to mention the countless hours she spent reading and critiquing this document. Without her endless patience, understanding and encouragement, it would not have been possible for me to complete this degree. It was my privilege to work with such an excellent teacher and mentor. Thank you!

I am indebted to Dr. Carol Brown for her friendship, continuous help and support. Her confidence in my abilities made all the difference in difficult times.

I thank Drs. Curtis Cook, Bruce D'Ambrosio and Robert Jarvis for serving on my committee and for their helpful comments. Your support is appreciated.

I am grateful for Dr. Donna Cruse who can be approached with any problem whatsoever. She is never too busy to listen and offer assistance. Thank you for being such a great teacher and friend!

I thank Dr. Moshé Zloof and Hewlett Packard Labs for supporting this research. I thank Moshé and Elyon DeKoven for their help with the Representation Design Benchmarks. Thanks also to Carole Gize for her assistance with this research collaboration.

A very special thank you to members of the Forms/3 research group - John Atwood, Rebecca Walpole, Paul Carlson, Herky Gottfried, Piet van Zee, Marla Baker, Judy Hays, Eric Wilcox and JJ Cadiz for their help with the implementation as well as their helpful suggestions and comments.

I thank Bernie Feyerherm, Clara Knutson and Sheryl Parker in the Computer Science office for all their help during my stay at OSU. I have enjoyed our friendship. Thanks for knowing just how to cheer me up.

I thank these faithful friends for their encouragement over the years: Paula Hannan, Cathy Delph, Mu-Hong Lim, Nabil Zamel, Kim Drongesen, Becky Brown, Karl Schricker, Chia-Chi Chou, Virginia Li, Rick Wodtli, Carisa Bohus, Rosie Saraga, Tamsen Fuller, Pat Ingram, Lois Van Leer, Karuna Neustadt, Nancy Haldeman and Lorene Hales. I especially thank Shan Luh for being there when I needed help the most. I am grateful for her love and support.

Most of all, I want to thank my family for their love, encouragement and patience throughout this process.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF FIGURES (Continued)

LIST OF FIGURES (Continued)

LIST OF TABLES

GENERALIZING ABSTRACTIONS IN FORM-BASED VISUAL PROGRAMMING LANGUAGES: FROM DIRECT MANIPULATION TO STATIC REPRESENTATION

1. INTRODUCTION

We believe that concreteness, direct manipulation and responsiveness are among the most important advantages of working in a visual programming language (VPL). Toward this end, visual programming languages such as Fabrik [Ingalls et al. 1988], VPL [Lau-Kee et al. 1991], VIVA [Tanimoto 1990] and Forms/3 [Burnett 1991; Burnett and Ambler 1994] allow programmers to program very concretely, and receive continuous visual feedback throughout the process. Although they use flexibility, direct manipulation and prototypical values extensively during development, they do so with the expectation that the program they enter in such a concrete fashion will work the same way for any future values that might someday replace the prototypical values. The problem that we address in this thesis is how to generalize such concrete programs in a particular class of VPLs so that this expectation of generality can be fulfilled.

The problem is divided into two components. The first component has to do with programmer input. It deals with how to derive the generalized version of a program from the concrete version input by the programmer. The second component deals with how to represent the generalized program back to the programmer. The second component also requires development of a method for measuring the attributes of the output schemes.

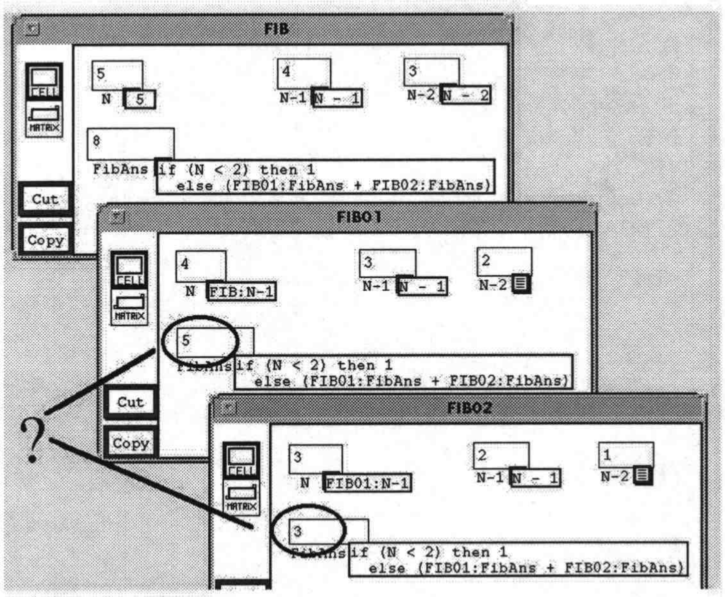
The class of VPLs that we are addressing in this thesis are form-based languages. In form-based VPLs, a program consists entirely of cells, each of which has a formula. Formulas do not contain circular cell references. Spreadsheets are an example of these form-based VPLs. Table 1-1 lists a set of terms relating to form-based VPLs that we will be using in the thesis and their meanings. We believe for a form-based VPL to scale up, that is, to support large and realistic programs instead of toy programs, the next generation of form-based VPLs should include at least some of the following features: procedural abstraction, data abstraction, cell aggregates, and an explicit approach to time. Generalized abstractions provide the foundation that will allow future VPLs to incorporate these features.

Term	Meaning
Cell	Program element, each with a formula and attributes
Form	Collection of one of more cells
Model	Form from which other forms maybe devised
New Instantiation	A form that inherits the model form's cells and formulas.
Cell Aggregates	Grouping of one or more cells with similar attributes and formulas.

Table 1-1. List of terms that we will use relating to form-based VPLs.

1.1. An Example: Deriving The Generalized Programs From Concrete Examples

We will begin with an example to illustrate the problem that we are addressing. Since our goal is aiming at the future form-based VPLs that support features such as procedural abstractions, the problem of deriving the generalized program is more complicated than just describing the physical relationship between cells in a spreadsheet. Rather, it is the logical relationships between forms and cells that define the generalized meaning of the program. The Forms/3 program shown in Figure 1-1 illustrates this logical relationship. The FIB program computes the n'th element of the Fibonacci sequence, which is the sum of the n-1'st and n-2'nd Fibonacci numbers. The program consists of three windows or forms. The prototypical formula "5" has been specified for cell N on form FIB so that the user can receive concrete feedback. The program involves three forms: one to compute the Fibonacci number for the desired N and two more to calculate N-1'st and N-2'nd Fibonacci numbers. We term the original FIB the *model,* and FIB01 and FIB02 *new instantiations* of FIB. Instantiations inherit their model's cells and formulas, unless the programmer explicitly provides a different formula for a cell on an instantiation. Changes in the model are propagated to instantiations.



```
FIB
5      N       5
4      N-1     N - 1
3      N-2     N - 2
8      FibAns  if (N < 2) then 1
                  else (FIB01:FibAns + FIB02:FibAns)

FIB01
4      N       FIB:N-1
3      N-1     N - 1
2      N-2
5      FibAns  if (N < 2) then 1
                  else (FIB01:FibAns + FIB02:FibAns)

FIB02
3      N       FIB01:N-1
2      N-1     N - 1
1      N-2
3      FibAns  if (N < 2) then 1
                  else (FIB01:FibAns + FIB02:FibAns)
```

Figure 1-1. A Forms/3 program to compute the Fibonacci numbers. Several of the concrete formulas are shown. The question mark points to values that cannot be calculated from such concrete formulas–they must be generalized first.

This version of the Fibonacci program is recursive. Forms FIB01 and FIB02 provide a concrete example of one level of invocation of FIB. The problem is how to enable the system to automatically invoke all the necessary FIB calls and to display the answer (8) as soon as the formula for FIB's FibAns is entered. The problem lies in concreteness. As entered by the programmer, the recursive part of the formula for FIB's FibAns is the sum of the FibAns cells on FIB01 and FIB02. This is too concrete–without generalization, all future instantiations of FIB, regardless of how their N cells are changed, will sum the specific FibAns cells on FIB01 and FIB02 (which compute the fourth and third Fibonacci numbers). To solve this problem, the system must recognize and record the logical relationship between FIB and its instantiations from the perspective of computing FibAns, instead of recording the concrete program exactly as it was entered.

1.2. Factors Involved In The Problem

Our goal is to provide automatic generalization without burdening the user with extra work. We believe the following characteristics are desired properties of form-based VPLs that meet our goal. However, these characteristics can interfere both with generalization and with immediate feedback and must be considered before devising the solution.

Modelessness: The first is the orderless nature of the input syntax. To encourage the programmer to concentrate on problem-solving rather than the computer's requirements, the programmer is entirely unrestricted in how she enters a program. Input is modeless, and formulas can be entered in any order. All cell references can be made by pointing at a cell, with no distinction in the way global references are made from the way parameter-passing is accomplished or from the way return values are referenced. To the programmer this means freedom to concentrate on the problem, but to the system it means lack of information. For example, in the Fibonacci example, the programmer may enter the formula for FibAns before providing information that there will be an input parameter N. In this case, the reference to FIB01's FibAns appears to be a global (absolute) reference when in fact it is intended as the result of a parameterized subroutine-like call.

Flexibility: Second, there are few restrictions on the patterns of references that can be made. This allows the programmers to modularize and re-use calculations flexibly including ways that are not supported in many programming languages. Thus, familiar program structures such as global references and subroutine-like uses of forms are possible, but less traditional referencing patterns such as mutually dependent modules (forms) and pipeline-like referencings are also possible. With this flexibility, a program's structure is hard to predict because it will not always fall into traditional patterns.

Avoid Circularity: Third, because of the support for recursion, a form must be able to be re-used before its definition is completed. This can cause circular dependencies to be generated by the instantiation, as in the example above, making immediate feedback impossible until the relationships behind the concrete forms are analyzed to derive a generalized version of the recursive form.

The generalization technique that we have developed uses deductive analysis to derive a generalized program from a concrete one. Generalization is accomplished through the analysis of logical relationships among concrete program entities from the perspective of a particular computational goal. Because it does not use guessing[1], there is no risk of "guessing wrong" and there is no need for additional dialog with the user. This technique supports a modeless direct manipulation interface, and it is flexible enough to handle all possible referencing and calling patterns, including some not commonly found in traditional programming languages. This allows the user to program concretely and flexibly, without unnecessary rules and restrictions about what computations can be used to assist other computations.

1.3. Static Representation And Representation Of Generalized Programs

We believe the static representation of a program is important in a VPL. Lack of an adequate static representation places a heavy cognitive burden on a VPL's programmers, because they must remember potentially long dynamic sequences of screen displays in order to understand a previously-written program.

Many visual programming languages are highly interactive. In such languages, the process of creating a program is often incremental, with many opportunities for interactive visual feedback along the way. We can place an object on the screen and experiment with its effects on other objects, peer into the components of an object by clicking on it, and watch its dynamic behavior simply by observing the changes that occur on the screen as a snippet of the program executes. Such dynamic visual feedback integrates support for

[1]In the field of automatic reasoning, the term inference refers to drawing a conclusion, certain or otherwise. Inference includes guessing and sound deduction. Our system uses sound deduction, and not guessing, thus it is less error-prone.

rapid program construction with continuous debugging, a feature that provides many advantages.

But after the program has been so constructed, the maintenance phase begins. Someone—probably someone different from the original programmer—must understand the previously-written program to be able to modify it. Understanding a previously-written program involves tasks that are not as common in creating a new program, because the maintenance process does not provide the contextual information that is inherent in the creation process. For example, the maintenance programmer will need to learn the overall structure of the program, will need to search for and identify the relevant section of the program without necessarily having seen it before, and will be trying to figure out what other pieces of the program exist that might be affected by the changes.

Although dynamic mechanisms can be very helpful during program creation and debugging, tasks such as those listed in the previous paragraph beg for a static view of the program—one that allows the programmer to study the logic and relationships within the program without the heavy cognitive burden of remembering fine-grained dynamic sequences of visual activity to obtain the needed information. Unfortunately, however, lack of adequate static representations has long been a weakness of interactive VPLs. Numerous research descriptions, taxonomies, and analyses have counted static representation as an important, largely unsolved, issue for many VPLs ([Myers 1990; Cypher et al. 1993; Burnett et al. 1995]). Although lack of adequate static representation in VPLs is widely acknowledged, research on how to design better static representations for interactive VPLs is still in its infancy.

We needed a representation of the generalized programs described in the previous section. The representation of generalized programs must contain complete generalized information so the programmer can understand how the abstraction was constructed. The representation must be static, i.e. lay rest on the screen so the programmers don't have to remember the sequences of screen displays in order to understand the program. There were no techniques specifically for VPL designers to design and evaluate static representations of VPLs. Motivated by the need for such a technique, we have developed *Representation Design Benchmarks*, a flexible set of measurement procedures for VPL designers to use when designing new static representations for their languages. The benchmarks focus exclusively on the static representation part of a VPL, and provide a designer with a yardstick for measuring how well a particular design fulfills design goals related to the static representation's usefulness to programmers.

1.4. Organization Of This Thesis

We begin with some background information and related work in Chapter 2. We describe the generalization algorithm in Chapter 3. In Chapter 4, we present the Representation Design Benchmarks. Applications of the benchmarks are also included in Chapter 4. We describe future work in Chapter 5 and conclude in Chapter 6.

2. BACKGROUND AND RELATED WORK

We have developed the generalization algorithm for the class of form-based VPLs, and have prototyped and evaluated our approach in one such language, Forms/3. The generalization algorithm enables Forms/3 to support procedural abstraction, data abstraction and generalized aggregation. To provide context for our discussion, we will first give a brief overview of Forms/3.

2.1. A Brief Introduction To Forms/3

Forms/3 is a declarative, form-based visual programming language. Declarative VPLs are VPLs in which the relationships among data are specified, as opposed to non-declarative VPLs, where state modification and control flow are explicitly specified by the programmer

We have chosen to prototype our approach in Forms/3 [Burnett 1991; Burnett and Ambler 1994] because it is a full-featured visual programming language in the form-based class of VPLs. Language issues such as event handling, information hiding, scope are supported in Forms/3. Data and procedural abstraction are already partially supported, but needed the generalization algorithm to be complete. Issues relevant to our research, user input and representation, had not been explored. Forms/3 also did not have a graphical user interface, so we first implemented an user interface for Forms/3 before examining the research issues.

Like spreadsheets, programs in Forms/3 are defined via *cells* on *forms*. Programming in Forms/3 consists of arranging and defining one or more forms. Forms contain cells. Each cell has a *formula* which defines its value. The computational dependencies between cells constrains evaluation ordering. The evaluation model used is lazy evaluation: a value is calculated only when it is required and the same evaluation will not be done again. Unlike traditional spreadsheets, cells in Forms/3 can be grouped into *matrices* and *abstraction boxes*.

Another extension not commonly found in traditional spreadsheets, the formula for a cell in Forms/3 actually defines a vector of values along a time dimension, rather than an atomic value. The idea of temporal assignment is that any given cell can define a single value or a sequence of values. If a cell's formula defines a sequence, each new computation in the sequence will also cause a cascade of further evaluation (if demanded) starting with those cells dependent on the original cell, and computing a new value for each dependent cell. This time-indexed model then produces for a given cell only a single value within each time interval, but over time it produces a vector of values called a *temporal vector*. Examples of time-based formulas are:

earlier X: At time t, results in the value of cell X at time t-1

x *fby* A: Specifies a sequence of values starting with x and followed by the values computed by A.

Figure 2-1 shows a simple program to compute the area of a square. There are two cells in the program, one for the length of one of the sides of a square and one for the area.

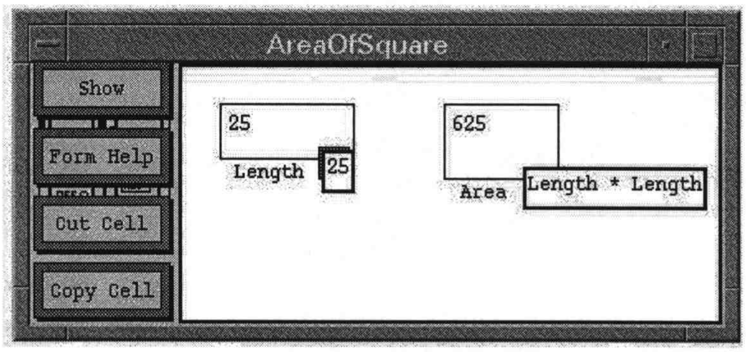
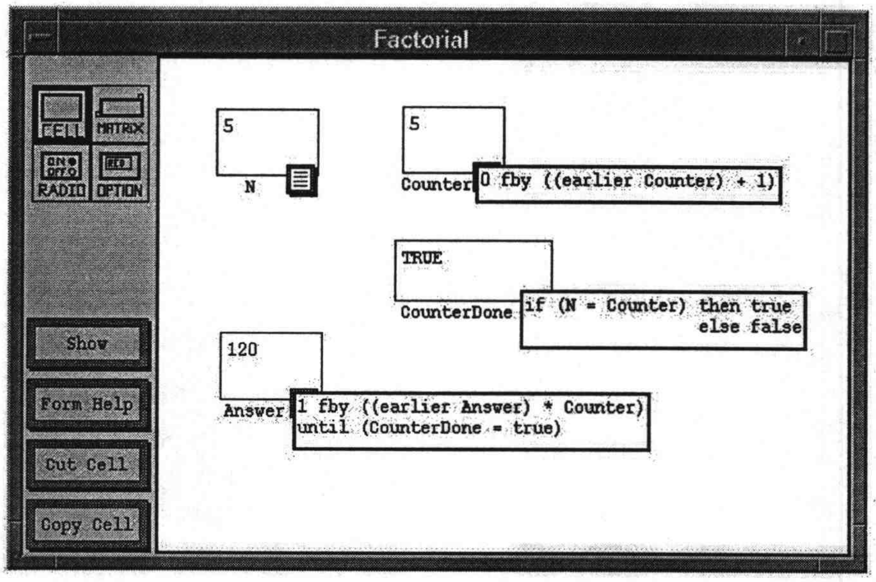


Figure 2-1. A Forms/3 program to compute the area of a square.

In Forms/3, as soon as a programmer enters a formula, it is immediately evaluated and the result displayed, as in a spreadsheet. There is no compile phase, no need to click on individual cells to see their values. Each addition or change to a program is immediately reflected on the screen. Important aspects of this spreadsheet-like approach are that the feedback is immediate, incremental, and automatic, imposing no effort on the programmer. In Forms/3, however, unlike spreadsheets, the source code (formulas) and accompanying values can be shown together.

Figure 2-2 shows a Forms/3 program to compute the Factorial function using a time-based approach. The Counter starts at 0 and at each time step, a new Answer is computed until Counter reaches N. Figure 2-3 shows the values of cells at each time step along the time dimension.



```
N            5
Counter      5      0 fby ((earlier Counter) + 1)
CounterDone  TRUE   if (N = Counter) then true
                                      else false
Answer       120    1 fby ((earlier Answer) * Counter)
                    until (CounterDone = true)
```

Figure 2-2. A time-based Forms/3 program to compute the Factorial function.

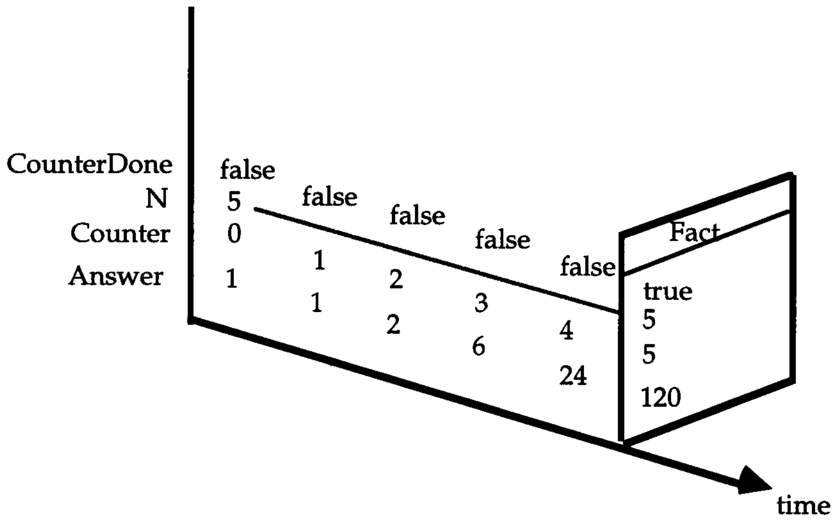


Figure 2-3. Values of cells along the time dimension.

2.2. Related Work: Deriving Generality From The Input

We will survey some work relating to driving the generality of the program from user input in this section. We will first look at some by-demonstration systems followed by form-based visual languages.

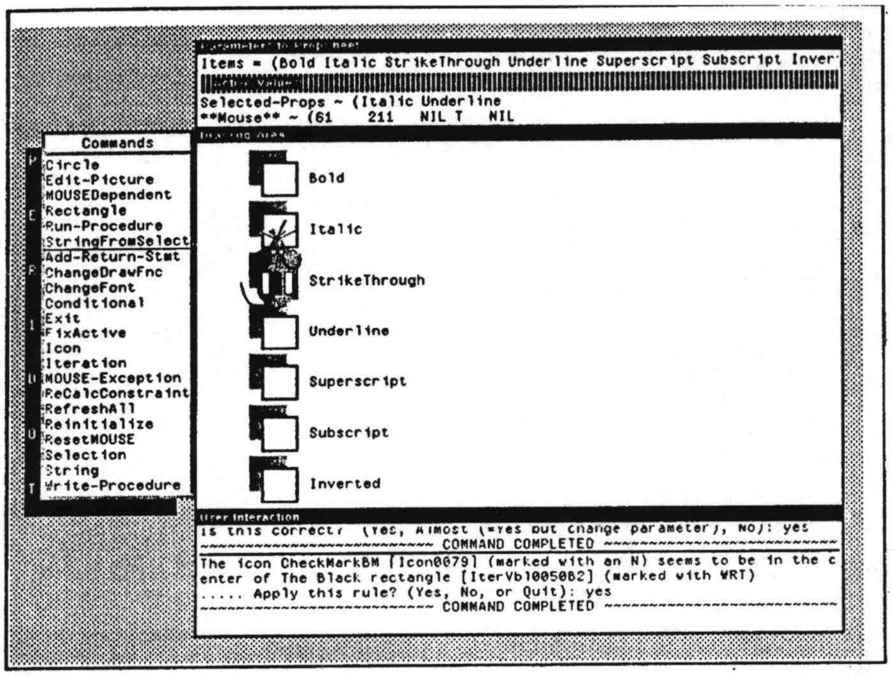
2.2.1. By-Demonstration Systems

Although Forms/3 does not use programming by example or programming by demonstration, because of its extensive use of prototypical values for concreteness and direct manipulation, Forms/3 shares with that family of VPLs some of the same difficulties in determining the generality intended by concrete prototypical values.

Many by-demonstration systems use inference to solve this problem. Inference is most effective in a limited problem domain; for example, inference has been used successfully in the by-demonstration system Peridot [Myers 1993], which is a language specifically for user-interface specification, and in Chimera [Kurlander and Feiner 1992], which is a language for graphical editing and user interface editing. Peridot and Chimera are representatives of by-demonstrational systems such as Turvy [Maulsby 1993] and Mondrian [Lieberman 1993] that use inference to perform generalization.

In Peridot, the user interface designer draws a picture of what the user interface should look like using a drawing package, and uses direct manipulation to demonstrate how the user interface should work. Based on what the designer demonstrates, Peridot creates code for the interface and its connection to actual application programs. The code produced is not simply a transcript of the designer's actions, however, because this would not provide sufficient functionality. For example, a pop-up menu might be designed with a particular set of strings, but the same code should work for any list of strings. Therefore, Peridot allows the designers to explicitly specify parameters in the code produced, where parts of the interface can depend on the values of the parameters. Peridot uses inference to decide how the graphics and mouse should change based on the actual values for the parameters.

Figure 2-4 shows a property sheet being created using Peridot. The designer specifies that the name of this procedure *PropSheet* and specifies one parameter *Items*. The items list is the labels of check boxes in the property sheet and is typed in by the designer. The check boxes are created using the drawing tool. The designer first draws one check box and gives it the label "Bold". She then makes a copy of the check box and gives the second check box the label "Italic". Peridot notices that Bold and Italic are part of an iteration over the "Items" parameter and asks the designer if she wishes to iterate over the entire list. If the designer answers "yes", then check boxes are automatically generated for rest of the items in the parameter list. When a check box is selected, a check mark is placed in the box. This behavior is demonstrated using the simulated mouse icon. When all the appearance and behaviors of the property sheet is completed, the PropSheet procedure is written and it can be called with any list of items to be displayed.



```
Items = (Bold Italic StrikeThrough Underline Superscript Subscript Inver
Selected-Props ~ (Italic Underline
**Mouse** ~ (61    211   NIL T   NIL
```

```
Is this correct?  (Yes, Almost (=Yes but change parameter), No): yes
~~~~~~~~~~~~~~~~~~~~~~~~ COMMAND COMPLETED ~~~~~~~~~~~~~~~~~~~~~~~~
The icon CheckMarkBM [Icon0079] (marked with an N) seems to be in the c
enter of The Black rectangle [IterVb1005082] (marked with WRT)
..... Apply this rule? (Yes, No, or Quit): yes
~~~~~~~~~~~~~~~~~~~~~~~~ COMMAND COMPLETED ~~~~~~~~~~~~~~~~~~~~~~~~
```

Figure 2-4. A property sheet being created using Peridot. The label of check list items are based on the items parameter on the top of the window. The appearance and behavior of a check list item is demonstrated and automatically generalized for all items in the list. A check mark is placed to show which items are selected. A simulated mouse icon is used to demonstrate that items should be selectable by the mouse.

Chimera is a tool for graphical and interface editing and uses inference to produce reusable editing commands. Chimera allows the user to edit 2D graphics, user interfaces and text. In Chimera, editing command histories are represented visually using a comic strip metaphor. Actions in the history of the editing are distributed over a sequence of panels. Related commands are coalesced into single panels by pattern matching rules showing logical commands instead of physical commands. Figure 2-5 shows commands that have added a text label to the oval and constructed a drop shadow for the oval. The commands are demonstrated by direct manipulation in the top drawing window and the history window depicts the sequence of logical commands, coalesced from one or more physical commands. The first panel, for instance, shows two commands that move the caret (cursor) to the desired position for the text, and then insert the characters. The second panel shows commands to select the text and change the font.

Some commands can be reused for other graphical objects by turning these command histories into a macro. Any set of panels in the history window can be turned into macros by selecting the panels and providing the arguments to the macro. For example, Figure 2-6 shows a macro builder window that contains commands to add a drop shadow to an object. The first argument to the macro is the object for which the shadow is generated. This is specified by the designer selecting the graphical object in the first panel and given a name *object*. Graphical properties, such as color can also be arguments, as shown in the second panel. Commands in the macro then need to be generalized to work in new contexts.

In Chimera, generalization can be either specified by the user explicitly, or inferred by the system. Each command is supplied with a set of different interpretations by the system, as well as heuristics for distinguishing when each interpretation is likely. When generalization of a command is invoked, the system evaluates the heuristics in the context of the graphics state to produce an ordered list of possible intents. The user can view the system's generalizations and can override them. For example, to generalize the drag command, the last panel in Figure 2-6 is selected and invoke the Generalize-Panel command. The window shown in Figure 2-7 appears, containing various generalizations that the system considered plausible in the given context, with the most likely interpretation selected. Once a macro has been generalized, it can be named, saved and invoked by providing the actual arguments.

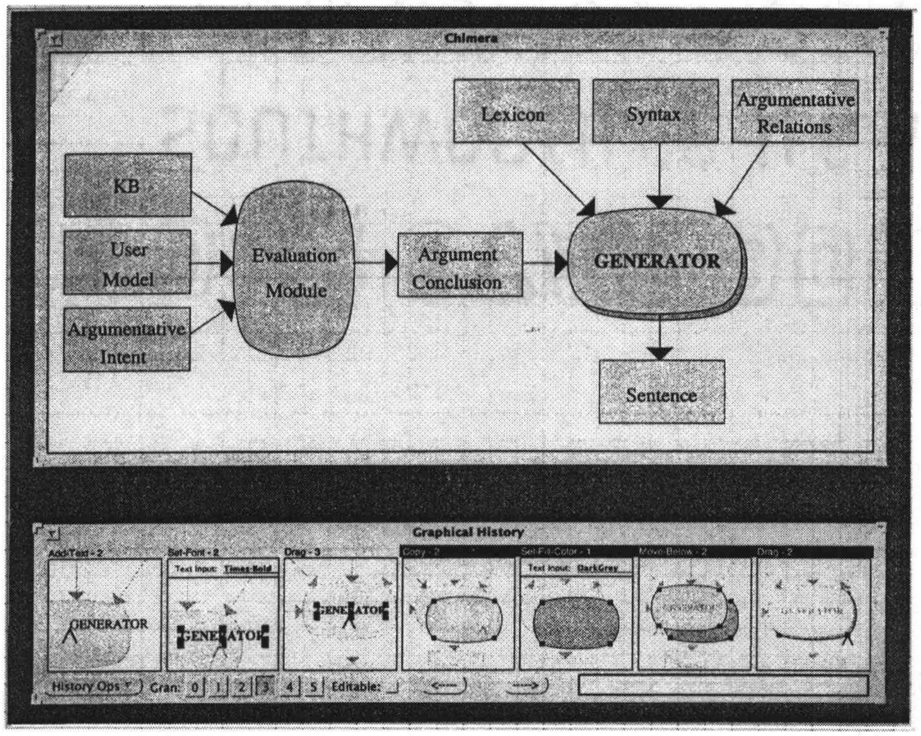


Figure 2-5. A Chimera editor window (top), and its graphical history window (bottom) showing steps that added text to an oval and created a drop shadow in the scene above.

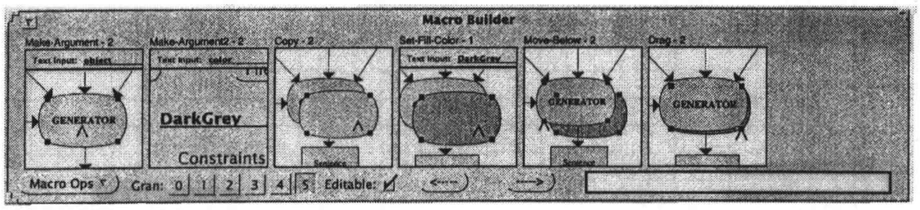
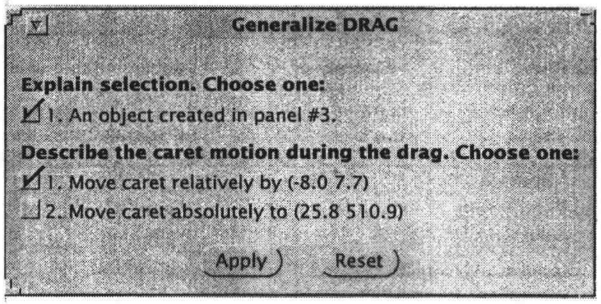


Figure 2-6. Macro Builder window containing operations to add a drop shadow to an object.


Generalize DRAG

Explain selection. Choose one:

[✓ 1. An object created in panel #3.]

Describe the caret motion during the drag. Choose one:

[✓ 1. Move caret relatively by (-8.0 7.7)]

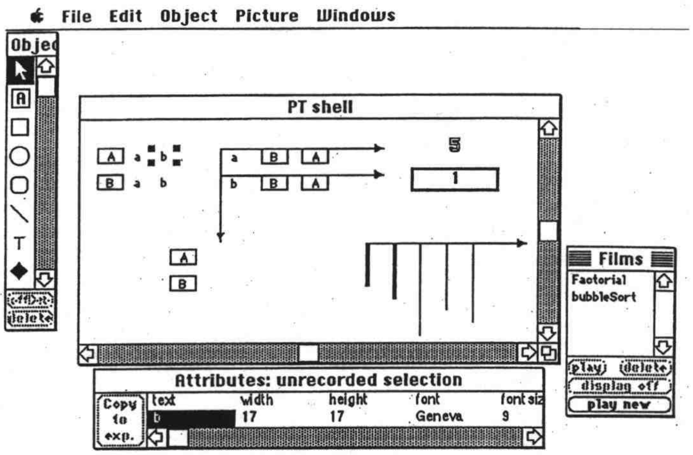
2. Move caret absolutely to (25.8 510.9)

Apply Reset

Figure 2-7. A window showing system's generalizations for the last panel of Figure 2-6.

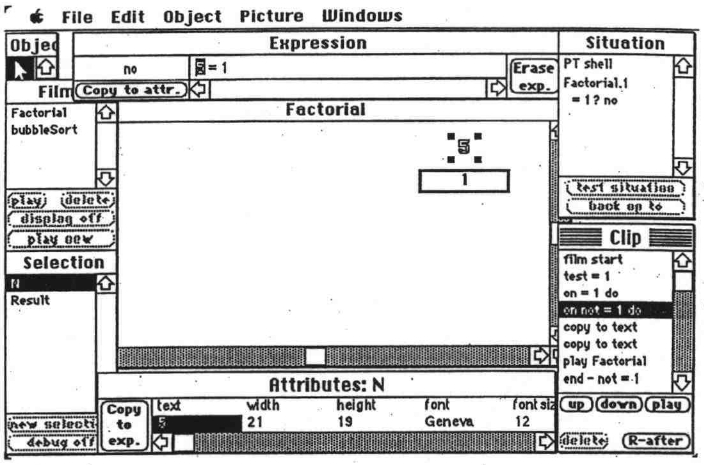
PT [Ambler and Hsia 1993] is a representative of by-demonstration VPLs, such as Pygmalion [Smith 1993] and Tinker [Lieberman 1993], that does not use inference. PT is a general-purpose VPL. Programming in PT requires designing graphical objects, placing them into a context, called a picture, and manipulating these objects graphically (Figure 2-8). The process of manipulating a picture is recorded as a film (Figure 2-9). A film corresponds to a procedure and a picture to the combined collection of parameters and local variables used by the procedure. Graphical objects in PT have an attribute-value list and a special procedure, called a display function. The display function is automatically invoked whenever the object is displayed and computes a graphical image of the object. When an object is selected by pointing and clicking on the object in the picture, its associated attribute values are displayed in the attribute window. This allows the programmer to change the attribute values which in turn may affect the visual appearance of the object. During filming or picture manipulation of objects, three types of manipulations are recorded: situation testing, action, and selection. A situation is a condition that is employed to condition action sequences. An action may be primitive actions like assignment of a value or the creation of an object, or an action may require playing an already recorded film. A selection identifies a set of target objects that may be used in other selections and in specifying object modifications, such as changing their attribute values. While the selection appears concrete in terms of sample data, any manipulation of the selected concrete data is abstracted as a manipulation of all objects satisfying the selection criteria. The feature of PT that makes this generalization possible without inference is the fact that generalized information is explicitly specified by the programmer as part of programming. For example, it is possible in PT to select an object by pointing at it, and to inform the system (using direct manipulation and formula-like operations) what attribute of the object caused it to be selected. An example of such an attribute might be the minimum-valued object in the selected group.

PT's generalization is similar to Forms/3 in that the logic part of formulas are general and immediate feedback are provided via examples. Unlike PT, however, where the programmer explicitly identifies the values that are examples versus those intended as constants, Forms/3 automatically sorts out the logical relationships and deduces their generalities from the concrete objects (cells) in the program without the programmer having to help in this process.



text	width	height	font	font siz
b	17	17	Geneva	9

Figure 2-8. *PT's drawing environment.*



text	width	height	font	font siz
5	21	19	Geneva	12

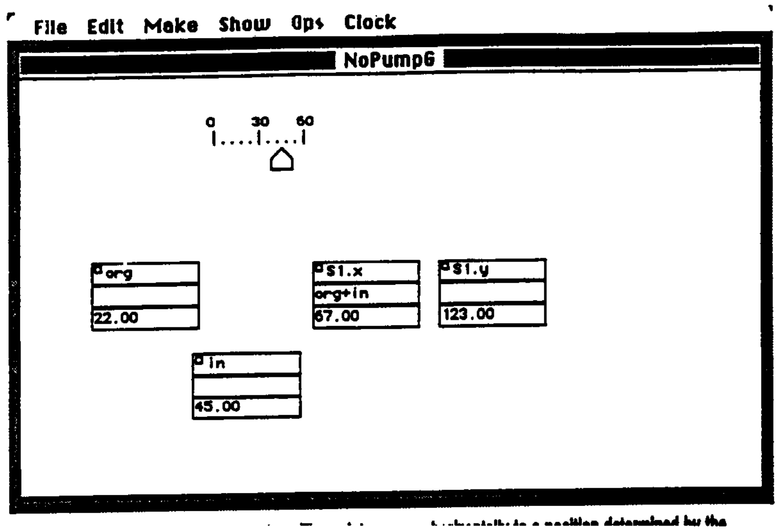
Figure 2-9. *PT's filming environment.*

2.2.2. Form-Based Languages

The distinction between concrete and generalized versions of a form-based program is similar to the difference between absolute and relative references found in traditional spreadsheets, which are among the earliest examples of the form-based approach to programming. However, in spreadsheets the generalization problem is binary; either a cell reference is absolute, or it is defined by a specific physical relationship that can be expressed by an integer offset. In Forms/3 the generalization problem is deriving reusable abstractions, which requires detecting logical relationships among cells that reference each other. For example, in Forms/3 if cell A references cell B on a different form, A could be similar to a formal parameter with B as the actual parameter, or B could be similar to a "return value" from another form's calculations, or B could be filling a role similar to a global constant.

Other form-based systems similar to Forms/3 either do not support generalized abstractions or the generalized information is explicitly provided by the user. For example, the form-based systems NoPumpG [Lewis 1990] and NoPumpII [Wilde and Lewis 1990] are extensions to spreadsheets that support interactive graphics. NoPumpG allows the user to create graphical primitives, and to define the behavior of those primitives with cells. Figure 2-10 shows an example of cells in NoPumpG. Each cell has a name, an optional formula and a value. Formulas are entered by clicking on the formula portion of a cell, selecting an operator from a pull-down menu, and by clicking on one or more other cells to act as operands. Graphical objects are created using a simple bitmap editor and fixed text strings. When objects are created, control cells for them are created and automatically displayed. Control cells can be modified in the same way as user-created cells. A line, for instance, is associated with four control cells, one for each coordinate of each end. Figure 2-10 also shows how graphical output from spreadsheet is done. The pointer is a sketch and cells S1.x and S1.y contain and control its position. The cell S1.x which controls the horizontal position of the pointer, contains a formula making its value the sum of the values of cells "in" and "org". The cell "in" holds the value to be displayed, and the cell "org" sets the origin of the pointer. The cell S1.y contains no formula since the pointer is intended to move horizontally. Whenever the value of "in" is changed, the pointer moves to the appropriate position.

Unlike traditional spreadsheets, the physical relationships of cells play no role in determining a program's meaning. There is also no facility for generalized abstractions based on logical relationships, because the NoPump systems are not intended to support general-purpose programming.



org
22.00

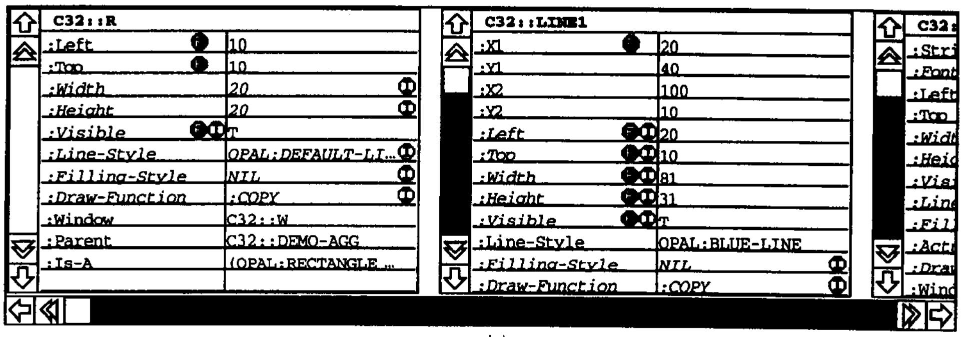
S1.x
org+in
67.00

S1.y
123.00

in
45.00

Figure 2-10. Horizontal slider device created with NoPumpG. The pointer moves horizontally to a position determined by the cell "in".

C32 [Myers 1991] is another form-based system that has facility for producing generalized abstractions. C32 is part of the Garnet user interface development system. It uses a spreadsheet model to allow users to construct one-way constraints, which are relationships among graphical objects. Figure 2-11 shows an example of C32. Each column contains a separate object. Rows are labeled with the names of the slots, such as :top, :left, :x1, :y1, etc. The spreadsheet cells show the current values of the slots. If a value changes, then the display will be immediately updated. If the user edits the value in the spreadsheet cell, the object's slot will be updated. The "F" icon by some slots in Figure 2-11 means the slot value is computed from a formula. Pressing the mouse on the icon causes the constraint expression to appear in a different window allowing the formula to be edited.



C32::R	
:Left	10
:Top	10
:Width	20
:Height	20
:Visible	T
:Line-Style	OPAL:DEFAULT-LI...
:Filling-Style	NIL
:Draw-Function	:COPY
:Window	C32::W
:Parent	C32::DEMO-AGG
:Is-A	(OPAL:RECTANGLE ...

C32::LINE1	
:X1	20
:Y1	40
:X2	100
:Y2	10
:Left	20
:Top	10
:Width	81
:Height	31
:Visible	T
:Line-Style	OPAL:BLUE-LINE
:Filling-Style	NIL
:Draw-Function	:COPY

C32:
:Stri
:Font
:Left
:Top
:Widt
:Heig
:Visi
:Line
:Fil
:Act
:Dra
:Wind

Figure 2-11. A C32 window showing objects and their slot values.

Object references can be specified in several ways in C32. As in a spreadsheet, the user can point to a slot and have a reference to that slot inserted into the formula. Objects can be read into a C32 spreadsheet column and any of its slots can be referenced by other objects. Alternatively, objects can also be referenced by just clicking on the graphical objects in the Garnet window. C32 uses inference to guess which slot of the graphical object the user wishes to specify. As an example, it uses the location where the mouse is pressed in the selected object to generate a reference to the right of the object or to the left of the object. When a formula is reused for another slot, for example, the formula to center objects horizontally in the :left slot of the object is copied to the :top slot of the object to center objects vertically in the window, C32 tries to guess which slots needed to be changed. References in the formula can also be generalized into variables. C32 provides a command that will change the formula into a function that takes the objects and slots as parameters.

In C32, prototypical values are explicitly designated by the user. C32 does not detect program structure based on the formula references. Instead the user explicitly provides formulas in LISP while referencing the prototypical objects by direct manipulation, and the user can later instruct C32 to substitute formal parameter variables for prototypical values in these LISP formulas.

Forms/3 is the first form-based language to support automatic generalization. This work is a revision of an earlier version of Forms/3 [Burnett 1991; Burnett and Ambler 1994], which used an internal textual notation to record the generality of a program. The earlier version did not contain a facility for interpreting direct manipulations in order to produce the notation. Also in the earlier version, the internal textual notation described each copy of a form by enumerating exactly how it differed from the model. The notation supported the standard structures found in programming languages such as global references and subroutine-like relationships, but did not support some non-traditional structures because of the circularity they introduced into the notation. In this work, we have generalized upon the previous internal notation, have added the facility to automatically devise the notation from the user's direct manipulations whenever needed, and have added a output static representation of the resulting generality of the program.

2.3. Related Work: Representation Of Generalized Programs

Generalized abstractions are rarely represented explicitly or statically to the programmers in VPLs that use concrete examples in the specification of generalized computations. Forms/3 is the only VPL that provides programmers with a representation of the generalized program with all three of the following attributes: 1) The representation is static, 2) it is a complete description of the generalized program, and 3) it uses the same language to represent the generality as the language that was used to create the program. In C32, for instance, programs are created using the spreadsheet interface and direct manipulation of graphical objects, but Lisp code is used to represent the generalized function. In PT, only one scene of the film can be represented on the screen at a time, i.e., the entire abstraction can not be represented statically at once. Also, PT uses a separate script language to show the filming sequence, it does not provide a complete description of the generalized program. Chimera's representation is static and is consistent with the graphical language programmers used to create the programs, but is not complete: generalizations are represented by textual supplements to the graphical display of commands.

2.4. Related Work: Design Aids For Designing Static Representations

In our effort to design a static representation for the generalized form-based programs, we looked to cognitive evaluative techniques, software metrics for VPLs and design-time evaluation techniques that have been specialized to VPLs. What we found did not perfectly fit our needs, but we were able to build upon works in the latter subfield to derive a technique suited to our task.

2.4.1. Cognitive Evaluative Techniques

The cognitive evaluative techniques that are not specific to VPL evaluation, such as those directed toward graphically oriented software systems in general, are not of much help in evaluating a VPL's static representation. The main reason is that these techniques focus on the user's interactions of a proposed (or implemented) user interface, not on the presence of information that is useful to programmers in such a representation. GOMS, pattern analysis, heuristic evaluation, and layout appropriateness are a few such methods.

GOMS [Card et al. 1983] is a detailed methodology for giving quantitative time predictions for the user to perform activities defined as a detailed sequence of simple tasks. The GOMS stands for Goal, Operators, Methods and Selection Rules. The GOMS model consists of descriptions of the Methods needed to accomplish specified Goals. The Methods are a series of steps consisting of Operators that the user performs. If there is more than one Method to accomplish a Goal, then Selection Rules choose the appropriate Method depending on the context. Figure 2-12 shows a Method for accomplishing the Goal of changing the word 'The cat' to boldface. The GOMS model of task performance can support analysis in a number of ways. It can be used when predicting speeds of task performance in cases where the method of operation is known, that is, when choices are not an issue, or it can used to compare the speed of alternative methods. GOMS techniques are particularly useful in cases where we know the sequence of operation and want to find out how quickly the sequence can be performed by an experienced operator. GOMS does not give accurate answers when the method of operation isn't known or the user is inexperienced.

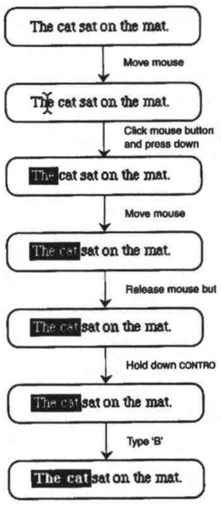


Figure 2-12. A GOMS method for accomplishing the Goal of changing the word 'The cat' to boldface.

Maximal repeating pattern analysis [Siochi and Hix 1991] detects patterns in a user's actions in a working application, with the intention of optimizing the user interface to the most commonly performed actions. In this method, all user interaction with a system is captured into user session transcript files. Transcripts are scanned by a program that detects and reports repeated user actions. On the hypothesis that repeated sequences of user actions indicate interesting user behavior, maximal repeating patterns may therefore indicate usability problems in the interface. For example, detection of frequently repeated user actions may indicate the need for a macro to accomplish those actions with a single action, which would improving user performance time and reduce user errors.

Heuristic evaluation (HE) [Nielsen and Molich 1990; Nielsen 1992] is a general evaluative method that relies on two techniques in combination. First, it employs a team of evaluators to carry out the analysis. Second, a set of design heuristics is used to guide the evaluators. Heuristics can be thought of as general-purpose guidelines. Figure 2-13 lists nine heuristics recommended by Nielsen and Molich.

Simple and natural dialogue	Provide feedback	Provide clearly marked exits
Minimize user memory load	Be consistent	Provide short cuts
Speak the user's language	Prevent errors	Good error messages

Figure 2-13. Usability heuristics used to guide a team of evaluators.

Layout appropriateness [Sears 1993] is a metric aimed at assisting designers in organizing widgets in user interfaces based on the frequency of different sequences of actions users perform. The idea behind layout appropriateness is that every layout can be assigned a cost that will correspond to measures such as time or user preference. See Figures 2-14 and 2-15 for example layouts of an office. To compute layout appropriateness, the designer must provide the set of widgets used in the interface, the sequences of actions users perform, and how frequently each sequence is used. Each sequence of actions represent one method of accomplishing a task. The cost of a layout is computed by assigning a cost to each sequence of actions and weighting those costs by how frequently each sequence is used.

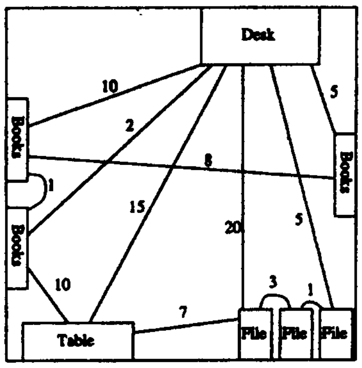


Figure 2-14. An office where links labels indicate average travel per day.

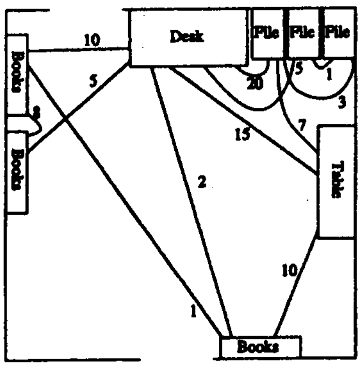


Figure 2-15. An office with reduced travel links.

The most important difference between these evaluative techniques for graphically oriented software and representation design benchmarks is that the former focus on a system's support for fine-grained user interactions, whereas the latter measure a representation's ability to present useful information about a program to programmers.

2.4.2. Software Metrics For VPLs

In the realm of software metrics for VPLs, Glinert introduced a framework for formulating software metrics to compare visual computing environments [Glinert 1989]. The attractiveness to users of a visual computing environment is measured by attributes such as speed of performance, debugging facilities, and support for animation and multiple views. This framework does not deal with the cognitive issues of program representation; it deals only with the features that make an environment appealing to users.

2.4.3. Design-Time Evaluation Techniques For VPLs

Only two design-time evaluation approaches have been applied to VPLs [BellM 1994], Programming Walkthrough [BellB et al. 1991; BellB et al. 1994] and Cognitive Dimensions [Green 1991; Green and Petre 1995]. The Programming Walkthrough is an analysis at the knowledge-level which identifies the language-specific facts needed to perform one or more tasks with a proposed language design. The method requires two things: a representative set of tasks or problems that the system is intended for, and a document describing what a naive user needs to know about the system, which are called the doctrine. The doctrine includes general concepts of the system and its use, as well as advice on how to go about solving problems. Programming Walkthroughs are conducted by a team that includes both the language's designer and an HCI expert (and may include others as well), and is intended for evaluation of a language with respect to its suitability for writing new programs. Because of this emphasis, the evaluation is done on a suite of sample programming problems in the context of the language, as opposed to the language itself.

Cognitive dimensions (CDs) are a set of terms describing the structure of a programming language's components as they relate to cognitive issues in programming. The CDs, which are listed in Appendix A, provide a framework for assessing the cognitive attributes of a programming system and for understanding the cognitive benefits and costs of various features in a language or its environment. The dimensions are intended to be used as high level discussion tools to examine various aspects of languages and environments, and were devised to be usable by language designers and other non-psychologists.

CDs have been used by several researchers to evaluate the cognitive aspects of VPLs, and to make broad comparisons of cognitive aspects of different VPLs. For example, Green and Petre used CDs to contrast cognitive aspects of the commercial VPLs Prograph [Cox et al. 1989] and LabVIEW [Kodosky et al. 1991] (see Appendix A for an excerpt). Modugno also used CDs to evaluate Pursuit, a research programming-by-demonstration VPL [Modugno et al. 1994], and Hendry used CDs to evaluate cognitive aspects of a modification to spreadsheet formula languages [Hendry 1995]. We selected CDs as the foundation for a new design-time approach we devised to provide high-level, design-time measures for a VPL designer to use in designing the language's static representation. We then used this approach to design a suitable static representation for representing generalized, form-based programs.

3. INPUT: DERIVING GENERALIZED ABSTRACTIONS FROM CONCRETE EXAMPLES

We will describe the generalization algorithm in this chapter. We will first give a scenario of a program entered concretely in a form-based VPL and describe the process of recognizing and generalizing the logical relationship in the concrete program. We then present the algorithm in pseudocode and discuss its complexity, correctness and generality. Finally, we will give some performance results of the algorithm.

3.1. Scenario

In satisfying our goal that automatic generalization should not burden the programmer with extra work, we want to support the process of entering a program in a form-based VPL as being flexible, orderless and modeless. We choose not to require the programmer to use an abstract textual programming-language approach to explicitly specify the intended generality, because such an approach would run counter to our goals of concreteness and programming with direct manipulation.

For example, to express the computation for the Forms/3 Fibonacci program in Section 1.1, the following set of actions is needed, but the programmer may enter them in any order. The only restriction is that FIB01 and FIB02 have to be created before the programmer can refer to them.

Copy **FIB** to create **FIB01**
Copy **FIB** to create **FIB02**

In Any Order

- Programmer selects **FIB**'s **N** and types 5
- Programmer selects **FIB**'s **N-1**, clicks in **FIB**'s **N** and types **-1**
- Programmer selects **FIB**'s **N-2**, clicks in **FIB**'s **N** and types **-2**
- Programmer selects **FIB01**'s **N** and clicks in **FIB**'s **N-1**
- Programmer selects **FIB02**'s **N** and clicks in **FIB01**'s **N-1**
- Programmer selects **FIB**'s **FibAns**, types **if (N < 2) then 1 else**, clicks in **FIB01**'s **FibAns**, types **+**, and clicks in **FIB02**'s **FibAns**

Figure 3-1. Sequence of steps to define the program to compute the Fibonacci numbers.

If the formulas happen to be entered in the sequence listed in Figure 3-1, the system will compute and display the value 5 as soon as the formula for FIB's N is entered. Likewise, the system will display the value 4 as soon as the formula for FIB's N-1 is entered, and so on for each formula, as shown in Figure 1-1.

3.2. How The Generalization Algorithm Works

Our approach to produce reusable generalized abstractions from concrete form-based programs is based on two key features:

(1) Deductive analysis of the relationship between concrete program entities to derive a generalized program: Inference-based approaches that involve guessing, which are commonly used in programming-by-demonstration systems to derive generality from concrete examples, were not well-suited to the problem because such inference is most successful in specific problem domains, and we wanted our approach to be usable by languages that support general-purpose programming. The generalization technique presented uses deductive analysis to derive a generalized program from a concrete one. Because it does not use guessing, there is no risk of "guessing wrong" and there is no need for additional dialog with the programmer. This technique supports a modeless direct

manipulation interface, and we will show that it is flexible enough to handle all possible referencing and calling patterns in form-based languages, including some not commonly found in traditional programming languages. This allows the user to program concretely and flexibly, without unnecessary rules and restrictions about what computations can be used to assist other computations.

(2) Use of a calculation's *perspective* during the deductive analysis: Generalization is accomplished through the analysis of logical relationships among concrete program entities from the perspective of a particular computational goal. Since the program structure is not explicitly specified by the programmer, perspective allows the system to locate the relationships that compute the intended results. Section 3.2.2.1 details the importance of perspective for this purpose.

In this section we explain informally how the algorithm works and why. Following this explanation, a more formal presentation will be given in a later section.

3.2.1. Step 1: Recognizing Relationships Among Program Entities

An important key in our algorithm is that as the programmer enters formulas textually or via direct manipulation, the system incrementally adds the cell references in the formula to a *cell reference graph.* The cell reference graph is used to store the relationships and to trigger formula generalization.

Informally, a cell reference graph is simply a representation of the cell references and the derived generalization information about them. Cell references are either references created directly by the user via typing or direct manipulation, called *direct references*, or they are *inherited references* that a new instantiation of a form inherited from the model form. Both of these kinds of cell references are either fully *generalized* or *concrete*.

More formally, the cell reference graph CG = (V, E) is a directed graph where

V= {u | u is a cell in the program}

E = {(u,v) | cell v makes a reference to cell u}

and a function f: E-->L that assigns a label to each edge according to the origin of the reference, where:

$$f(E) = \left[\begin{array}{l} \text{dg: Direct and generalized reference} \\ \text{dc: Direct and concrete reference} \\ \text{ig: Inherited and generalized reference} \\ \text{ic: Inherited and concrete reference} \end{array}\right.$$

We will use the following terminology in describing subsets of E:

$$\begin{array}{ll} DGE \subseteq E & = \{(u, v) \mid f(u, v) = dg\} \\ DCE \subseteq E & = \{(u, v) \mid f(u, v) = dc\} \\ IGE \subseteq E & = \{(u, v) \mid f(u, v) = ig\} \\ ICE \subseteq E & = \{(u, v) \mid f(u, v) = ic\} \end{array}$$

DGE, DCE, IGE and ICE are disjoint sets and their union is E.

Note that u's formula may be a constant instead of references to other cells. If u's formula is a constant, then the constant is stored with u. Table 3-1 gives examples of entries in the cell reference graph. Since the constant case is trival, we will only discuss how the cell references are generalized by the generalization algorithm in the rest of this chapter.

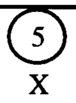
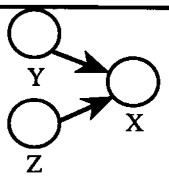
Type of Formula	Example Formula	Dependency information in the cell reference graph
Constant	X = 5	
Expression with references to other cells	X = Y + Z	

Table 3-1. The types of formulas and their corresponding entries in the reference graph.

3.2.1.1. Incremental Processing Of The Programmer's Actions

Because the cell reference graph is built incrementally, incremental analysis is possible. Incremental analysis of the direct references occurs as soon as the programmer enters a formula for a cell that references other cells. An *internal reference* is an edge (u,v) in E such that cell u and cell v are on the same form. Each direct reference that is internal is an element of DGE (and each direct external reference is an element of DCE). Direct internal references do not need further processing to become general—the relationship between cells on the same form is made clear by the fact that they are encapsulated in one form. (This is also true of inherited internal references). Figure 3-2 is a cell reference graph for the Fibonacci example showing only the direct references.

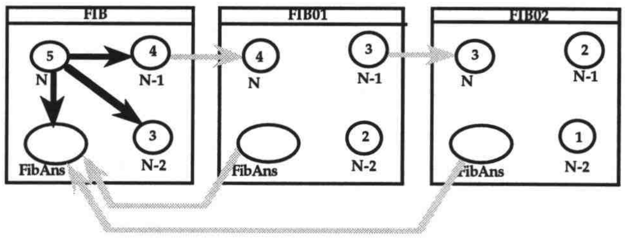


Figure 3-2. A portion of cell reference graph showing only direct references. The direction of the edge indicates the direction of dataflow. For instance, the edge from N to N-1 in FIB indicates that N-1 is a reference to N, i.e., the value of N flows into N-1. Figure 3-3 explains the edge patterns.

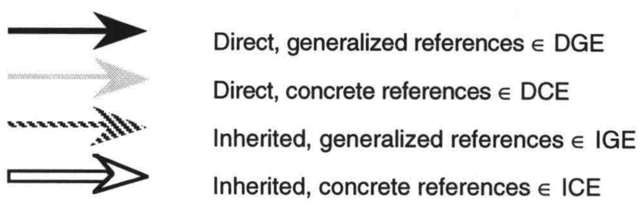

Direct, generalized references ∈ DGE
Direct, concrete references ∈ DCE
Inherited, generalized references ∈ IGE
Inherited, concrete references ∈ ICE

Figure 3-3. Legend for the depiction of edges in the figures.

When the programmer makes a new instantiation of a form for reuse, inherited edges are added to the cell reference graph. If the original edge was in DGE, its inherited version is by definition also fully general and it is placed in IGE. Figure 3-4 shows that the internal references were generalized, and that this is reflected in the inherited references in form instantiations.

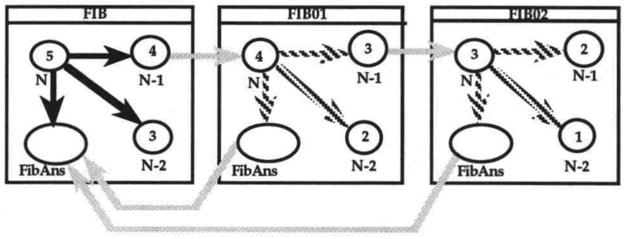


Figure 3-4. A portion of the cell reference graph showing generalized references propagated to instantiations.

Inherited edges that did not originate in DGE, i.e., those that have not been generalized yet, are placed in ICE. This is depicted in the edges connected to the three FibAns nodes in Figure 3-5.

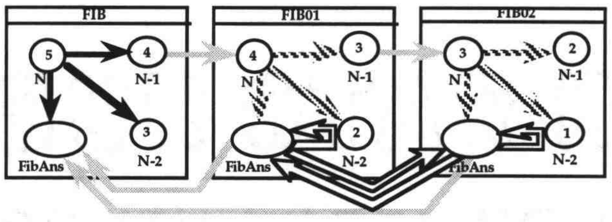


Figure 3-5. The complete cell reference graph for the FIB program.

At this point, the cell reference graph has some similarities to a dataflow graph, but it contains anomalies and information about relationships that needs to be analyzed before it can be reduced to a true dataflow graph. In Figure 3-5, for example, the formula for FibAns in FIB01 and FIB02 is incorrect, because instead of the circular references to themselves and each other, the FibAns cells on FIB01 and FIB02's FibAns should reflect the general roles that they have in computing the N-1'st and N-2'nd numbers in the sequence. This type of anomaly triggers generalization, as is discussed in the next section.

3.2.1.2. Cycle Detection To Trigger Formula Generalization

Whenever edges are added, the graph is analyzed to find out if a cycle has been formed. Detection of a cycle will either result in an error message or will trigger generalization.

An error message is generated by a cycle formed by direct references entered by the programmer, because circular formulas are not allowed in form-based VPLs. Since the cycle detection is done incrementally, the programmer's most recent reference created the cycle, and therefore this last reference must be illegal. The programmer will be warned about the error and the last reference will be rejected.

However, if a cycle is formed that includes at least one edge in ICE, i.e., an inherited and concrete reference, generalization might remove the cycle. In fact, generalization must occur right away to try to remove the cycle because the cycle prevents responsiveness–the computation in its concrete form would be non-terminating. We will refer to such cycles as *possible* cycles. Figure 3-5 included an example of a possible cycle formed by the inherited references of the cell FibAns. (If after generalization, a cycle remains, then the last reference was illegal and an error message will be produced.)

3.2.1.3. Other Triggers

Detecting cycles in the cell reference graph as described above is one way to trigger formula generalization. The other ways that trigger formula generalization are:

(a) Saving a model form: Generalization must be done when saving; otherwise the system might be saving concrete references that are not reusable.

(b) Making a new instantiation of a form: Reusing a model form in this way requires the model form to be generalized first.

(c) Unloading or putting away an instantiation: We will use the term "unloading a form" to mean removing the form from memory, and "putting away a form" to mean removing the form from the screen but not from memory. In either case, since cell reference dependencies are removed from the cell reference graph, the system will not be able to use it later in deriving a generalization. Thus generalization is required before this information is lost.

3.2.2. Step 2: Generalizing The Relationship

The next step of generalization is to reduce the cell reference graph to a generalized dataflow graph. This is done by first performing a modified topological sort on the cell reference graph minus the edges in ICE, which we will call the *reduced cell reference graph* (RCG), to discern the flow of the logical relationships. The formulas are then generalized and recorded by describing the relevant references from the perspective of each cell being generalized.

3.2.2.1. Identifying And Using Perspective

It is from the perspective of each cell being generalized that the generalized references to the other cells contributing to it take place. Finding this perspective is important because (1) it makes known the beginnings and ends of the dependencies, and (2) in mutually dependent forms, as in the case of co-routines and other non-traditional structures, perspective allows the system to deal with each individual cell's computation path separately to avoid generating circular expressions of dependency that would occur if dealing with all the cells on a form as a group.

A modified topological sort is performed to identify the logical relationships of the referenced cells from the perspective of the cell whose formula the algorithm is generalizing. The topological sort is modified in that it preserves the edges in the graph. In the topologically-sorted RCG = (V, E), the vertices in V are ordered such that if RCG contains an edge (u,v), then vertex u appears before vertex v in V. Figure 3-6 shows the topologically-sorted cell reference graph for the Fibonacci example. Notice only those cells relevant to the computation of the cell FibAns are shown in the figure. For example, the N-2 cells are not present in the graph because the generalization algorithm did not actually use them in any way that affects FibAns.

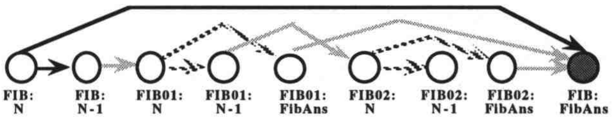


Figure 3-6. Computational path of the cell FibAns (shaded).

3.2.2.2. Recording The Generalized Relationships

After the relationships have been located and sorted out by the topologically-sorted reduced cell reference graph, the relevant portions of these relationships are recorded in the fully generalized formulas. To accomplish this, in the formula for a cell X, each reference to a cell on another form instantiation is described and recorded by enumerating each way the form instantiation is different from the model, if that difference is relevant to the cell X. This recording is done using an internal textual notation, which is described in the next section. (Since recording this information makes the internal description rather long and involved, it is never seen or used by the user.)

It is important for this textual notation to provide enough information for the system (1) to recognize the needed form instantiation if it exists, and (2) to create the needed instantiation from its model form if such an instantiation doesn't exist. This is critical because if the system were not given enough information to automatically locate and create these instantiations, the only way a form could be reused *during execution* would be for the programmer to manually create a new instantiation from the model and modify it, just as he or she did while programming it originally.

If a form instantiation is different from the model in ways that are not relevant to computing the cell X, then that difference is recorded as *don't care*. The *don't care* differences are not used in the computation, but they provide information necessary for the system to recognize the needed form instantiation if it already exists.

For example, recall the Fibonacci problem posed in Chapter 1. As shown in Figure 3-5, the formula for the cell FibAns makes references to three cells, FIB's N, FIB01's FibAns and FIB02's FibAns, and they are recorded as:

FIB's N: this is an internal reference. This reference is recorded as "self:N". (The use of "self" denotes the fact that this reference is on the same form as cell FibAns).

FIB01's FibAns: This is a reference to another cell FibAns on a form instantiation that, when viewed in Figure 3-6 from the perspective of the cell FibAns, is the instantiation of FIB in which N is defined as FIB's N-1. FIB is "self" from the perspective of FibAns, so this reference is recorded as "FIB(N ☞self: N-1):FibAns". The symbol ☞ represents a cell reference. (We chose this symbol because cell referencing is usually done by pointing).

FIB02's FibAns: FIB02's FibAns refers to N, which refers to FIB01's N-1. FIB01's N-1 refers to its N, which refers to FIB's N-1. Since this path leads back to the form FIB, ("self"), the reference is generalized as:

"FIB(N ☞FIB(N ☞self: N-1):N-1):FibAns"

Putting these three references together gives the complete generalized formula for FIB's FibAns:

```
        "If (self:N < 2) then 1 else
                FIB(N ☞ self: N-1):FibAns +
                FIB(N ☞ FIB(N ☞self: N-1):N-1):FibAns"
```

Several additional examples are given in section 3.5, which discusses the expressiveness of the approach.

Note that the Fibonacci example contains many relative (i.e., non-absolute) relationships. Examples of non-absolute relationships in other languages are those created by parameter-passing in traditional languages, by relative referencing in commercial spreadsheets, and referencing patterns such as the networks that can be built up in dataflow languages.

Absolute relationships, the functional equivalent of absolute references in commercial spreadsheets and of references to global variables in traditional languages, are unchanged by generalization. For example, if a cell X on some form F referred to cell FibAns on an instantiation of FIB in which N's formula was 5+2, X's formula would be "FIB(N ☞ 5+2)". The FIB(N ☞ TaxTable:Z) describes an instantiation of FIB in which N's formula is an absolute reference to cell Z on the (model) form TaxTable.

3.2.2.3. Reduction To A Generalized Dataflow Graph.

As each cell's formula is generalized and recorded, the resulting notation represents a generalized dataflow graph. After generalization, it is possible to discover if any cycles remain that were formed by at least one of the generalized edges. Since such a cycle would be illegal, the dataflow graph is analyzed for cycles after generalization is complete.

3.3. The Algorithm And Its Complexity

In this section we present the algorithm more formally in pseudocode and analyze its complexity. The cell reference graph (CG) is implemented as a hash table. A hash table entry for cell reference u is (key, value) where:

key = (u, temporal label)

value = (constant, dependedOn, affects, tag)

constant = number, string, Boolean or constant expressions

temporal label = t or nil where t indicates a time-based reference in the formula

dependedOn = list of tuples (cell reference, temporal label, edgeType) where cell references in the list of tuples are those on-screen references upon which u depends. This list of tuples represents the set of edges ($E_{dep} \subseteq E$).

affects = list of tuples (cell reference, temporal label, edgeType) where the cell references in the list of tuples are those on-screen references that u affects. This list of tuples represents the set of edges ($E_{aff} \subseteq E$).

tag = t or nil where t indicates the hash table entry has been visited already

edgeType = dg, dc, ig or ic

In our prototype implementation of the generalization algorithm, we consider "on-screen" to be those forms at least partially visible on the screen, even if iconified or obscured by other forms. (This particular definition is not the only one possible for the algorithm to work. For example, it would be possible to consider iconified forms to be off-screen.)

Table 3-2 summarizes the algorithm's operations in pseudocode and presents the cost of each operation in isolation. The output notation produced by GeneralizeModel and GenDeps functions directly reflect the language definition of cell references: Table 3-3 lists the types of cell references and their meaning and Tables 3-4 and 3-5 list the textual output notation produced by the generalization algorithm.

Table 3-2. Generalization algorithm's operations in pseudocode and cost.

	Operation	Pseudocode	Cost
G1	Adding an edge e (u, v) to CG	Built-in Lisp hashtable-add Linear search through E_{dep} to check if v exists	$O(1)^{*} + \theta(\lvert E \rvert)$
G2	Deleting an edge e (u, v) from CG	Built-in Lisp hashtable-lookup returns E_{dep} Linear search through E_{dep} to delete v	$O(1) + \theta(\lvert E \rvert)$
G3	Finding an edge e (u, v) in CG	Built-in Lisp hashtable-lookup returns E_{dep} Linear search through E_{dep} to find v	$O(1) + \theta(\lvert E \rvert)$

* In theory, the cost of built-in hash functions are O(1). However, in the worse case scenario, the cost could be as bad as O(n) if there were many hash key collisions.

Table 3-2. Continued.

	Operation	Pseudocode	Cost
G4	Detecting direct cycles in CG	Standard depth-first-search (DFS) algorithm as follows: **DFS** (CG) 1 for each vertex $u \in V(CG)$ 2 do tag[u] $\leftarrow$ notVisited 3 for each vertex $u \in V(CG)$ 4 do if tag[u] = notVisited 5 then **DFS-Visit** (u) **DFS-Visit** (u) 1 tag[u] $\leftarrow$ Visiting 2 for each $v \in Adj[u]$ 3 do if tag[v] = notVisited 4 then **DFS-Visit** (v) 5 else if tag[v] = Visiting 6 then Cycle = true 7 tag[u] $\leftarrow$ Visited where $Adj[u] = \{v \mid (u,v) \in (DGE \cup DCE)\}$	θ(|V| + |E|)
G5	Detecting possible cycles in CG	Standard DFS algorithm same as in G4 except $Adj[u] = \{v \mid (u,v) \in E\}$	θ(|V| + |E|)
G6	Topological Sort	Standard DFS algorithm to sort the set of vertices V and maintain $V_{sorted} \subseteq V$ in search order. From V_{sorted}, generate sorted edges $E_{sorted} \subseteq E$ as follows: **GeneratedTopSortedEdges** 0 $V_{sorted} \leftarrow$ empty 1 for each $u \in V_{sorted}$ 2 do for each v = Adj[u] 3 do $E_{sorted} \leftarrow E_{sorted} \cup \{(u,v)\}$	θ(|V| + |E|)
G7	Recording Generalized Relationship	Standard DFS algorithm to sort the set of vertices V. For each $u_m \in V_{model}$ where $V_{model} \subseteq V$ is the set of cells on model forms, generalize u_m as follows: **GenFormula**(u_m) 1 for each element v in u_m's formula: 2 do case v of 3 cell reference: **GenCell** (u_m,v) 4 else: write v **GenCell**(u_m,v) 1 do case f (u_m,v) of 2 dg: write FormID of v: CellID of v 3 dc: case v of 4 model cell: write FormID of v: CellID of v 5 instantiation cell: write model FormID of v (**GenDeps**(v)):CellID of v **GenDeps** (u) 1 for each $v \in Adj[u]$ 2 do case f (u,v) of 3 dc: write CellID of u <- **GenFormula**(v) 4 ig: **GenDeps** (v) 5 otherwise: error	θ(|V|+ |E|)

Table 3-2. Generalization algorithm's operations in pseudocode and cost.

Cell Reference for X	**Comments**	**Meaning**
Y	Direct reference to a cell Y on the same form	**self:Y**
F:Y	Direct reference to a cell Y on a model form F	**F:Y**
F':Y	Direct reference to a cell Y on a form instantiation F'	**F(***Deps(F':Y)***):Y**

Table 3-3. Language definition table for cell references where Deps(F':Y) is the formulas of cells upon which F':Y directly depends that are relevant to X and are different from cell formulas on F.

GenCell

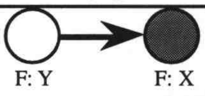
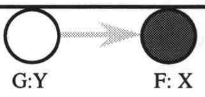
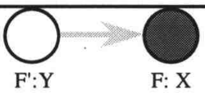
Cell Reference in CG	**Comments**	**Generalized Textual Notation**
	F:X is a direct reference to any arbitrary cell Y on the same form F	**self:Y**
	F:X is a direct reference to any arbitrary cell Y on another model form G	**G:Y**
	F:X is a direct reference to any arbitrary cell Y on a form instantiation F'	**F(***GenDeps(F':Y)***): Y**

Table 3-4. Translations of Table 3-3 as used in the output of GenCell.

GenDeps (F':Y)

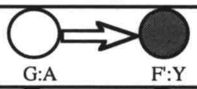
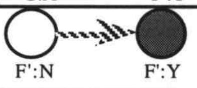
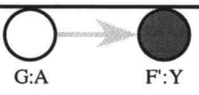
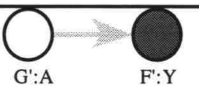
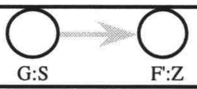
Cell Reference in CG	**Comments**	**Generalized Cell Reference**
	F':Y is an inherited concrete reference to a cell A on a form G.	
	F':Y is an inherited generalized reference to a cell N on the same form instantiation F'.	*GenDeps (F':N)*
	F':Y is a direct reference to a cell A on a model form G	**(Y ☞ G:A)**
	F':Y is a direct reference to a cell A on a form instantiation G'	**(Y ☞ G(*GenFormula(G':A*)):A**
	F':Z is not F':Y's computational path. This doesn't affect the result.	**(Z ☞ DON'TCARE)**
ELSE	All other references are illegal.	

Table 3-5. Output textual notation produced by GenDeps.

From the cost of each operation, the total cost of the generalization algorithm is derived by considering the four cases in which the operations are invoked:

1. Entering a new formula or modify an existing formula of a cell X. There are three possible sub-cases (a-c):
   a. The new formula caused a direct cycle. (Table 3-6a)
   b. The new formula caused a possible cycle. (Table 3-6b)
   c. The new formula does not cause a cycle. (Table 3-6c)
2. Generalization is performed before a form is saved or removed from screen or memory (see Section 3.2.1.3):

   There are two possible sub-cases (a, b):
   a. All the cells on one form are generalized. (Table 3-7a)
   b. All the cells on the screen are generalized. (Table 3-7b)
3. A new instantiation is created from the model. (Table 3-8)

4. A form is removed from the screen or memory. (Table 3-9)

As these tables show, the most expensive of any of these situations costs $|V| * \theta(|V| + |E|)$.

Pseudocode	Total Cost
• Delete edges (E_{dep}) from existing formula from CG (G2)	$\theta(\|E\|)$ +
• Add edges (E_{dep}) from the new formula to CG (G1)	$\theta(\|E\|)$ +
• Direct cycle detection (G3)	$\theta(\|V\| + \|E\|)$ =
	$\theta(\|V\| + \|E\|)$

Table 3-6a. Pseudocode for a new formula which caused a direct cycle.

Pseudocode	Total Cost
• Delete edges (E_{dep}) from existing formula from CG (G2)	$\theta(\|E\|)$ +
• Add edges (E_{dep}) from the new formula to CG (G1)	$\theta(\|E\|)$ +
• Direct cycle detection (G3)	$\theta(\|V\| + \|E\|)$ +
• Possible cycle detection (G4) & topological sort (G5)	$\theta(\|V\| + \|E\|)$ +
• Record generalized relationship for X (G7)	$\theta(\|V\| + \|E\|)$ +
• for each $u \in V_{affects}$ where $V_{affects}$ are those on screen cells depended on X:	
- Re-record generalized relationship for u (G7)	$\|V_{affects}\| * \theta(\|V\| + \|E\|)$ where $\|V_{affects}\| < \|V\|$ =
	$\|V\| * \theta\ (\|V\| + \|E\|)$

Table 3-6b. Pseudocode for a new formula which caused a possible cycle.

Pseudocode	Total Cost
• Delete edges (E_{dep}) from existing formula from CG (G2)	$\theta(\|E\|)$ +
• Add edges (E_{dep}) from the new formula to CG (G1)	$\theta(\|E\|)$ +
• Direct cycle detection (G3)	$\theta(\|V\| + \|E\|)$ +
• Possible cycle detection (G4) & topological sort (G5)	$\theta(\|V\| + \|E\|)$ =
	$\theta(\|V\| + \|E\|)$

Table 3-6c. Pseudocode for a new formula which does not cause a cycle.

Pseudocode	Total Cost
for each $u \in V_F$ where V_F are cells on F	$\|V_F\| \le \|V\|$ *
• Topological sort (G5)	$(\theta(\|V\| + \|E\|)$ +
• Record generalized relationship for u (G7)	$\theta(\|V\| + \|E\|))$ =
	$\|V\| * \theta\ (\|V\| + \|E\|)$

Table 3-7a. Psuedocode for generalizing all the cells on a form.

Pseudocode	Total Cost
for each $u \in V(CG)$ • Topological sort (G5) • Record generalized relationship for u (G7)	$\lvert V\rvert * (\theta(\lvert V\rvert + \lvert E\rvert) + \theta(\lvert V\rvert + \lvert E\rvert)) =$ $\lvert V\rvert * \theta(\lvert V\rvert + \lvert E\rvert)$

Table 3-7b. Psuedocode for generalizing all the cells on the screen.

Pseudocode	Total Cost
for each $u \in V_F$ • Add edges (E_{dep}) from the new formula to CG (G1)	$\lvert V_F\rvert \le \lvert V\rvert *$ $(O(1)) =$ $O(\lvert V_F\rvert)$

Table 3-8. Psuedocode for creating a new instantiation from the model.

Pseudocode	Total Cost
for each $u \in V_F$ • Delete edges (E_{dep}) from existing formula from CG (G2)	$\lvert V_F\rvert \le \lvert V\rvert *$ $(O(1)) =$ $O(\lvert V_F\rvert)$

Table 3-9. Psuedocode for removing a form.

3.4. Correctness Of The Generalization Algorithm

The proof proceeds in two parts. First, we will show that whenever the generalization algorithm is triggered, the generalized formula produced by the generalization algorithm is correct. Second, we will show that the generalization algorithm is triggered often enough.

3.4.1. Part 1: Correctness of Generalized Formulas

We consider a concrete formula for X entered directly by the programmer to be correct, i.e. they are exactly what the programmer entered. We define a generalized formula derived from the concrete formula for X to be *correct* if the generalized formula accurately and completely describes all of the cell dependencies relevant to X. We will first show that the generalized formulas for X on a model form is correct and then show that if the X on the model form is correct, X on an instantiation is also correct.

We will first define three lemmas to help with this proof.

Lemma 1: Before generalization, the cell reference graph contains all of the cell dependencies relevant to X based on its concrete formula.

Proof: In form-based VPLs, the cell dependencies upon which a cell directly depends are contained in formulas. When a new formula is entered or modified for cell X, the concrete formula is parsed and all of the cell dependencies contained in X's formula are added to the cell reference graph as edges in the graph. The formulas for those cells upon which X directly depends on are also added to the cell reference graph in the same way, forming a path to X. This path contains all of the cell dependencies relevant to computing X.

Lemma 2: If the generalization algorithm accurately describes all of the cell dependencies relevant to X based on its concrete formula, then after generalization, the generalized formula also contains all of the cell dependencies relevant to X.

Proof: By Lemma 1, we know that cell reference graph contains all of the cell dependencies relevant to X based on its concrete formula and we know that the generalization algorithm replaces each of the cell dependencies based on its concrete formula with a generalized notation. If the generalization algorithm accurately describes the concrete cell dependency based on its concrete formula (see next lemma), then the generalized notation has the same cell dependencies. Thus the generalized formula also contains all of the cell dependencies relevant to X.

Lemma 3: The generalization algorithm accurately describes all of the cell dependencies in the cell reference graph relevant to X based on its concrete formula.

Proof: Cell dependencies in the cell reference graph based on X's concrete formula are either a reference to a cell Y on a model form or a reference to a cell Y on an instantiation. We will now show that the resulting generalized notation produced by the generalization algorithm accurately describes these two types of references. The generalized notation is produced by three functions, GenFormula, GenCell and GenDeps. See Table 3-2 for the psuedocode of these functions. GenFormula goes through a cell X's concrete formula and replace all of the concrete cell references with generalized ones by calling GenCell.

1) The generalization algorithm accurately describes a reference to a cell Y on a model form.

The first two entries in Table 3-4 shows the generalized textual notations for cell references in X's formula to a cell Y on the same model form and on a different model form. These textual notations are correct because they are the direct translation of the language definition entries in Table 3.3.

2) The generalized algorithm accurately describes a reference to a cell Y on an instantiation.

The remaining entry in Table 3-4 shows the generalized notation for this case. According to language definition Table 3-3, the generalized notation for a reference to a cell Y on a form instantiation F' is defined by an instantiation of the model form F with the formulas for cells upon which F':Y directly depends that are relevant to X and are different from cell formulas on the model F. By Lemma 1, we know that the cell reference graph contains the complete cell dependencies for cell F':Y before generalization. We will now show that the generalized notation produced by GenDeps accurately describes the formulas for cells upon which F':Y directly depends that are relevant to X and are different from cell formulas on the model F. There are several subcases in Table 3-5 which are each proven below:

a) F':Y inherits a cell dependency on a cell A on any form G.

This is an example of the ICE edge in the cell reference graph. This is an inherited external reference from the model which means the cell Y on the model form F also directly depend on a cell A on a form G. By definition, since F':Y is not different from F:Y, an empty string is produced by GenDeps to correctly describe this case. (Note that these ICE edges do not add to the resulting generalized notation, thus they are first removed from the cell reference graph before generalization is performed. See reduced cell reference graph in Section 3.2.2).

b) F':Y inherits a cell dependency on a cell N on the same form.

This is an example of the IGE edge in the cell reference graph. This is an inherited internal reference from the model which means the cell Y on the model form F also directly depend on a cell N on the same model form. Note that F':Y is not different from F:Y, however, since this is an internal reference, there might be other cells on the form F' that F':Y directly depends that are relevant to X and are different from cell formulas on the model F. By definition, GenDeps is called recursively on F':N to correctly describe all the cell formulas that F':Y directly depends and are different from cell formulas on the model F.

c) F':Z directly depends on a cell S on a form G.

This is an example of a cell that is different from the model but it is not one of the cells that F':Y directly depends. Recall from Section 3.2.2.2, if F' contains changes that are not relevant with respect to Y, the *don'tCare* is used in the generalized notation to correctly describe this case.

d) F':Y directly depends on a cell A on a model form G.

This is an example of the DCE edge in the cell reference graph. This is a direct reference which means F':Y is different from the cell Y on the model. By definition, the generalized notation generated correctly describe the cell A on the form G upon which F':Y directly depends and is different from the cell formulas on the model.

e) F':Y directly depends on a cell A on a form instantiation G'.

This is another example of the DCE edge in the cell reference graph. This is a direct reference which means F':Y is different from the cell Y on the model. By definition, the generalized notation generated correctly describe the cell A on a form instantiation G' upon which F':Y directly depends and is different from the cell formulas on the model. GenFormula is called recursively to describe the dependencies of the (shorter) path ending at G':A, eventually reaching case 1 or one of the subcases of case 2 above. Since these cases are correct, the resulting generalized notation is correct also.

3.4.1.1. Correctness of X on a Model Form

We will now show that a cell X on a model form is correct according to the definition of correctness given earlier. By definition, a cell X on a model form is correct if its generalized formula derived from the concrete formula accurately and completely describes all of the cell dependencies relevant to X. By Lemma 1, we know that before generalization, the cell reference graph contains all of the cell dependencies relevant to X based on its concrete formula. By Lemma 3, we know that the generalization algorithm accurately describes all of the cell dependencies in the cell reference graph relevant to X based on its concrete formula. And by Lemma 2, we know that since the generalization algorithm accurately describes all of the cell dependencies relevant to X based on its concrete formula, then after generalization, the generalized formula also contains all of the cell dependencies relevant to X. Thus, X on a model form is correct because the generalization algorithm accurately and completely describes all of the cell dependencies relevant to X.

3.4.1.2. Correctness of a Cell X on a Form Instantiation

Instantiations inherit generalized formulas from the model form, but cells on the instantiations are never generalized by the generalization algorithm because they are not used to generate new instantiations. (New instantiations are always generated from the model form.)

Given that the cell X on the model form is correct as shown in Section 3.4.1.1, a new instantiation of the model form initially inherits the same generalized formula as the model form, thus X's generalized formula on the instantiation also accurately and completely describes all of the cell dependencies relevant to X. Whenever a cell X on the model form is re-generalized, the generalized formula is propagated to the X on the instantiation, thus cell X's formula is kept correct on the instantiation.

If a new concrete formula is entered directly by the programmer for the cell X on an instantiation, the new concrete formula overrides the generalized formula inherited from the model. Since the instantiations are not generalized themselves, cell X will depend only on its new concrete formula which is correct. However, if the new concrete formula requires the model to be re-generalized, we will prove next that model is re-generalized often enough to remain correct.

3.4.2. Part 2: Triggering of Generalization Algorithm

We define the triggering of the generalization algorithm to be *often enough* if the generalization algorithm is triggered in time for ungeneralized information to be reused, and while the information needed for generalization is still in the cell reference graph.

1) The generalization algorithm is triggered in time for ungeneralized information to be reused.

We will use the term *reuse* to mean generating new instantiations from a model or changing a model. There are three cases that information is needed for reuse: a) the programmer makes a new instantiation of the model form, b) the programmer makes a model form available for reuse by saving, and c) the programmer enters or modifies a formula on the model form that may need to be propagated to all the on-screen instantiations. We will show that the generalization algorithm is triggered in time for each of the three cases:

Case 1: The programmer makes a new instantiation of the model form.

As the result, the generalization algorithm is triggered to generalize all ungeneralized information on the model form before a new instantiation is made. Thus the generalization algorithm is triggered in time for Case 1.

Case 2: The programmer makes a model form available for reuse by saving.

As the result, the generalization algorithm is triggered to generalize all ungeneralized information on the model form before the model form is saved. Thus the generalization algorithm is triggered in time for Case 2.

Case 3: The programmer enters or modifies a formula on the model form that may need to be propagated to instantiations.

Instantiations are never reused in the sense defined above, so there is never a need to generalize them. However, on-screen instantiations could prevent the model from being reused if there is a recursive reference. As the result, the new concrete formula is propagated to all instantiations and dependencies from the on-screen instantiations are added to the cell reference graph. Then cycle detection is performed to determine if one or more possible cycles exist in the cell reference graph. The presence of a possible cycle indicates a recursive reference in the concrete formula. If the cell reference graph contains a possible cycle, generalization algorithm is triggered immediately to resolve the recursive reference; if the cell reference graph does not contain a possible cycle, then generalization does not need to be performed right away. Thus, the generalization algorithm is triggered in time for Case 3. Therefore the generalization algorithm is triggered in time for reuse for all three cases.

2) The generalization algorithm is triggered while information needed for generalization is still in the cell reference graph.

Information is only removed from the cell reference graph when information is taken off from the screen. Since the generalization algorithm is always triggered before information is taken off the screen, the generalization algorithm always has all the information it needs to perform generalization in the cell reference graph.

3.5. Examples Of The Expressiveness Of The Approach

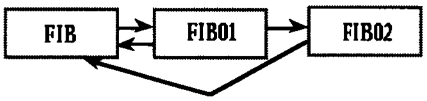


Figure 3-7. A form-collapsed multi-graph of the Fibonacci program.

Because the generalization algorithm does not require any particular calling structure or referencing pattern, it can be used to generalize all legal program structures. We will informally illustrate this by presenting four categories of examples, explaining them through the use of *form-collapsed multi-graphs*, a diagram we introduce solely for the purpose of demonstrating the generality of the approach. Form-collapsed multi-graphs are not part of the approach itself.

Each node in the form-collapsed multi-graph is a form and all the edges from the fully-generalized dataflow graph are retained at the form level. Thus each form has the same number of incoming and outgoing edges as in the cell reference graph. Figure 3-7 shows the form-collapsed multi-graph for the Fibonacci program. This figure points out the slightly unconventional structure that the programmer used in this program.

Program structures are reflected by these form-collapsed multi-graphs. There are four possible patterns of these graphs. An example of each is given in Figure 3-8. For each of the patterns, we will show an example and illustrate how generalization is performed.

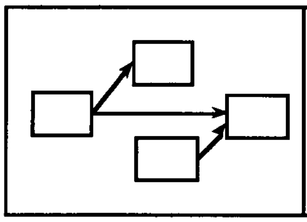


Figure 3-8a. An example with no cycles

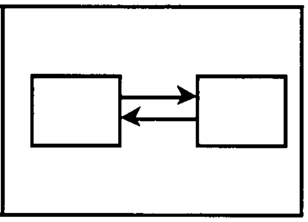


Figure 3-8b. A 2-node cycle

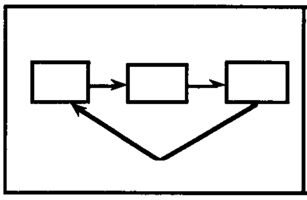


Figure 3-8c. A cycle with more than 2 nodes

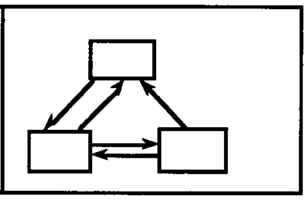


Figure 3-8d. A combination of the other patterns

<u>Pattern 1, no cycles</u>: This pattern reflects abstractions with no parameter-passing, i.e., all references are absolute references, like global variables in other languages. Figure 3-9 shows the dataflow graph of such a program. This program defines a screen saver with a floating image. The location of the floating image is computed and updated based on the system clock. Because all the references are absolute, they don't change with generalization. The formula for cell FloatingImage thus needs only to reflect the dataflow path in absolute terms in recording how referenced cell Image's form differs from the

model in ways that are relevant to cell FloatingImage: "Picture(X ☞ SystemClock:Sec; Y ☞ SystemClock: Min):Image".

Pattern 2, a graph with a 2-node cycle: A 2-node cycle models the functionality of a traditional subroutine with parameters. (Notice that such a cycle is at the granularity of forms, not cells. As previously discussed, circular references at the granularity of cells are not allowed after generalization is complete.) Figure 3-10 shows the dataflow graph of a factorial program. Factorial of a number N is defined as N! = N * (N-1)!. In defining Ans's formula, those cells contributing to the computation and also on the same form as the Ans, namely N and N-1, are recorded as generalized references. The generalized formula for Ans becomes "If (self:N = 1) then 1 else self:N * Fact(N ☞ self:N-1):Ans".

Pattern 3, a cycle with more than 2 nodes: This pattern corresponds to a program structure that is not commonly found in most textual programming languages, because the values are passed forward, and only travel back to the original caller at the end of the forward passing. Figure 3-11 shows the dataflow graph of a program that computes the average of a list of numbers. The generalized formula for Average reflects the path of the parameters as they travel through the forms: "ComputeAvg(ΣX ☞ Summation(List ☞ Counter(List ☞ self:List):List):ΣX; n ☞ Counter (List ☞ self:List):n):$\bar{x}$".

Pattern 4, any combination of the above patterns: Form FIB was one example of Pattern 4, because it included both a 2-node cycle and a 3-node cycle. We have shown how each of the basic three patterns can be handled. Any combination of the above patterns can also be handled, because each cell's formula is recorded by isolating information relevant from that particular cell's perspective. The granularity of describing the form instantiations is at the cell level as it relates to one cell's computation, which means that regardless of how many ways the basic three patterns are combined, no circularity in the notation can arise, because circularity at the cell level is not present.

Another example of Pattern 4 is depicted in Figure 3-12, a dataflow graph containing two 2-node cycles. This is a program to compute the permutation of N,K, defined by P(N,K) = N! / (N-K)!. The generalized formula for P(N,K) is "self:N! / self:N-K!" The formulas for cells N! and N-K! are subsequently generalized as "Fact(N ☞ self:N):Ans" and "Fact(N ☞ self:N-K):Ans" respectively. Two mutually-dependent co-routines would also be modeled by Pattern 4 as a group of 2-node patterns.

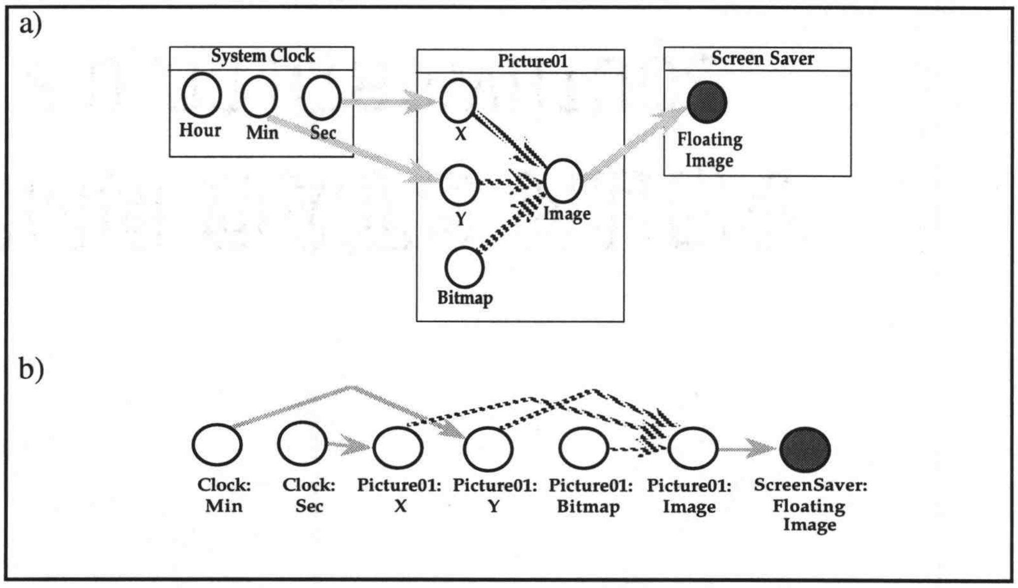


Figure 3-9. a) The cell reference graph of the screen saver program, superimposed on the form-collapsed multi-graph. This is an example of the first pattern, in which there are no cycles. b) The result of the topological sort.

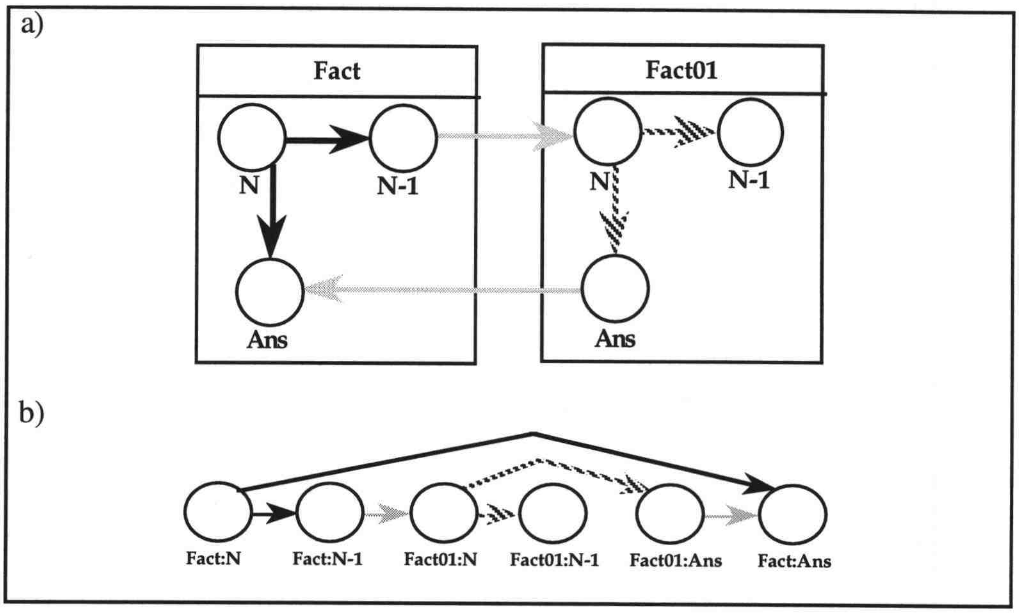


Figure 3-10. a) The cell reference graph of the factorial program, superimposed on the form-collapsed multi-graph. This is an example of the second pattern, a 2-form cycle. b) The result of the topological sort.

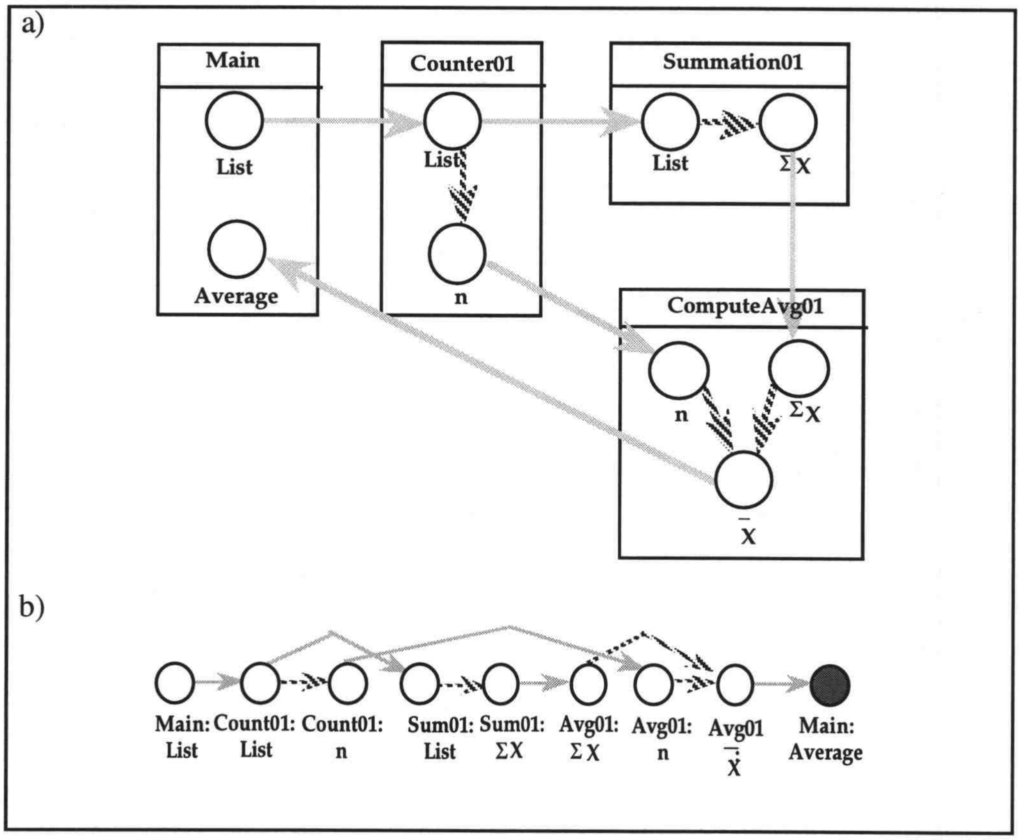


Figure 3-11. a) The cell reference graph of a program to compute the average of a list of numbers, superimposed on the form-collapsed multi-graph. This is an example of the third pattern, a cycle involving more than 2 forms. b) The result of the topological sort.

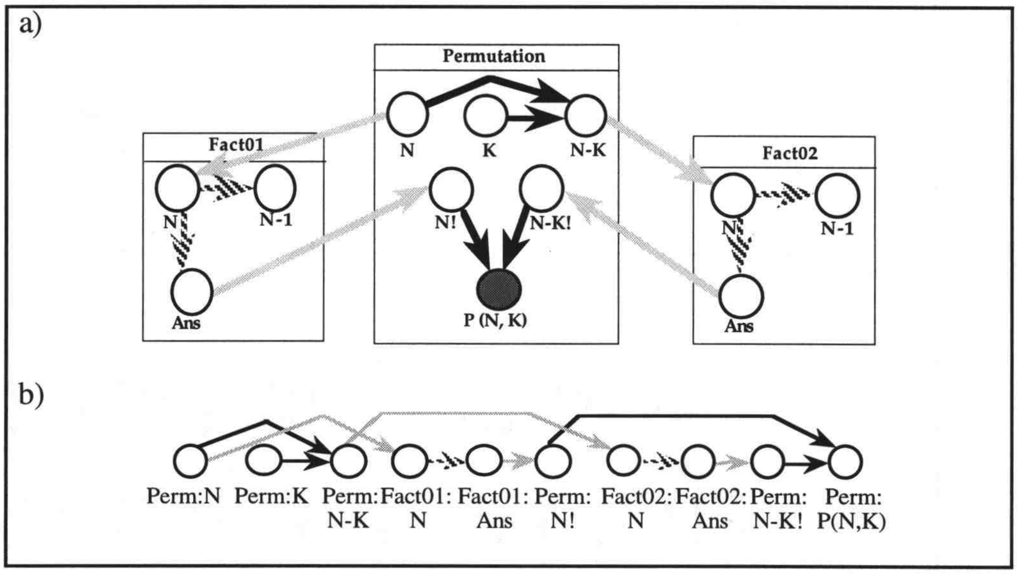


Figure 3-12. a) The cell reference graph of a permutation program, superimposed on the form-collapsed multi-graph. This is an example of the fourth pattern, which is any combination of the other three patterns. b) The result of the topological sort.

3.6. Extensions: Time-Based Formulas, Cell Aggregates And Abstract Data Types

In this section we will extend the generalization algorithm so that it supports not only the form-based VPLs of the present, but also supports additional features that languages under this paradigm might want to add in the future.

3.6.1. Time-based Formulas

There is a recent increase in interest in including an explicit approach to time in VPLs and VPEs ([Du and Wadge 1990], [Freeman et.al 1995], [Almada et.al 1996]). The explicit time approach has also been applied to debugging systems, one-way constraint systems and graphical user interface specification systems ([Tolmach and Appel 1993], [Guimaraes, Correia and Carmo 1992]). In form-based VPLs, formulas for cells can define a vector of values along an explicit time dimension, rather than just an atomic value.

We explored whether such an explicit approach to time in a form-based VPL could be supported by the generalization algorithm using Forms/3 as a prototype.

3.6.1.1. Implementation of Time-based Formulas in Forms/3

In Forms/3's support of such an explicit time dimension, formulas can reference cells' values at earlier moment in time. When cell A's formula references cell B's earlier value in time, cell B is represented in the cell reference graph as a node ($u \in V$) with a special temporal label attached to it to denote the earlier value. The temporal label is needed to distinguish formulas referencing to cells' earlier values from circular references in the cycle detection routine. For instance, if cell Counter's formula is a reference to its own earlier value plus one, "earlier Counter + 1", the reference is added to the graph as shown in Figure 3-13. Without the temporal label "earlier", this would have been detected as a cycle and rejected by the system.

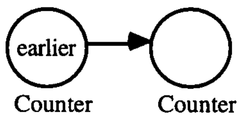


Figure 3-13. Temporal label added to the cell Counter.

3.6.1.2. Impact On The Generalization Algorithm

The temporal label is only used in the cycle detection routine. It does not change the generalization algorithm itself, therefore it does not affect the complexity or correctness of the algorithm.

3.6.2. Cell Aggregates

Arrays and matrices in programming languages are ways to group data elements of similar attributes so they share the same code. User interface components in a GUI toolkit can be also grouped together into aggregates for the same purpose. In a form-based VPL, we can realize the same benefit as arrays, matrices and GUI aggregates by grouping cells into aggregates. To show how our generalization algorithm can be extended to support such cell aggregates, we prototyped the approach in Forms/3.

3.6.2.1. Implementation Of Cell Aggregates In Forms/3

In Forms/3, cells can be grouped into matrices. A matrix has a row number and a column number (called *size cells*) and element cells divided in one or more regions (Figure 3-14). A *region* is a mechanism to define a common formula for one or more cells within a portion of the matrix. The basic approach to matrices is due to Burnett, Walpole, and others; the use of generalization with these matrices is new with this thesis.

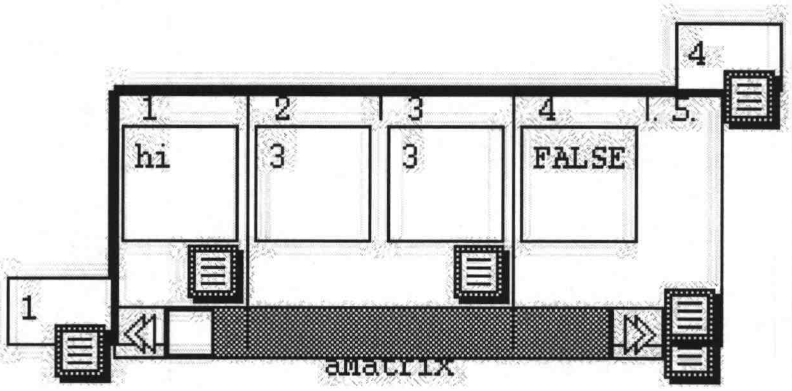


Figure 3-14. An example of a matrix with one row and four columns. There are three regions in this matrix. The first region has one cell with a text string "hi" as its value. The second region has two cells with a value of 3. The last region has one cell with a Boolean value of FALSE.

Table 3-10 presents the semantics of the formulas of references to matrices and their components. As the Table 3-10 shows, a matrix can have a formula that results in a matrix and thus specifies its entire contents, or its component parts can have individual formulas. Region formulas are specified as the formula for representative cell at position (I,J) in the region.

	Formula	Preconditions	Results
Formula for A	B	A and B are matrices of same size	Cells in A will have the same answers as corresponding cells in B. A's row and column cells will have the same answer as B's row and column cells. A will have same number of regions as B.
Specific formula for region r in A	B[x@y]	A and B are matrices x and y are positive integers	All cells in r will have the same answer as the cell in B at row x and column y. This formula overrides formula for A if any.
General formula for region r in A	B[I@J]	A and B are matrices	Cells in r will have the same answers as cells in B at corresponding row I and column J positions. This formula overrides formula for A if any.
Formula for row cell of A	x	A is a matrix x is any expression that produces a positive integer	A will have x rows. This formula overrides the number of rows specified in the formula for A if any.
Formula for column cell of A	y	A is a matrix y is any expression that produces a positive integer	A will have y columns. This formula overrides the number of columns specified in the formula for A if any.
Specific formula for cell X	B[x@y]	B is a matrix X is a cell x and y are positive integers	X will have the same answer as the cell in B at row x and column y.
General formula for cell X	B[I@J]	B is a matrix X is a cell	X will have the same answer as the cell in B at row 1 and column 1. However, this general formula is used for generalized all elements of the matrices.

Table 3-10. Formulas for matrices and their components.

A reference to a matrix cell can be a *specific reference* or a *general reference*. A specific reference "aMatrix[1@1]" refers to the cell at the first row and column of aMatrix. A general reference "aMatrix[I@J]" refers to the corresponding cell in aMatrix. Specific and general cells references can be entered via direct manipulation. If a programmer clicks on a specific cell within a matrix, then the specific cell reference is entered in the formula; if the programmer clicks somewhere inside the matrix but not on a specific cell, then the general formula is automatically generated and entered in the formula.

One of the benefits of regions is that they allow the programmer to define a formula for one of the cells in the matrix and apply the same formula to the rest of the region. If the formula contains concrete references, those references are first generalized before applying to the rest of the region. An example in Figure 3-15 illustrates how this is done. The aMatrix on form In contains a matrix of input numbers and OutM on form Out displays results of adding one to the number. Notice even though cell X on 98-AddOne is displaying the value of the first cell in aMatrix, it has a general reference in its formula. This general reference allows the calculation to be defined once and when the formula is generalized, it can be applied to all the cells in aMatrix. As soon as OutM region's formula is defined to refer to 98-AddOne's Result cell, the formula is immediately generalized and applied to other cells in OutM. The textual notation of the OutM's general formula is "AddOne(X ☞ IN:aMatrix[I@J]):Result". When this general formula is applied to the rest of the matrix OutM during evaluation, new instantiations of AddOne are generated substituting actual subscripts for I and J's in the formula.

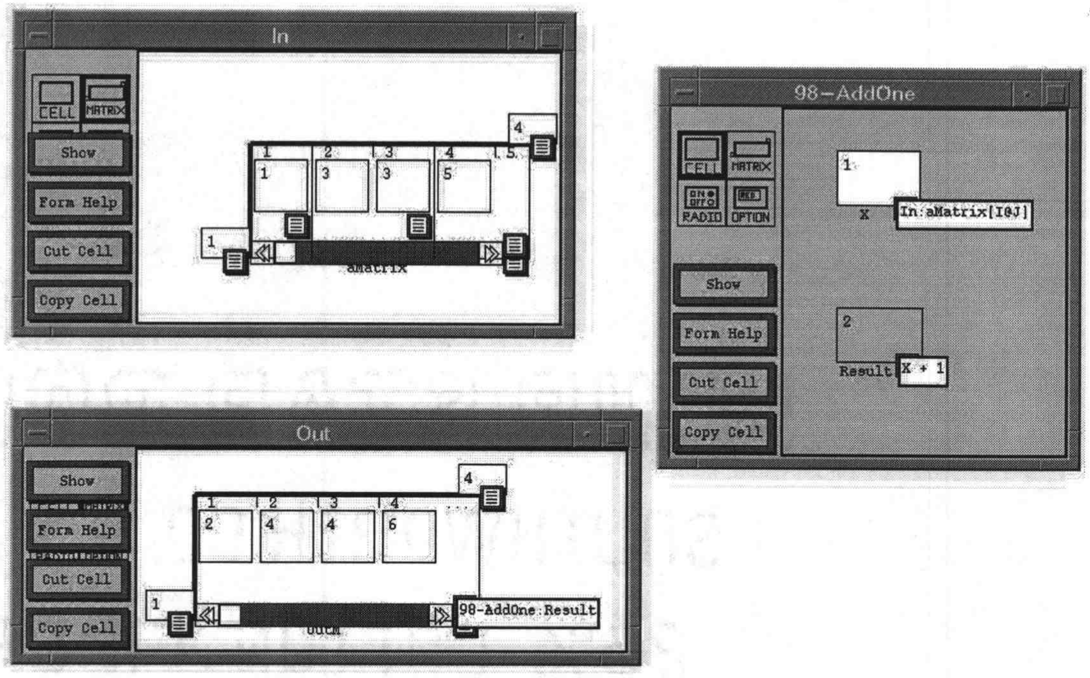


Figure 3-15. An example of a calculation involving matrices.

3.6.2.2. Impact On The Generalization Algorithm

There are new formula dependencies introduced by matrices that needed to be maintained in the cell reference graph. The dependencies between the region, size cells and the matrix are *implicit* and they are updated in the cell reference graph automatically whenever a new formula is entered for the matrix, its regions or size cells. We had to add the ability to automatically maintain this implicit dependency. These implicit dependencies are added to the cell reference graph as though they were explicit cell references made by the programmer. We also made some changes in the programming environment to support the matrices extension. These changes include: 1) Trigger generalization when a region's formula is entered by the programmer and 2) Allow the programmer to generate general matrix references via direct manipulation. The generalization algorithm itself, however, was not changed; therefore the complexity and correctness of the algorithm is unchanged.

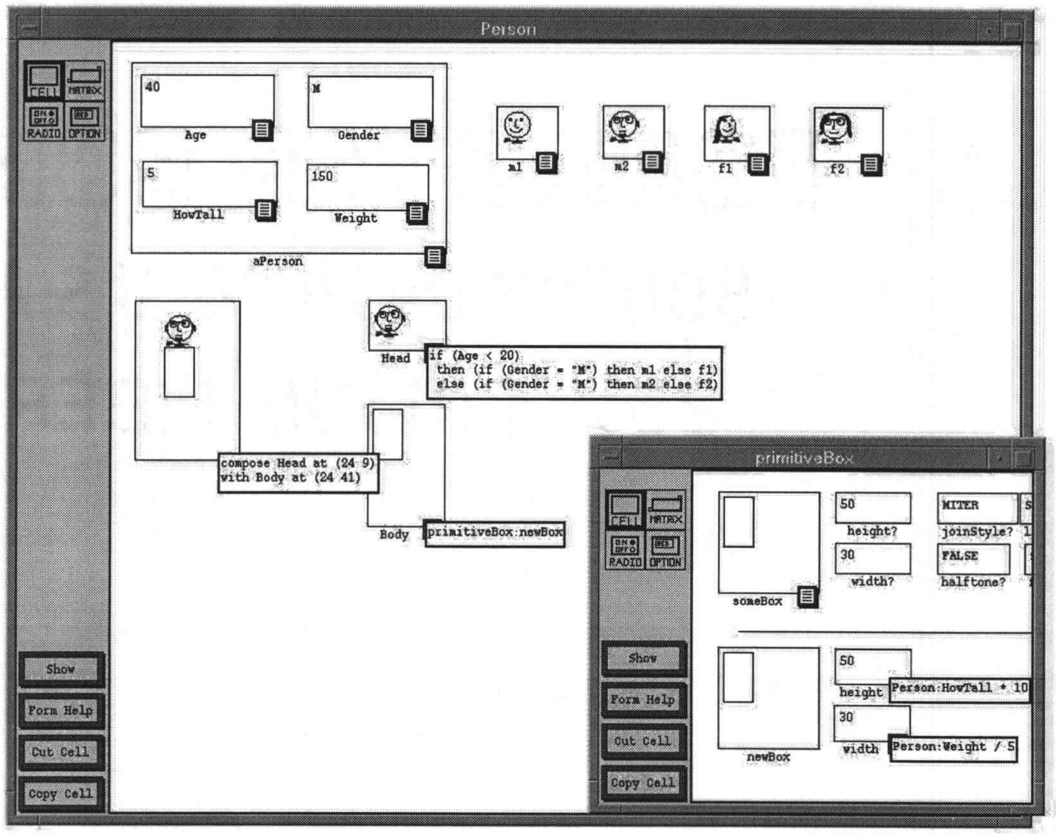
3.6.3. *Abstract Data Types*

An abstract data type is a type defined as a collection of data components and a set of operations that operate on the type. Data abstraction, the concept of abstract data types plus information hiding that enforces use of abstract data types only through the defined operations, is generally accepted as an important feature of programming languages. VPL researchers have also identified data abstraction as an important problem to address for VPLs to scale up to large, realistic programming problems [Burnett et. al 1995]. To show how our generalization algorithm can be extended to support abstract data types, we prototyped the approach in Forms/3.

3.6.3.1. Implementation Of Abstract Data Types In Forms/3

The basic approach to data abstraction in Forms/3 is described in [Burnett and Ambler 1994]. To briefly summarize it, in Forms/3, an *abstraction box* is used to define a *visual abstract data type* (VADT). A visual abstract data type is a 4-tuple: (components, operations, graphical representations, interactive behaviors). The graphical representation and interactive behaviors are important differences between the concept of visual data type and a traditional textual concept of an abstract data type.

A visual abstraction data type is defined on a VADT form. The difference between a VADT form and other forms is that a VADT definition form contains two distinguished items, one is the abstraction box which defines the components of the data, and the other defines the appearance of the data (called the *image cell*). An abstraction box contains a number of interior cells, each of whose formula defines some part of the visual abstract data type. We will illustrate the VADT concept with an example in Figure 3-16. We are defining a visual abstract datatype Person. The programmer has specified that a person is implemented from the composite of its Age, Gender, Height, and Weight. These attributes are defined as interior cells inside the abstraction box Person. The appearance of instantiations of type Person is depending on these four interior cells and using the four bitmaps of faces. Any operation on the visual abstraction datatype Person is also defined on the Person VADT form.



```
Age       40
Gender    M
HowTall   5
Weight    150
```

```
Head
if (Age < 20)
 then (if (Gender = "M") then m1 else f1)
 else (if (Gender = "M") then m2 else f2)
```

```
compose Head at (24 9)
with Body at (24 41)
```

```
Body   primitiveBox:newBox
```

```
height?     50
width?      30
joinStyle?  MITER
halftone?   FALSE
```

```
height  50   Person:HowTall * 10
width   30   Person:Weight / 5
```

Figure 3-16. A person visual abstract data type.

Table 3-11 presents the semantics of the formulas of references to abstraction boxes and their interior cells.

	Formula	Preconditions	Results
Formula for A	B	A and B are abstraction boxes with same interior cells	Interior cells in A will have the same answers as corresponding interior cells in B.
Formula for A	B	A is an abstraction box B is a cell whose formula refers to an abstraction box C with same interior cells as A	Interior cells in A will have the same answers as corresponding interior cells in C.
Formula for any interior cell X in A	Any arbitrary formula F	A is an abstraction box	Cell X will have the answer produced by the formula F. Formula F overrides formula for A if any.

Table 3-11. Formulas for abstraction boxes and their interior cells.

3.6.3.2. Impact On The Generalization Algorithm

There are new formula dependencies introduced by abstraction boxes that needed to be maintained in the cell reference graph. The dependencies between an abstraction box and its interior cells are *implicit* and they are updated in the cell reference graph automatically whenever a new formula is entered for the abstraction box or its interior cells. We had to add the ability to automatically maintain this implicit dependency. These implicit dependencies are added to the cell reference graph as though they were explicit cell references made by the programmer. The generalization algorithm itself, however, was not changed; therefore the complexity and correctness of the algorithm is unchanged..

3.7. Performance Results: Scalability Of The Generalization Algorithm

The generalization technique we developed is amenable to scaling up because its time complexity is bounded by |V|, the number of cells *currently on the screen.* Thus the complexity does not grow with the size of the program, but only with the amount of the program currently on the screen. This section provides some performance results of example programs in Forms/3. All program listings can be found in Appendix C.

Each time generalization is triggered, it is sufficient to generalize only those forms that are currently on the screen, even though they may contain references to off-screen forms. This is because all the other forms were already generalized when they were saved or removed from the screen. This fact is what keeps the time complexity of the generalization technique proportional to the amount of the program currently displayed on the screen.

We started our analysis by creating the factorial function. We measured the number of cells on the screen, number of cells used by the generalization algorithm and total number of cells in the program. When the factorial function was fully specified, it was saved and removed from the screen. We then created the Fibonacci numbers function. Again the measurements were taken and the forms used to define the function were saved and removed from the screen. We continued adding three more functions to the growing program: animated matrix sort without color boxes, animated matrix sort with color boxes and binary tree search. (Some of the forms used for the first animated matrix sort function are reused for the second.) Table 3-12 shows the results. As can be seen from the table, when a function or procedure is fully defined, it can be removed from the screen and thus does not add to the time cost of the generalization algorithm.

	Number of cells on screen	Number of cells used in generalization	Number of cells in the program
Factorial function to compute the factorial of 5	4	4	10
Fibonacci function to compute the Fibonacci number of 5	6	6	40
Animated Matrix Sort without color boxes	50	18	132
Animated Matrix Sort with color boxes	120	33	272
Binary Tree Search	28	13	488

Table 3-12. Performance results of the generalization algorithm. The number of cells in the program include on-screen and off-screen cells. The five functions were created in the same session, one after the other, and the rightmost column reflects the running total.

Table 3-13 shows timing measurements on these functions. All timings were done on an HP 9000/715 with 1 user. The system runs under Lucid Common Lisp 4.1.4 (with dynamic garbage collection disable) and the Garnet User Interface Development Environment 3.0. The numbers in Table 3-13 reflect the total time required to perform the generalization algorithm for each of the functions.

	Total time to perform generalization (in seconds)
Factorial function	0.077
Fibonacci function	0.136
Animated Matrix Sort without color boxes	2.843
Animated Matrix Sort with color boxes	7.025
Binary tree search	0.834

Table 3-13. Timings for the generalization algorithm. Total for each function.

4. REPRESENTATION DESIGN BENCHMARKS

In the last chapter, we presented the algorithm for generalizing the programmer's concrete input into generalized abstractions. Once the generalization has been completed, the generalized abstractions need to be represented back to the programmer. Devising such a representation turned out to be difficult because there is quite a lot of information to convey about the generalized formulas. When we turned to the literature for help in solving this problem, we were surprised to find no design techniques pertinent to the design of static representation of visual programming languages. The Representation Design Benchmarks grew out of our need as VPL designers for a yardstick to use in designing our visual language's static representations. We developed them for use in the general problem of designing a VPL's static representation, and also used them to solve the specific problem of representing the generalized programs produced by the algorithm in the previous chapter.

We will describe the Representation Design Benchmarks in this chapter. We will first define the terminology and show how the benchmarks were developed from cognitive dimensions (CDs). We then present the Representation Design Benchmarks in detail, followed by how to use them in designing static representations for VPLs Forms/3 and ICBE [Zloof and Krishnamurthy 1994; Krishnamurthy and Zloof 1995]. Finally, we will describe an empirical study on the usefulness of the benchmarks to VPL designers.

Building upon the cognitive dimensions developed for programming languages by cognitive psychologists Green and others, the Representation Design Benchmarks provide design-time information that can be used to improve a VPL's static representation while it is still in the design stage. The Representation Design Benchmarks are a concrete application of several of the cognitive dimensions for programming systems by researchers from the field of cognitive psychology [Green 1991; Green and Petre 1995]. See Appendix A for a complete list of cognitive dimensions. The cognitive dimensions provide a foundation that is appropriate to the cognitive issues of representing programs, and provide an increment in formality over previous ad-hoc methods. We based our measures on the particular cognitive dimensions that could be applied to VPL static representations, and added three kinds of refinements: we provided concrete ways of measuring several of the cognitive dimensions at design time, directly focusing them on the static representation part of a VPL.

4.1. Terminology And Overview

The problem to which we intend Representation Design Benchmarks to contribute is the design of better VPL static representations. To focus directly on this problem, we measure a VPL's static representation in isolation from the rest of the VPL. We believe that measuring only the static representation of a VPL—even if the rest of the VPL is highly interactive and dynamic—is necessary if we are to get a clear view of the strengths and weaknesses of that static representation. To do this, we must first be precise about exactly what is to be measured by the benchmarks, namely the VPL's navigable static representation, which we define next.

Informally, a VPL's static representation is the appearance of a visual program "at rest" such as on a screen snapshot. More formally, we will use the term *static representation* to mean the set of every item of information about a program that can be displayed simultaneously on an infinitely large piece of paper or screen.

Although the paper supply expands flexibly to accommodate the size of the program being printed, a computer's display screen does not. Thus, to account for the accessibility of static representations when viewed on a display screen, we must also consider a VPL's set of dynamic navigational devices (menus, scrollbars, etc.) that map a static representation on the infinitely large screen to a finite physical screen. We will term this set of such devices that take a static representation as input and map it to a subset of that static representation as output the *navigational instrumentation*. Finally, we define a language's *navigable static representation* as the tuple (S, NI), where S is the VPL's static representation and NI is the VPL's navigational instrumentation.

For example, consider a programming-by-demonstration VPL that displays a static story board of the modifications that were demonstrated on the objects in the program. Also suppose a static dataflow view of the program may be placed on the screen via a pull-down menu selection, and removed similarly. Let us consider whether the dataflow view is part of the VPL's navigable static representation.

Following the definition of navigable static representations, this view is in the navigable static representation if and only if it is in S or NI. Static views do not fit the definition of dynamic navigation devices, so the static dataflow view is not in NI. A key point in determining whether it (or any a visible item of information) is in S lies in the word "simultaneously" in the definition of static representations.

In order to achieve simultaneousness, the on-screen lifetime of the item of information must not be curtailed unless the programmer chooses to remove it. Returning to our example, if the programmer cannot have the dataflow view on display at the same time the other items of S are displayed (on the infinite screen), then that view is not in S. After all, if the screen snapshot cannot include both views, then the views are not static. In other words, if adding the availability of a dataflow view decreases the story board view's availability, as would be the case if both are accessed by a browser tool allowing only one view at a time, then neither view is an element of S because the first view's lifetime ends when the second view is displayed. However, if both views can be displayed simultaneously and permanently, such as by multiple dynamic browser tools that operate independently of one another, then both views are elements of S and therefore of the VPL's navigable static representation.

As this example demonstrates, there are elements of VPLs that are neither in S nor in NI. Examples include animation, sound annotations, and alternative views that cannot remain indefinitely on the screen. Elements of a VPL that are not in S or NI are not measured by the benchmarks. This is not to say that such elements are not valuable, but only that they are outside the scope of the benchmarks, which were devised to help the designer focus exclusively on just one portion of the VPL—the navigable static representation.

While isolating a portion of the VPL for study in this way is artificial, it provides the designer with a more precise view of that portion than has been achieved in the past. From a cognitive standpoint, it is reasonable to expect better static representations to reduce a programmer's memory load, because the more information about a program on the screen or printout, the less information that must be recalled from a programmer's memory.

Also note that the definition of a navigable static representation does not distinguish between language-related versus environment-related aspects of a VPL. Thus, classifying an item of information as language-related or environment-related does not help determine whether it is in the navigable static representation. This is because Representation Design Benchmarks focus on the availability and quality of information provided to the programmer, not on which piece of the VPL is doing the providing a particular item of information.

4.2. From Cognitive Dimensions To Representation Design Benchmarks

We selected CDs as the foundation for our approach because they were the most conducive to our goal of providing high-level, design-time measures for a VPL designer to use in designing the language's navigable static representation. From this foundation, we derived a set of benchmarks to obtain quantitative measurements of navigable static representations as follows.

We started by selecting the CDs that could be applied to considering (1) the characteristics (denoted Sc) or (2) the presence (denoted Sp) of the elements of a static representation S. For example, the Closeness of Mapping CD pertains to characteristics of static representation elements (Sc), because it considers the characteristic of how a programming language's constructs compare to the entities in a particular domain. On the other hand, the Progressive Evaluation CD refers to the presence of a program's answers in a programming environment; since these answers could also be shown on a static view, this CD can be applied as a possible element (Sp) of the static representation.

We then narrowed the selected dimensions to focus them solely on navigable static representations. In the above example, the Progressive Evaluation CD relates to the dynamic display of answers, so it was narrowed to focus solely on inclusion of answers in the navigable static representation.

For this narrowed set of CDs, we devised quantitative Sc and Sp measures. In addition, for each Sp benchmark for S, we devised a corresponding coarse-grained effort measure of the number of steps the navigational instrumentation NI requires for the programmer to display that element of information, i.e., to map S from the infinite screen to a finite screen in such a way that the element is visible. The benchmarks are summarized in Table 4-1.

Table 4-1. Summary of the Representation Design Benchmarks.

Benchmark Name	Sc	Sp	NI	Aspect of the Representation	Computation
D1		x		Visibility of dependencies	(Sources of dependencies explicitly depicted) / (Sources of dependencies in system)
D2			x		The worst case number of steps required to navigate to the display of dependency information
PS1		x		Visibility of program structure	Does the representation explicitly show how the parts of the program logically fit together? Yes/No
PS2			x		The worst case number of steps required to navigate to the display of the program structure
L1		x		Visibility of program logic	Does the representation explicitly show how an element is computed? Yes/No
L2			x		The worst case number of steps required to make all the program logic visible
L3	x				The number of sources of misrepresentations of generality
R1		x		Display of results with program logic	Is it possible to see results displayed statically with the program source code? Yes/No
R2			x		The worst case number of steps required to display the results with the source code.

Table 4-1. Continued.

Benchmark Name	Sc	Sp	NI	Aspect of the Representation	Computation
SN1		x		Secondary notation: non-semantic devices	SNdevices / 4 where SNdevices = the number of the following secondary notational devices that are available: optional naming, layout devices with no semantic impact, textual annotations and comments, and static graphical annotations.
SN2			x		The worst case number of steps to access secondary notations
AG1		x		Abstraction gradient	AGsources / 4 where AGsources = the number of the following sources of details that can be abstracted away: data details, operation details, details of other fine-grained portions of the programs, and details of NI devices.
AG2			x		The worst case number of steps to abstract away the details
RI1		x		Accessibility of related	Is it possible to display all related information side by side? Yes/No
RI2			x	information	The worst case number of steps required to navigate to the display of related information.
SRE1	x			Use of screen	The maximum number of program elements that can be displayed on a physical screen.
SRE2	x			real estate	The number of non-semantic intersections on the physical screen present when obtaining the SRE1 score
AS1, AS2, AS3	x x x			Closeness to a specific audience's background	ASyes's / ASquestions where ASyes's = the number of "yes" answers, and ASquestions = the number of itemized questions, to questions of the general form: "Does the <representation element> look like the <object/operation/composition mechanism> in the intended audience's prerequisite background?"

Table 4-1. Summary of the Representation Design Benchmarks. Sc denotes measures of the characteristics of elements of S. Sp denotes measures of the presence of potential elements of S. Each Sp measure has a corresponding NI measure for the worst case number of steps required to navigate to the corresponding element.

Using the benchmarks is a 3-step process. First the designer determines whether the aspect of the representation measured by a benchmark applies to their VPL and if so, identifies the aspect of their language's representation that corresponds to the element or characteristic to be measured by the benchmark. (For example, a designer of a VPL intended for only tiny applications would probably omit the scalability benchmarks.)

Second, the designer computes the measurements. Third, the designer interprets this computation; that is, he or she maps the measurement to a subjective rating scale. We have provided a sample of such a mapping in Appendix B. Since such a mapping necessarily reflects the goals and value judgments of a particular language's designers, we would expect different designers to use mappings that are different than the sample.

4.3. The Benchmarks In Detail

In this section, we will discuss the relationships of each of the benchmarks with their corresponding CDs and how to compute the measures. The benchmarks are divided into three categories: Understandability, Scalability and Audience-specific benchmarks.

4.3.1. Understandability Benchmarks

This section describes benchmarks for elements that relate to understandability of a program's representation.

4.3.1.1. Visibility Of Dependencies

We will say there is a *dependency* between P1 and P2 to describe the fact that changing some portion P1 of a program changes the values stored in or output reported by some other portion P2. P1 and P2 can be of arbitrary granularity, from individual variables to large sections of a program. Dependencies are the essence of common programming/maintenance questions such as "What will be affected if P1 is changed?" and "What changes will affect P2?" Green and Petre noted hidden dependencies as a severe source of difficulty in understanding programs in their discussion of the Hidden Dependencies CD [Green and Petre 1995].

Benchmarks D1 and D2 are based upon this CD. D1 is an S_p benchmark that measures whether the dependencies are explicitly depicted in the representation, and D2 is an NI benchmark that measures how easily this information can be accessed via the supporting elements of NI.

To compute benchmark D1, first the designer identifies the dependencies in the VPL using the definition at the beginning of this section, subdividing them into groups based on the sources of the dependencies. For example, a standard dataflow language might have only one source of dependencies, namely the data's flow, while a spreadsheet might have two sources, a cell's formula dependencies and macro-based effects on a cell. Second, the designer multiplies the number of sources found by two to account for the fact that every bi-directional source of dependency is actually two, uni-directional dependency sources: one direction tells what will be affected by a portion of a program P1, and the other tells what other portions P1 affects. For example, in a digraph of such dependency information, one direction tells what nodes are reachable from P1, and the other tells what nodes have paths to P1. Finally, the designer divides the number of these uni-directional dependency sources that are explicitly represented by the total number of uni-directional dependency sources in the VPL.

Like all S_P benchmarks, D1 is measured under the assumption of an infinite screen size. Each S_P benchmark's accompanying NI benchmark then measures the cost of mapping the elements from the infinite screen to a finite screen. For dependencies, the NI benchmark is D2, which is simply a count of the number of steps needed to navigate to the dependency information.

4.3.1.2. Visibility Of Program Structure

We will use the term *program structure* to mean the relationships among all the modules of a program, where a module is a collection of program elements, and the boundaries of a module are determined in a language-specific manner. For example, in some languages a module is a procedure, function, or macro; in others it is a class or a method; and in others it is a form or a storyboard. Examples of relationships among them include caller/callee relationships, inheritance relationships, and dataflow relationships.

From the programmer's standpoint, a depiction of program structure answers questions such as "What modules are there in this program?" and "How do these modules logically fit together?" Example depictions of program structure include call graphs, inheritance trees, and diagrams showing the flow of data among program modules.

The benchmarks in this group are related to the Role Expressiveness CD. The Role Expressiveness CD describes how easily a programmer can discern the purpose of a particular piece of a program. Some of the devices that have been empirically shown to help communicate role expressiveness are use of commenting and other secondary notations, meaningful identifiers, and well-structured modules. The benchmarks in this section consider the representation of the structural role of a portion of a program, and the benchmarks in the section on secondary notation consider some other kinds of role information. Benchmark PS1 shows the presence or absence of program structure information in S, and benchmark PS2 measures the number of steps required for a programmer to navigate to this information.

4.3.1.3. Visibility Of Program Logic

If the fine-grained logic of a program is included in a static representation, we will say the program logic is *visible*. If the visibility of the program logic is complete, the representation includes a precise description of every computation in the program. This benchmark group is one of two benchmark groups derived from the Visibility and Side-by-Side Ability CD, and measures visibility. (The other group of benchmarks based on this CD focuses on side-by-side ability, and will be presented in the scalability section.) Textual languages traditionally provide complete visibility of fine-grained program logic in the (static) source code listing, but some VPLs have no static view of this information. Without such a view, a programmer's efforts to obtain this information through dynamic means can add considerably to the amount of work required to program in the language. For example, one study of spreadsheet users found that experienced users spent 42% of their time moving the cursor around, most of which was to inspect cell formulas [Brown and Gould 1987].

Benchmark L1 measures whether S provides visibility of the fine-grained program logic, and benchmark L2 measures the number of steps to navigate to it. Benchmark L3 is an S_C benchmark focusing on a problem of completeness of visibility common in many VPLs that use concrete examples, namely accuracy in statically depicting the generality of a program's logic. For example, in a by-demonstration VPL, a programmer might create a box expansion routine by demonstrating the desired logic on one particular box. If the static representation S consists solely of before, during, and after pictures of that one particular box, it does not provide general enough information to tell what the "after" picture would be if a different-sized box were the input.

4.3.1.4. Display Of Results With Program Logic

This group of benchmarks measures whether it is possible and feasible to see a program's partial results displayed with the program source code. The benchmarks in this group are derived from the Progressive Evaluation CD. The idea behind the original CD, which related to the dynamics of interactive programming environments, was that the ability to display fine-grained results (values of each variable, etc.) at frequent intervals allows fine-grained testing while the program is being developed, which has been shown to be important in debugging (see [Green and Petre 1995] for a discussion). Our projection of this notion to navigable static representations is to consider whether such results are included in S. Including these results in a navigable static representation would allow the programmer to study a static display of this test data integrated with the static display of the accompanying program logic.

Benchmark R1 measures whether or not it is possible to see the results displayed statically with the program source code and benchmark R2 measures the number of steps required to do so.

4.3.1.5. Secondary Notation: Non-Semantic Devices

A VPL's *secondary notation* is its collection of optional non-semantic devices that a programmer can include in a program. Since it is a collection of non-semantic devices, changing an instantiation of secondary notation, such as a textual comment, does not change a program's behavior. The benchmarks in this group are derived from the Secondary Notations CD, and are also related to the Role Expressiveness CD discussed previously. Petre argues that secondary notation is crucial to the comprehensibility of graphical notations [Petre 1995]. For example, the use of secondary notations such as labeling, white space, and clustering allows clarifications and emphases of important information such as structure and relationships.

This group of benchmarks focuses on the subset of a VPL's secondary notational devices that are static. Benchmark SN1 simply measures the presence of such notational devices, and benchmark SN2 measures the number of steps required to navigate to instantiations of them. We identified four non-semantic notational devices that might be included in a VPL's navigable static representation: (1) optional naming or labeling, i.e. the non-required ability to attach a name or label to a portion of the program; (2) layout of a program in ways that have no semantic impact; (3) textual annotations and comments; and (4) static graphical means of documenting a program, such as the ability to circle a particular portion of the program and draw an arrow pointing to it. (Time-based annotations such as animation and sound are by definition not part of a navigable static representation.) To compute benchmark SN1, the designer divides the number of secondary notational devices available in the representation by four, the total number of secondary notational devices listed above[1].

4.3.2. Scalability Benchmarks

In [Burnett et al. 1995], a VPL's navigable static representation is counted as an important aspect in the language's overall scalability. By measuring factors pertinent to the representation's ability to display large programs, the benchmarks in this section reflect both the scalability of the representation itself and its influence on the VPL's scalability as a whole.

4.3.2.1. Abstraction Gradient

In the Abstraction Gradient CD, the term *abstraction gradient* was used to mean a VPL's amount of support for abstraction. When applied to VPL representations, to support abstraction means to provide the ability to exclude selected collections of details from the representation, replacing such a collection by a more abstract (less detailed) depiction of that collection of details. Abstraction is a well-known device for scalability in programming languages, because it usually reduces the number of logical details a programmer must understand in order to understand a particular aspect of a program. In addition to this benefit, support for abstraction in a navigable static representation generally allows a larger fraction of a program to fit on the physical screen, since replacing a collection of details by an abstract depiction almost always saves space. Thus there are

[1]Four is simply the number we were able to identify. Obviously, this is a case where experience in practice may turn up additional kinds of secondary notations, in which case the divisor should be increased. An alternative benchmark would have been to eliminate such a divisor by using a raw count instead of a ratio, but our experiences indicated that this benchmark was more useful in alerting designers about opportunities for improvements if it computed a ratio.

both cognitive and spatial ways that a representation's abstraction gradient is tied to its scalability.

Benchmark AG1 measures the sources of details that can be abstracted away from a representation, and benchmark AG2 measures the number of steps required to do so. As with the secondary notations benchmark SN1, AG1 is a ratio instead of a raw count, to bring out opportunities for improvement. For the denominator, we identified four sources of detail in a VPL that might be abstracted away in a representation: data, operations, other fine-grained portions of the program, and details of navigational instrumentation devices (control panels, etc.)[1]. Thus, to calculate the benchmark AG1, the designer divides the sources of detail that can be abstracted away in S by four.

4.3.2.2. Accessibility Of Related Information

From a problem-solving point of view, any two pieces of information in a program are related if the programmer thinks they are. Based on the Visibility and Side-by-Side-Ability CD, the benchmarks in this group measure a programmer's ability to display desired items side by side. Green and Petre argued that viewing related information side by side is essential, because the absence of side-by-side viewing amounts to a psychological claim that every problem is solved independently of all other problems [Green and Petre 1995]. Benchmark RI1 measures whether it is possible to include all related information in S, and benchmark RI2 measures the number of steps to navigate to it.

[1]Unlike SN1, the coverage of this list is complete. Recall that the definition of a navigable static representation is the tuple (S, NI). The first two elements in the list cover two particular portions of S and the third covers anything else in S. The fourth element in the list covers NI.

4.3.2.3. Use Of Screen Real Estate

Screen real estate denotes the size of a physical display screen, and connotes the fact that screen space is a limited and valuable resource. The benchmarks in this group are S_C benchmarks derived from the Diffuseness/Terseness CD, and have two purposes. First, they provide measures of how much information a representation's design can present on a physical screen without obscuring the logic of the program. Second, they bring important trade-offs to the fore, providing a critical counterbalance to the other benchmarks by accounting for the screen real estate space costs of the design decisions.

As in other aspects of computer science, designing VPL representations involve time/space trade-offs. However, for representation design, "time" is the programmer's time to locate the needed information on the screen (or navigate to it if it is off the screen), or to reconstruct it from memory if it cannot be displayed simultaneously with other needed information. "Space" is physical screen space. The tension between time and space in this context is that, if the information is already on the screen, the programmer's time to locate it is reduced but more screen space is spent; on the other hand, if the information is not displayed, less space is spent but the programmer must expend more time to locate or reconstruct the information.

Time versus space is not the only trade-off to be considered in representation design—there are also trade-offs between space versus quality of presentation. One way quality of presentation deteriorates is if so much information is placed on the screen, it will not fit unless there are *non-semantic intersections*. A non-semantic intersection is a spatial connection or overlapping of screen items, in which the intersection has no effect on the program's behavior. See Figure 4-1.

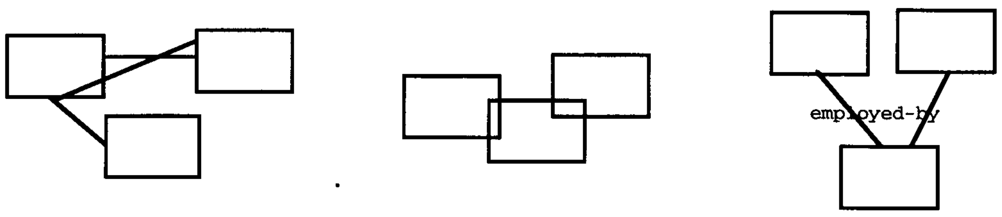


Figure 4-1. Non-semantic intersection examples that might be found in a VPL. Non-semantic intersection examples that might be found in a VPL. (Left): Line crossings. (Middle): Unrelated boxes overlapping, seeming to imply a logical grouping. (Right): A line's label overlaps an unrelated line.

Since the benchmarks in this group relate to physical screen space, the designer should perform these benchmarks on a physical screen representative of those upon which the language is expected to be run. For example, a language intended for low-end Macintosh computers should be measured on the screen size most commonly included/purchased with such systems. Benchmark SRE1 is the maximum number of program elements that can be laid out on such a physical screen. (The term "program element" is defined by the designer in a manner specific to the VPL being measured.) In performing the benchmark, the designer may assume any layout strategy, as long as it is one that the VPL's programmers might use. This benchmark allows the designer to quantitatively compare how alternative design ideas increase or decrease screen space utilization. Benchmark SRE2 is the number of non-semantic intersections that can be counted on the layout chosen in performing benchmark SRE1, thereby providing a measure of whether such a layout makes non-semantic intersections likely.

4.3.3. Audience-Specific Benchmarks

Many VPLs are special-purpose languages designed to make limited kinds of programming accessible to a particular audience. The target audience is composed of people who do not want to use conventional programming languages for those kinds of programming. We will use the term *audience-specific VPLs* to describe such VPLs.

Examples of audience-specific VPLs range from coarse-grained VPLs for scientists and engineers to use in visualizing their data, to embedded VPLs for end-users to use in automating repetitive editing tasks. Although the benchmarks in the previous sections apply to these VPLs, because the task at hand is indeed programming, a new issue not covered by the benchmarks described so far arises: whether the audience-specific VPL's representation is well suited to its particular audience.

The benchmarks in this category focus on this issue. They were derived from the Closeness of Mapping CD. This CD considers the question of whether programming in a given language is similar to the way its audience might solve the same problem by hand in the "real world". This question has implications regarding how well the audience can use the language. For example, Nardi points to a number of empirical studies indicating that people consistently perform better at solving problems couched in familiar terms [Nardi 1993]. In the realm of representation design, the issue narrows to whether the *appearance* of a VPL's elements is similar to the appearance of the corresponding elements in the audience's experience and background.

These benchmarks are unlike the benchmarks presented thus far in two ways. The first difference is that they compare representation elements with the prerequisite background expected of the VPL's particular audience, and thus make sense only for audience-specific VPLs. The second difference is that all the benchmarks in this section are performed the same way—by answering the following question: Does the <representation element> look like the <object/operation/composition mechanism> in the intended audience's prerequisite background?

The audience-specific benchmarks AS1, AS2 and AS3 are S_C benchmarks for the objects, operations, and spatial composition mechanisms respectively. Computing them is a matter of answering the question from the previous paragraph for each element of the representation.

To do this, the designer must first identify what is in the intended audience's prerequisite background; that is, what prerequisites this audience is expected to have fulfilled. The prerequisites include whatever prior computer experience (if any) is expected as well as other kinds of knowledge that might be expected. For example, the intended audience of a macro-building VPL for graphical editing might be expected to know not only about editing graphics on a computer, but also about everyday objects and phenomena such as telephones, the flow of water through pipes, and gravity.

The next step is to identify the objects and operations that are depicted in the representation, along with the ways these objects and operations can be spatially composed. (It is not of critical importance whether a particular element is classified as an object, as an operation, or as a composition mechanism, since all are measured the same way; the division into the three groups is simply a way to help organize the identification process.) Finally, for each object, operation, and composition mechanism identified, the designer notes whether its appearance looks like the corresponding item from the audience's prerequisite background.

Thus, to compute AS1, the designer asks, for each object in the representation, "Does the <representation element> look like the <object> in the intended audience's prerequisite background?" and divides the total number of "yes" answers by the total number of objects. AS2 and AS3 are computed the same way: AS2 for the operations, and AS3 for the spatial composition of objects and operations.

4.4. Applying Representation Design Benchmarks In The Design Process

In this section, we will provide examples of applying the Representation Design Benchmarks using VPLs Forms/3 and ICBE.

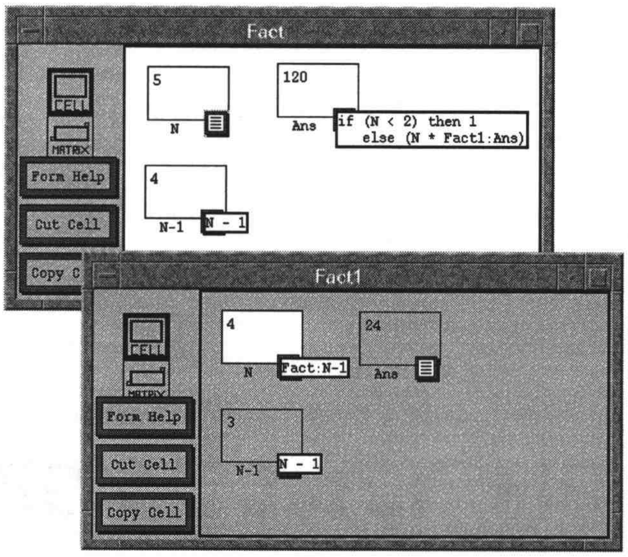
We will use Forms/3 for Understandability and Scalability benchmarks. Because Representation Design Benchmarks are intended to help in the process of design, we will show how the designs of Forms/3's navigable static representation was improved as the result of using the benchmarks. We will designate the representation of Forms/3 before using the benchmarks as Design 1, and the new design that we created with the help of the benchmarks as Design 2. All the Design 1 figures are screen shots from the actual implementation, and the figures of Design 2 as it emerges through use of the benchmarks are hand-constructed sketches.

4.4.1. Understandabilty Benchmarks

In this section, we will provide examples of applying the Understandability benchmarks using Forms/3.

4.4.1.1. Visibility Of Dependencies

We will first look at a Forms/3 program to determine the kinds of dependencies in Forms/3. Figure 4-2 shows how Design 1 representation scheme represents a recursive solution to the factorial function. The cells' formulas are shown in a text box at the bottom of the cell. The prototypical formula "5" has been specified for cell *N* on form Fact so that the programmer can receive concrete feedback. The solution involves two forms: one form that computes the factorial of the desired *N* and another, similar form that computes the factorial of *N-1*. The formula for cell Ans has been generalized in order to produce the result but its formula is shown in its concrete form. We will address how to represent the generalized formula in Section 4.4.1.3.



```
if (N < 2) then 1
   else (N * Fact1:Ans)
```

```
N - 1
```

```
Fact:N-1
```

Figure 4-2. Forms/3 Design 1 screen snapshot: Program to compute the factorial function with selected formulas shown. Instances (gray shaded) inherit their model's cells and formulas unless the programmer explicitly provides a different formula for a cell on an instantiation, in which case the cell background is shown in white, such as for Fact1's N.

We performed the visibility of dependencies benchmarks on Forms/3's representation Design 1 and Design 2. There are two bi-directional sources of dependencies in the Forms/3 language itself: dependencies due to formulas, and dependencies due to copying a model form. For example, in the program in Figure 4-2, the formula for *N-1* on Fact defines a formula-based dependency between cell *N* and cell *N-1* on Fact. Fact1's *N-1* cell is dependent on Fact's *N-1* by virtue of the fact that Fact1 was copied from the model form Fact. (Since later changes to the model Fact automatically propagate to the instantiations—except for formulas that the programmer has explicitly changed on the instantiation—this is an important dependency in Forms/3.) Multiplying these two bi-directional sources by two gives four uni-directional sources of dependencies.

In Design 1, one direction of copy-based dependencies is shown in the name of copied forms, which include the name of the model. This allows the programmer to answer the question "changes on what (model) form will change form Fact1?" directly from the name "Fact1." But the other direction is not shown; to answer the question "if I change form Fact, what copies are there that will be changed?", the programmer must manually search for forms whose names start with "Fact."

Regarding formula-based dependencies, Design 1 explicitly depicts only about half of one direction: the direct dependencies only. For example, cell *Ans* at the upper right of Figure 4-2 explicitly shows what cells directly affect the result of cell *Ans*, but does not explicitly show the indirect effects of Fact's *N-1* on Fact1's *Ans*; to find out, the programmer would have to search through the program. It does not show the other direction at all. For example, it does not explicitly show what cells are affected by the result of *Ans*; once again, the programmer would have to search through the program to find out. We were somewhat startled to see from this benchmark that, despite their popularity, such spreadsheet-like formula displays are a rather impoverished depiction of formula-based dependency information—even when all the formulas are displayed together on the screen.

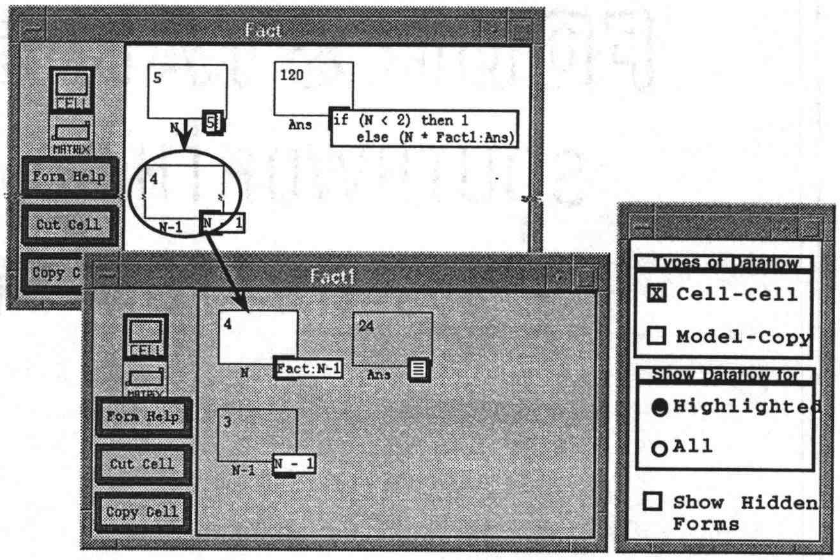
Dividing the total of the numerators by four (the number of unidirectional sources of dependencies) gives 1.5/4 = 0.375 for benchmark D1. D2 measures steps to navigate to that information or to bring it all onto the physical display screen. To add a cell's formula to the display, a programmer pulls down a cell's formula tab and selects it. This is one step per cell, or a total of n steps to add all the cells' formulas to the display, where n is the number of cells in the program.

Mapping these measurements to a subjective rating scale is done by individual designers according to the design goals of their language. We used the rating scale in Appendix B. They interpreted both D1 and D2 to be roughly "fair" according to the scale.

For Design 2, we devised improvements to increase the sources of dependencies shown (reflected by D1) and reduce the number of steps needed to do so (reflected by D2). In Design 2, dependencies can be shown explicitly by dataflow lines superimposed on forms and cells, as shown in Figure 4-3. The programmer can tailor the amount of information included in the display via the control panel. With this design, D1 results in

4/4 = 1.0 when all possible information is displayed. D2 is the number of steps to include the desired dataflow lines in the representation, including the steps needed to interact with the control panel. It takes one step per cell to include the desired dataflow lines if done cell-by-cell, or optionally the programmer can include the lines for all cells in one step and then deselect cells one by one if desired. Thus no more than n/2 steps are required to include the dataflow lines for all desired cells, plus one to two steps to interact with the control panel. This is roughly half the number of steps that were needed by Design 1. (The steps required to also display the formulas for each cell are not considered for Design 2 because dataflow lines alone are sufficient to show the dependencies. However, formulas are needed to understand the program logic, as will be discussed in the visibility of program logic section.)

Thus, representation Design 2 makes all the dependencies visible, but there is a cost—Design 2 occupies more real estate and may add clutter. This is the first of many such occurrences of this problem: if a designer adds features to the representation in order to solve deficiencies exposed by one benchmark, he or she may generate new problems that will be reflected in other benchmarks. Since this is characteristic of the process of design, it is not surprising that it is present in the benchmarks. In particular, many of these trade-offs are reflected in the Scalability benchmarks.



```
if (N < 2) then 1
   else (N * Fact1:Ans)
```

Types of Dataflow
☒ Cell-Cell
☐ Model-Copy

Show Dataflow for
● Highlighted
○ All

☐ Show Hidden Forms

Figure 4-3. The design changes represented by Forms/3's Design 2 (shown via hand-drawn additions to the current implementation). Dataflow lines are superimposed on the cells. The rightmost window is the control panel. The programmer can select more than one cell at a time, but in this example, only cell N-1 *was selected. There is also an option on the control panel to show all the dependencies.*

4.4.1.2. Visibility Of Program Structure

In Forms/3 a module is a form, and Design 1 does not explicitly show how the forms relate to one another. Nor does the dataflow wiring added in the previous section explicitly show program structure, because it is too fine-grained—the programmer still must search the diagram manually, looking for sources and sinks, to detect the overall structure.

We decided to add an optional view of the hierarchical dataflow between forms (Figure 4-4). This representation is based on the *form collapsed multi-graph*, a variant of dataflow graphs that is useful for describing the relationships among related forms that was described in Chapter 3. We elected to use this vehicle to depict not only program structure but also optional fine-grained details in the context of program structure as follows. The default is for all forms except those containing sources and/or sinks to be represented as collapsed icons, but the programmer can override this to display details of the collapsed icons as well. The sources and sinks are the beginning and the end of the dataflow path, which are circled in the figure. With this addition, Design 2's PS1 benchmark is "yes", and benchmark PS2 is 1 (it requires one step to add the program view to the physical screen via a button on the main control panel).

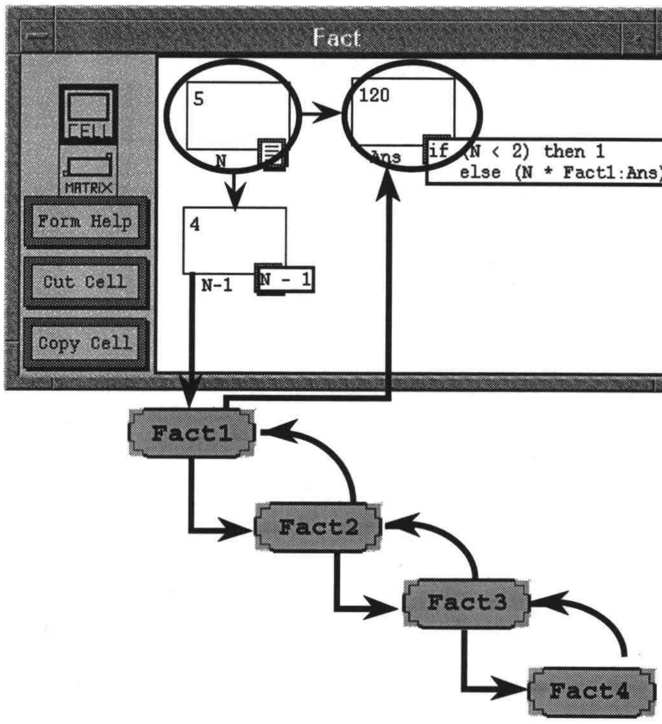


Figure 4-4. Forms/3 Design 2's program structure view of the factorial function. The source and sink of the dataflow are circled. Those forms that do not contain sources or sinks are shown as collapsed icons.

4.4.1.3. Visibility Of Program Logic

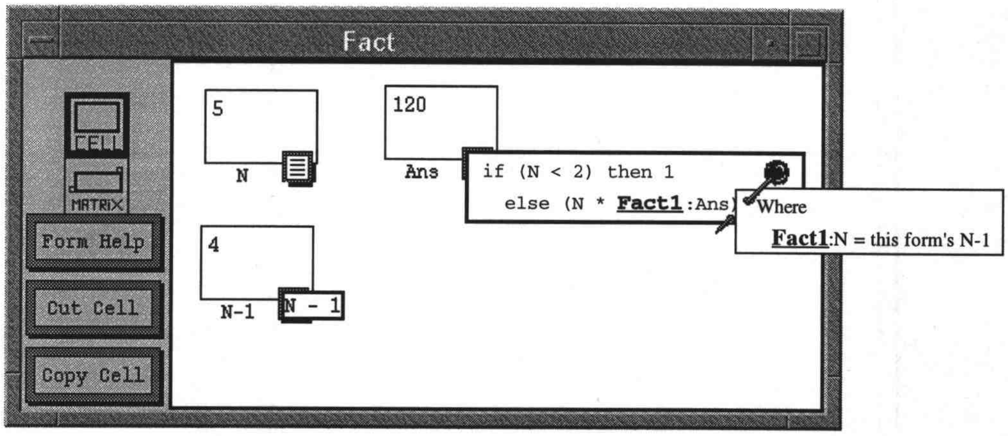
In Forms/3, the program logic is entirely specified by the cell formulas. However, unlike spreadsheets, as many formulas as desired can be displayed on the screen simultaneously with the cell values. In Design 1, a programmer can pull down a formula tab and select the displayed formula to cause it to remain permanently on display; thus L1 = "yes". It takes one step per cell to include a formula, for a total of n steps to include all the formulas for benchmark L2.

We decided to reduce the number of steps reflected by L2, because for large programs, making n cells' formulas visible would be burdensome. Design 2 adds a "show all" and a "hide all" option to the NI to reduce the number of steps. Since it takes one step to toggle the options on the control panel, this allows all formulas to be displayed in only 1 step, and allows any subset of the program to be displayed in no more than n/2 steps. This change reduced the number of steps by half.

To compute L3, the designer counts the sources of misrepresentations of generality. To be able to represent the generalized abstractions was the motivation behind the work of Representation Design Benchmarks. Recall from Figure 4-2, Forms/3's Design 1 contains one such source of misrepresentation, namely the use of concrete examples to identify form instantiations. For example, the formula of cell *Ans* on form Fact appears as "if (N<2) then 1 else (N * Fact1:Ans)", which seems to refer to the particular instantiation Fact1 (which computes 4 factorial); however, in actuality the formula refers to a generic instantiation of Fact whose computations are defined relative to the value of the *N-1* cell on the referring form. In Design 2, we modified the static representation as in Figure 4-5 to correct this misrepresentation in the following ways:

a) Visually distinguish formulas containing concrete examples versus generalized references by underlining the concrete form ids.

b) Add a legend to describe the generalized meaning that was derived from the concrete specification.



```
if (N < 2) then 1
  else (N * Fact1:Ans)
```

Where
Fact1:N = this form's N-1

Figure 4-5. Forms/3 Design 2: The factorial function with legend. The bold and underlined form name Fact1 indicates that the concrete form name is just an example of a more general relationship. Clicking on this name causes a legend to be attached to the formula display explaining the generalized relationship between this form and the form represented by Fact1, namely that this form's N-1 is what N on the latter form uses as its formula.

4.4.1.4. Display Of Results With Program Logic

In Forms/3's Design 1, each partial program result (cell value) is automatically displayed for each cell next to its formula (or by itself if the programmer has not chosen to leave the formula on display). Thus R1="yes" and, since no action is needed to navigate to these partial results, R2=0. We considered these Design 1 scores to be excellent, and made no changes in Design 2.

4.4.1.5. Secondary Notation: Non-Semantic Devices

Forms/3's Design 1 includes all of these notational devices. Textual annotations and graphical annotations can be anywhere on a form. Layout is also entirely flexible, which allows non-semantic spatial grouping of related cells, etc. Cell names are optional but are often provided by programmers, because use of meaningful names provides additional non-semantic information. Thus SN1 = 4/4 = 1.0. The number of steps required to navigate to the secondary notations, SN2, is zero because these secondary notations are always automatically visible.

4.4.2. Scalability Benchmarks

In this section, we will provide examples of applying the Scalability benchmarks using Forms/3.

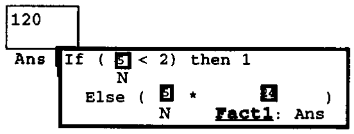
4.4.2.1. Abstraction Gradient

Forms/3's strong emphasis on abstraction was reflected in the Design 1 benchmark scores for this group. In Design 1, forms can be collapsed into a name or into an icon. Data structures can also be collapsed into graphical images. Cells can be made hidden, which excludes them from the representation. Control panels that are part of the NI can be collapsed into icons. Thus, the AG1 score is 4/4 = 1.0, reflecting the fact that in Forms/3's Design 1 there is no source of detail that cannot be abstracted away. This score is also true of the Design 2 features that have been described in this paper. Turning to AG2, the number of steps required to collapse a form or a control panel is 1. The amount of detail shown for data structures and for hidden cells is automatically controlled without any programmer interaction through automatic maintenance of the information-hiding constraints of Forms/3 (0 steps). The programmer may override this automatic behavior when desired at a cost of 1 step per form (n/c steps per program, where c is a constant representing the average number of cells on a form).

4.4.2.2. Accessibility Of Related Information

In Forms/3's Design 1, it is possible to view related cells side by side (RI1="yes"). A cell can be dragged around on a form as needed; most of the navigational effort arises in moving the needed forms near each other on the screen. One way is by double-clicking on the form's icon if it is visible, but this can involve manually moving things around to look for the icon. A less ad-hoc way is by scrolling to the form's name in the control panel's list of forms and clicking the "display" button, which brings the selected form into a visible portion of the screen. Thus, counting the time to scroll through the list, RI2 can approach the square of the number of forms in the program, or $(n/c)^2$, where n is the number of cells in the program and c is the average number of cells per form.

At first, it appeared that the dataflow lines that had been added to Design 2 might altogether eliminate the need for programmers to do this searching. However, it soon became apparent that dataflow lines do not eliminate the need to search if the lines are long. We decided to make changes in both S and NI for Design 2 to reduce the number of steps to search. The change in S is to include the *value* of all referenced cells in a formula, as in Figure 4-6, so that if the programmer is merely interested in how the values contribute to the new answer no searching at all is required. The change in NI is that if the related cell is on a different form, clicking on the cell reference in the formula will automatically bring the form up on the representation. This navigation mechanism reduces the worst-case score of RI2 to one step per form, for a maximum of n/c steps.



```
120

Ans  If ( 5 < 2) then 1
          N
       Else ( 5  *        24      )
              N       Fact1: Ans
```

Figure 4-6. Forms/3 Design 2: The values are displayed with the cells referenced in the formula. This eliminates the need for a programmer to search for these cells to find out their current values contributing to the value of Ans.

4.4.2.3. Use Of Screen Real Estate

Returning to the Forms/3 example, the program elements are the cells. In performing SRE1 and SRE2 for Design 1, we decided to measure Forms/3 in a layout strategy in which SRE2 would be minimized, measuring the maximum number of cells that would fit on the screen in the absence of any non-semantic intersections. Approximating with an average cell size and formula length, the maximum number of cells that fit on the physical screen of a Unix-based graphical workstation or X-terminal with no non-semantic intersections is 36 when all formulas (and values) are shown. This is approximately 54% of the amount of source code that would be shown in a full-length window (66 lines) for a textual language. However, the Forms/3 display also includes all the intermediate values and final outputs, which in the textual language would require adding a debugger window and a window to show the execution's final results. This score points out that a strength of this cell-based representation is that it is a reasonably compact way to combine a presentation of source code, intermediate results, and final outputs, while still avoiding non-semantic intersections.

The space and non-semantic intersection costs of the design features in Design 2 are compared with Design 1 individually and in combination in Table 4-2. Not surprisingly, Design 1 allows more program elements to fit on the screen with fewer intersections than Design 2, because Design 1 contains less information than Design 2. This is an example of the trade-offs these benchmarks help bring out. We decided that the space and intersection costs of Design 2 were acceptable because the navigational instrumentation portion of Design 2 allows the programmer to be the judge of these trade-offs, including or excluding from the screen as many of the Design 2 features as desired.

Table 4-2. Design trade-offs in screen real estate.

Design Options	**SRE1 (units=cells)**	**SRE2 (units=intersections)**
Base: Design 1, all formulas showing	36	0
Design 1 + dataflow lines (if request is for a small number of selected cells)	no change: 36	a $(a \geq 0)$
Design 1 + dataflow lines (if request is for all cells)	no change: 36	b $(b \geq a \geq 0)$ These intersections are a superset of the a intersections in the previous row.
Design 1 + program structure view	approximately 20% fewer: 29	c $(b \geq c \geq 0)$ These lines are a more coarse-grained view of the dataflow lines in the previous row.

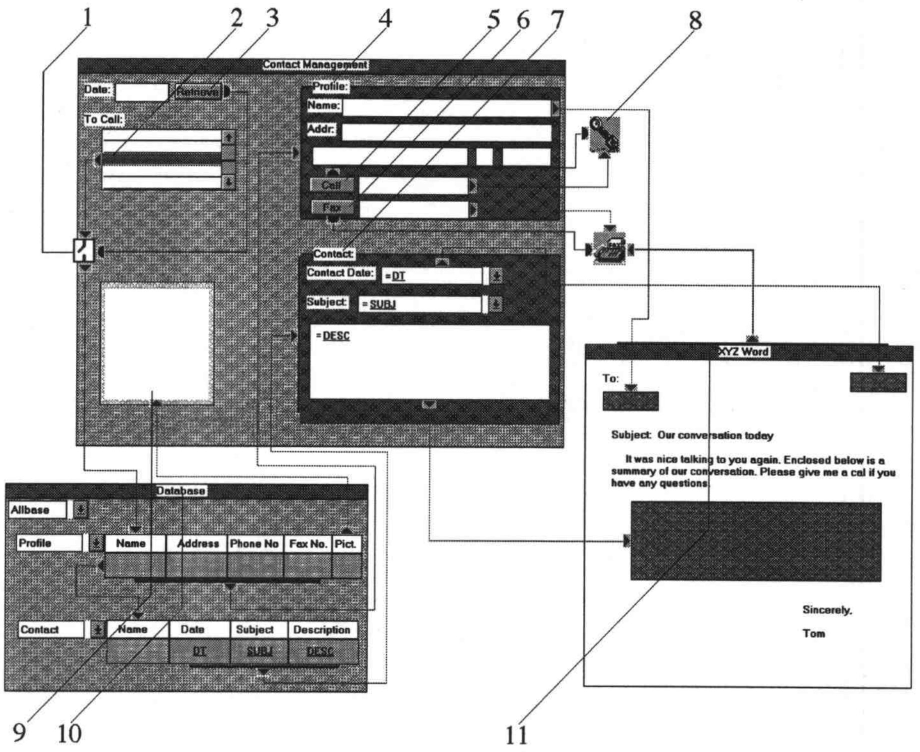
Table 4-2. Continued.

Design Options	SRE1 (units=cells)	SRE2 (units=intersections)
Design 1 + legends	approximately 1 fewer per legend: 18 if each cell has 1 legend displayed	0
Design 1 + cell icons in formulas	approximately 20% fewer: 29	0
Design 2 (All features)	approximately 40% fewer: 22	b

Table 4-2. Design trade-offs in screen real estate. Trade-offs between features added to save the programmer time versus their real-estate space costs become apparent in this comparison of the real estate costs of the Forms/3 Design 2 features. This table shows Design 1 in the top row, Design 1 supplemented by each individual feature of Design 2 starting in the second row, and finally all of Design 2 together in the last row. When there were trade-offs between SRE1 and SRE2, we used layouts that optimized SRE2 in performing these benchmarks. The variables a, b, and c represent numbers of line crossings, and their values vary with the actual dependencies in each program. Since the lines are not necessarily straight, there is no upper bound on the values of these variables other than in their relationships with each other.

4.4.3. Audience-Specific Benchmarks

For concrete examples of applying the audience-specific benchmarks, we will turn to the audience-specific language ICBE (Interpretation and Customization By Example). ICBE is a set-oriented dataflow VPL with a strong emphasis on interoperations between systems—such as database, spreadsheets, and graphics—aimed at end-user professionals. Its goal is to allow such users to create custom applications by combining GUI objects, built-in capabilities such as database querying, plug-in objects such as virtual fax machines and telephones, and interoperations between other applications such as spreadsheets and graphics packages. Programming in ICBE is a matter of simply connecting these objects using dataflow and control-flow lines. See Figure 4-7 for an example. ICBE is a generalization of the kind of declarative by-example programming used in QBE and OBE [Zloof 1977; Zloof 1981]; a more complete description of ICBE can be found in [Zloof and Krishnamurthy 1994; Krishnamurthy and Zloof 1995].



Subject: Our conversation today

It was nice talking to you again. Enclosed below is a summary of our conversation. Please give me a cal if you have any questions.

Sincerely.

Tom

Figure 4-7. A salesperson is creating a program for a contact management application in ICBE. To make a call, the salesperson will highlight a customer (2) in the "To Call" list and press the "Retrieve" button (3). This will close the gate (1) and thereby complete the circuit, allowing the highlighted list entry to flow into the table (10). This completes the selection criterion for the query, which results in retrieval of the customer's picture (9), profile (4), and contact data (7). If the salesperson pushes the "Call" button (5), the customer's phone number will be dialed automatically by the Telephone plug-in object (8). If the salesperson integrates a word processing document into the system (11), it can be faxed to the customer by pushing the "Fax" button (6).

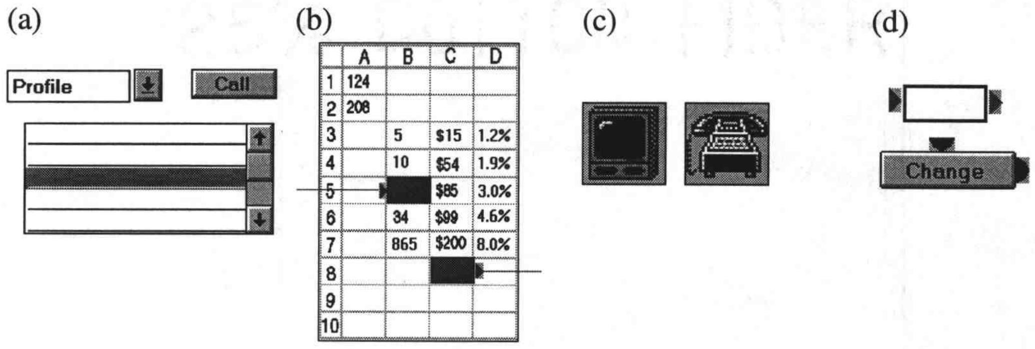
4.4.3.1. ICBE's Intended Audience

To apply the audience-specific benchmarks to ICBE, the first step is to identify the intended audience in a precise enough fashion that the intended audience's prerequisite background becomes clear. ICBE is intended to be used by "power users": users who are already competent in general office applications, such as spreadsheets, HyperCard-like systems, and e-mail. (However, there is no assumption that ICBE users can use the most advanced capabilities of these systems; for example, ICBE users are not assumed to be able to create spreadsheet macros, program textually in HyperTalk, or write shell scripts or

.BAT files.) Examples of such users might include salespeople, administrators, and accountants.

4.4.3.2. Benchmark AS1: The Objects

The objects in ICBE are user interface primitives, interoperation objects, external plug-in objects, and flow ports. Examples of each are shown in Figure 4-8. The user interface primitives include objects such as text fields, buttons, and lists. Interoperation objects include such external applications as spreadsheets, databases, and business graphics packages, and are represented by grids, tables, and graphs. External plug-in objects, which appear as icons, are vendor-supplied objects that can be added to the system to expand its capabilities. Instances of the fourth kind of object, flow ports, are shown as arrows, and are attached to the other three kinds of objects to specify the direction (incoming or outgoing) of the dataflow and control flow.



	A	B	C	D
1	124			
2	208			
3		5	$15	1.2%
4		10	$54	1.9%
5			$85	3.0%
6		34	$99	4.6%
7		865	$200	8.0%
8				
9				
10				

Figure 4-8. (a) Some ICBE user interface primitives. (b) A grid represents a spreadsheet, which is an example of an interoperation object. (c) Television and telephone plug-in objects. (d) Arrows represent ports: the red (pointed) arrows are dataflow ports, and the blue (rounded) arrows are control flow ports.

To perform the AS1 benchmark, the ICBE design team answered the following questions (one for each object):

ObQ1: Do the user interface primitives look like the user interface objects in the intended audience's prerequisite background?

ObQ2: Do the representations of the interoperation objects (such as grids, tables, and graphs) look like the spreadsheets, databases, and graphics packages in the intended audience's prerequisite background?

ObQ3: Do the plug-in objects' icons look like the corresponding objects in the intended audience's prerequisite background?

ObQ4: Do the arrows look like incoming and outgoing information ports in the intended audience's prerequisite background?

The ICBE design team answered "yes" for ObQ1, ObQ2, and ObQ3. The "yes" answers to ObQ1 and ObQ2 are because the ICBE user interface primitives and interoperation objects look like user interface objects and miniaturized windows from common office packages, which are expected as part of these power users' prerequisite backgrounds. ObQ3's "yes" is actually "potentially yes," since the answer depends on the external vendors' icon design skills.

The ICBE design team answered "no" for ObQ4. Although arrows are common indicators of directionality, there is nothing in ICBE users' prerequisite backgrounds to suggest that arrows would look like information exchange ports to ICBE's power users. (However, this representation might look like information exchange ports to a different audience, such as professional programmers, because it is commonly seen in CASE tools and component-building software aimed at professional programmers.) Additionally, the two styles of arrows, pointed and rounded, do not look particularly like data directionality as versus control directionality.

The total AS1 score is thus 3/4 = 0.75; that is, 3/4 of the objects in the representation look like objects from the intended audience's prerequisite background. This high score reflects the emphasis placed by the ICBE designers on gearing their language directly to this audience. The ICBE designers rated this score well, but they also decided as a result of the benchmark to study their potential audience's ability to understand the two different kinds of ports, to see if a different representation is needed for them.

4.4.3.3. Benchmark AS2: The Operations

The six operations in ICBE are dataflow, event-based control flow, transfer of control (call or goto constructs), interruption of dataflow, event triggers, and selection over a list or a database. See Figure 4-9. Dataflow (shown via red lines) is the directed flow of data through the objects in the system. Event-based control flow (shown via blue lines) allows the occurrences of events, such as button clicks or key presses, to generate program activity. The call and goto constructs transfer control to another part of the program, and as a variant of control flow are also shown via blue lines. Dataflow can be interrupted if there is an open gate in the path. Triggers in ICBE, depicted with gun icons, are used to generate events internally, usually because a particular data condition has arisen. (For instance, a trigger can be attached to a database of customer accounts to monitor delinquent customer accounts. When such a customer is encountered, a trigger can cause a warning dialog to appear.) Query sliders and decision tables allow specification of the data selection criteria over a list or a database.

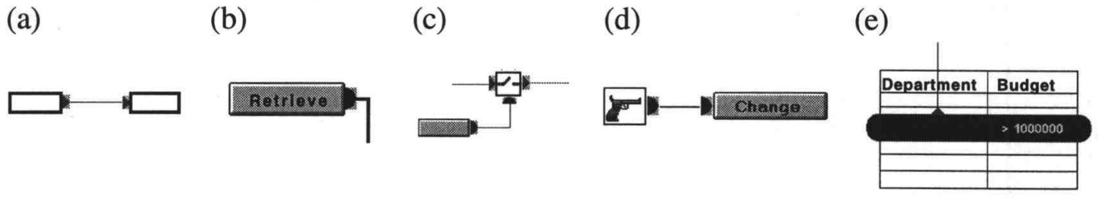


Figure 4-9. Some ICBE operations. (a) Dataflow. (b) Event-based control flow, initiated by pressing the Retrieve button. Control flow for transfer of control is also shown via these blue lines. (c) An open gate interrupts dataflow. (d) A trigger causes the change button to be "pushed" automatically. (e) A query slider is a data selection operator.

The AS2 benchmark for these six operations requires answering the following six questions:

OpQ1: Does the (red) line look like a conduit for the flow of data in the intended audience's prerequisite background?

OpQ2: Does the (blue) line look like a conduit for event-based control in the intended audience's prerequisite background?

OpQ3: Does the (blue) line look like a conduit for the transfer of control in the intended audience's prerequisite background?

OpQ4: Do the open gates look like a way to interrupt dataflow in the intended audience's prerequisite background?

OpQ5: Does the gun trigger look like a mechanism for triggering events in the intended audience's prerequisite background?

OpQ6: Do the decision tables and query sliders look like mechanisms for data selection over a database or a list in the intended audience's prerequisite background?

The ICBE designers answered "yes" for OpQ1, because the red lines, which are connected to the arrow objects discussed earlier, look similar to widely-understood conduits for directed flow such as water pipes or map representations of one-way streets. They also answered "yes" for OpQ2, because the blue lines look and behave the same way as electrical wires.3 Regarding OpQ3, the designers noted that using the same blue line to indicate transfer of control overloads this device in the representation. However, this does not impact AS2's score; rather it would be reflected in the score for Benchmark L1 (Visibility of Program Logic). For AS2's OpQ3, while lines for transferring control may be familiar to professional programmers and others who have seen flowcharts, they do not resemble anything from prerequisite backgrounds of ICBE's intended audience, and earned a "no" answer. Interrupting potential flow by opening a gate to disconnect the lines looks like a mechanism that would interrupt the flow of water or traffic, and earned a "yes" for OpQ4. The ICBE designers gave questions OpQ5 and OpQ6 "no"s because, although both of these devices might be familiar to programmers or engineers, they do not necessarily look like devices ICBE's intended audience has seen before. Adding up the numerators and dividing by 6 gives an AS2 score of 3/6 = 0.50.

4.4.3.4. Benchmark AS3: Spatial Composition

The spatial composition of elements of a language's representation is the way they are arranged and connected on the screen. Especially for programs simulating some physical environment, this aspect of a representation can have a strong influence on how closely the representation matches the way the problem appears in the audience's prerequisite background. In ICBE's representation, there are four ways objects and/or operations can be spatially composed: by their layout, by their connections with lines, by their placement into containers as a grouping mechanism, and by nesting containers within other containers as a constrained grouping mechanism. Figure 4-7 shows one example of layout with several examples of line connections, and Figure 4-10 shows a container nested within another container.

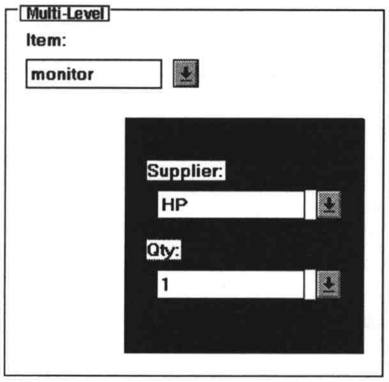


Figure 4-10. ICBE containers. The blue inner container combines a supplier list, list of possible quantities, and textual labels for each. The outer container in turn combines the inner container with an item list and textual label. The nesting implies a constrained relationship; for example, if the value "monitor" were "keyboard" instead, the contents of the nested container would reflect the supplier and quantity of the keyboard order.

To measure whether the spatial composition mechanisms in the representation mimic the way the objects and operations fit together in the intended audience's prerequisite background, the ICBE designers answered the following questions:

SQ1: Does the layout of the objects and operations look like the way these objects and operations are laid out in the intended audience's prerequisite background?

SQ2: Do the lines connecting the objects and operations look like the way these objects and operations are connected in the intended audience's prerequisite background?

SQ3: Does the container look like a way of grouping objects in the intended audience's prerequisite background?

SQ4: Does the nested container look like a way groupings are nested in the intended audience's prerequisite background?

ICBE designers answered "yes" for the first three questions and "no" for the fourth. The SQ1 "yes" is somewhat qualified, because it depends on how the user chooses to lay out a program. ICBE's problem domain is not restricted to a particular kind of simulation, and thus there is no automatic layout mimicking a particular physical environment; however, because ICBE allows complete flexibility in laying out objects and operations on the screen, the user can match a physical layout if desired. The answers to SQ2 and SQ3 are more obvious: Lines are well-known ways of connecting objects, and even operations, in many office, project-management, and organization chart applications, and as such are part of these power users' prerequisite backgrounds. Putting objects into containers (jars, shopping bags, etc.) is a grouping mechanism from everyday life. The ICBE designers' "no" answer for SQ4 was a borderline case. Nested containers do indeed look like the way groupings are nested in everyday life, but the constraining aspect of nesting a container does not exist in these everyday-life nestings. Thus their character is sufficiently different from ICBE nested containers that the designers decided on a "no" answer. The AS3 score is therefore 3/4 = 0.75; that is, three of the four spatial composition mechanisms look like corresponding mechanisms in the audience's prerequisite background.

4.5. Outcomes Of Using The Representation Design Benchmarks

In this section, we will describe the direct outcome of using the Representation Design Benchmarks for Forms/3 and ICBE.

4.5.1. Outcomes Of The Benchmarks For Forms/3

Using the Representation Design Benchmarks helped us to design a representation for the generalized formulas as well as identified several problems in the representation. The legend representation for the generalized formula and all the other new representation features of Design 2 have since been implemented, except the proposed program structure view in Section 4.4.1.2. Figure 4-11 - 4-13 show screen shots of the new representations.

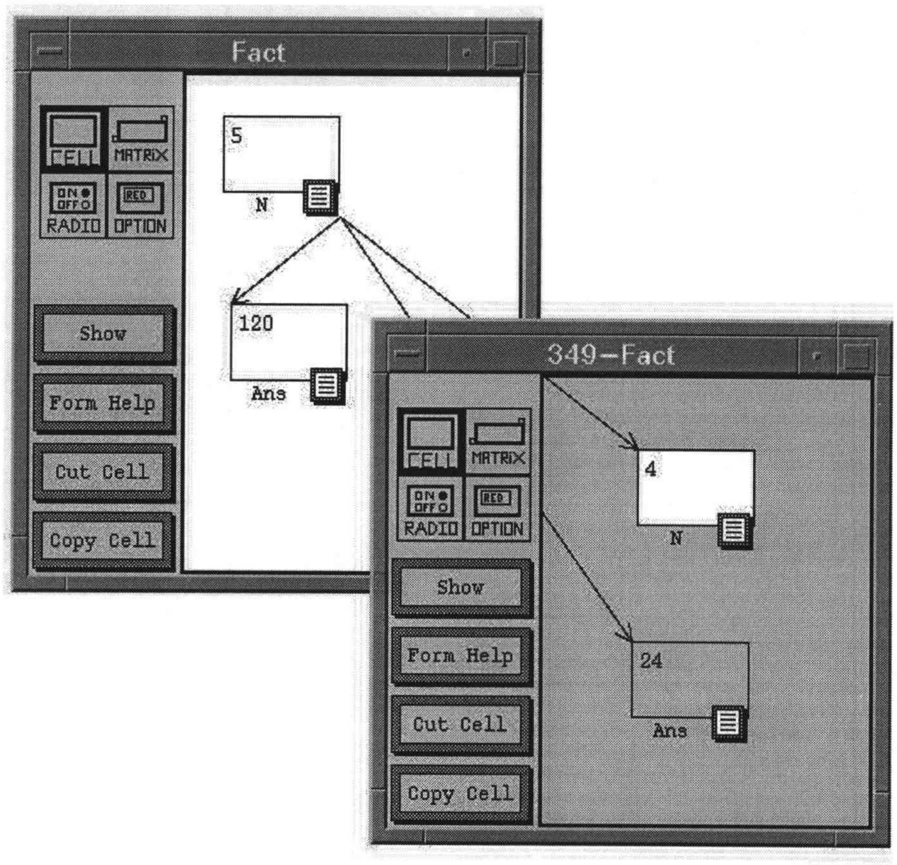
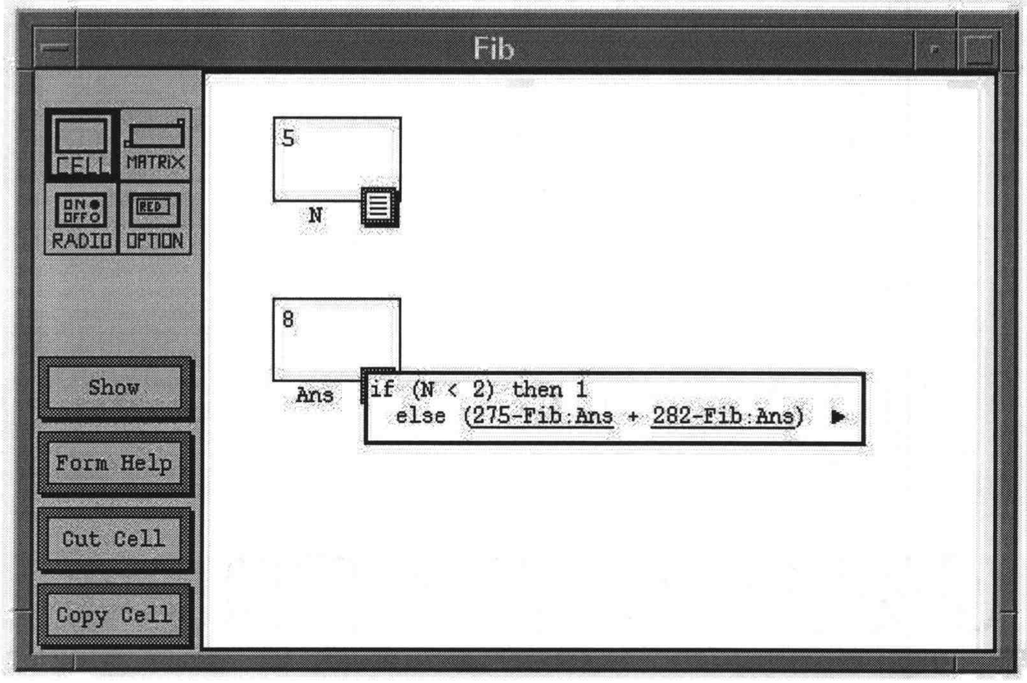
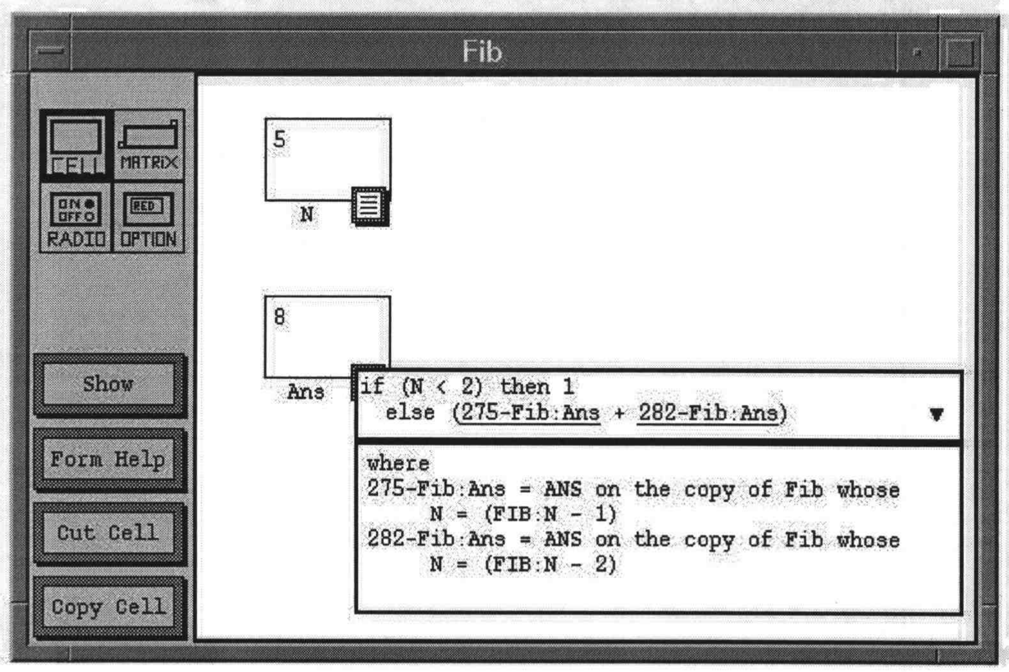


Figure 4-11. The Factorial program with dataflow arrows superimposed on the cells to explicitly show the cell dependencies.



```
if (N < 2) then 1
  else (275-Fib:Ans + 282-Fib:Ans)
```

Figure 4-12a. The Fibonacci program showing the underlined concrete form names.



```
if (N < 2) then 1
  else (275-Fib:Ans + 282-Fib:Ans)

where
275-Fib:Ans = ANS on the copy of Fib whose
      N = (FIB:N - 1)
282-Fib:Ans = ANS on the copy of Fib whose
      N = (FIB:N - 2)
```

Figure 4-12b. The Fibonacci program showing the expanded legend describing how the forms were constructed.

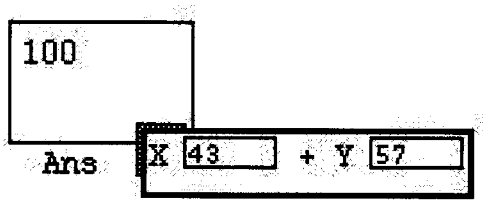


Figure 4-13. Values of X and Y are displayed with the formula.

4.5.2. Outcomes Of The Benchmarks For ICBE

The ICBE designers found that using the Representation Design Benchmarks identified previously-unnoticed issues in the representation. For example, the AS1 audience-specific benchmark pointed to a possible need for a new port representation. Also, audience-specific benchmarks AS2 and AS3 pointed out the fact that some of the representation elements, while they are very likely familiar to programmers or engineers, are not necessarily familiar to the intended audience for ICBE. For the representation elements with "no" answers, the next logical step is audience testing to determine whether the lack of familiarity to this audience of these particular elements will affect ICBE's long-term usability; that is, whether or not these particular representation elements can be learned easily by ICBE's intended audience after seeing the language in action.

4.6. An Empirical Study Of VPL Designers

In considering the usefulness of the Representation Design Benchmarks to designers, the following question arises: Does using the Representation Design Benchmarks in the design process actually produce better representations? Unfortunately, empirically arriving at the answer to this question is probably not feasible. Such a study would require evaluating many VPLs with dual implementations, one implementation of each VPL corresponding to a design created without the use of the benchmarks, and the other corresponding to the design created with the use of the benchmarks. The two implementations of each language would have to be empirically compared for their usefulness to programmers. The primary difficulty with such a study would be finding several different VPL design teams willing to expend the effort to design and implement dual versions of their representations.

However, useful insights can be gained about this question by considering two related questions that are more tractable for objective analysis:

1) How usable are the Representation Design Benchmarks by VPL designers?
2) Does using the Representation Design Benchmarks in the design process uncover problems and issues that would otherwise be overlooked?

To learn more about the answers to these two questions, we conducted a small empirical study with two goals. The first goal (Goal 1) was to uncover problems VPL designers might have in using the benchmarks. The second goal (Goal 2) was to learn whether VPL designers other than ourselves could use the benchmarks, and whether their doing so would be useful in uncovering problems in their designs of navigable static representations. The hypothesis to be tested for this second goal was that the subjects would be able to use the benchmarks and would find at least one problem and make at least one change, addition, or deletion to their representation designs as a direct result of using the Representation Design Benchmarks. The study was very helpful regarding Goal 1, and the Goal 2 results were favorable about the usefulness of the benchmarks to VPL designers.

4.6.1. The Subjects

The subjects for the study needed to be VPL designers who were in the midst of designing a VPL representation. Such subjects would normally be hard to find, but we timed the study so that we could recruit the five Computer Science graduate students who were in the process of designing VPLs (and navigable static representations for them) for a graduate course. Recent studies of usability testing show that five test participants are sufficient to uncover approximately 80% of the usability problems [Virzi 1992]. (Virzi also reports that additional subjects are less and less likely to reveal new information.) Thus, this was a reasonable number of subjects for addressing our first goal, finding the usability problems that we and the ICBE design team had missed. We would have liked a larger number of subjects for our second goal, learning whether the benchmarks were useful to VPL designers. However, this sample size is fairly typical of studies relating to non-traditional programming languages, due to the difficulties in finding suitable subjects for them.[1]

4.6.2. The Procedure

The subjects were already in the process of designing a small VPL. To test our Goal 2 hypothesis, we chose a within-subject experimental design with a before-benchmarks design task and a during-benchmarks design task. These tasks also provided information we needed to achieve our first goal, finding usability problems.

4.6.2.1. Before Using The Benchmarks

The subjects' before-benchmarks task was to submit a design of all viewable aspects of their VPLs. This task served two purposes: it provided the baseline data about the designs created without the benchmarks, and it served as a training function to help them understand what a navigable static representation was.

Because one purpose of this training task was for the collection of baseline data, it was important to make sure that the subjects' reporting of their designs was complete, i.e., that they would not omit important information through misunderstandings about what was part of the navigable static representation. We avoided this potential problem by having

[1]See, for example, the study of the VPL LabView (5 subjects) [Green et al. 1991], the study of the VPL ChemTrains (6 subjects) [BellB et al. 1991], and the study of a generic Petri-net language (12 subjects) [Moher et al. 1993].

them include *everything* viewable in this task. The training purpose was accomplished by having the subjects classify the elements of the design in three categories: the static representation S, the navigational instrumentation NI, and dynamic representations used in the VPL not in NI or S, such as animation, balloon help, etc. They then received feedback about the correctness of their classifications. To give them an incentive to do their best at devising a good representation without the use of the benchmarks, the task was set up as a graded project. The subjects were given one week to perform the task.

The students had been gradually prepared for this task during the term. Throughout the course, they had been reading papers about VPLs, writing programs in a variety of VPLs, and discussing the research problems associated with VPLs, including static representation. Just before they were asked to perform the task, we defined what a navigable static representation was and motivated its importance, but we did not introduce the benchmarks.

4.6.2.2. During Use Of The Benchmarks

After the first task was completed, the subjects were given a lecture on Representation Design Benchmarks. They were then asked to perform the second task, which was to measure the navigable static representation part of their VPL's design using the benchmarks, being allowed to make any modifications they thought necessary. The purposes of this task were to find usability problems with using the individual benchmarks (Goal 1) and to test our hypothesis about whether they would be able to use the benchmarks and in doing so would find any problems and make any changes to their designs as a result of using the benchmarks (Goal 2).

The subjects were instructed to measure their designs as follows. They were to start with their representation design as of the end of the previous task. They were then to measure it using the benchmarks. If the outcome of any benchmark pointed out problems to them, they were permitted to change the design to solve the problem, and then remeasure. (During the same period, the students were designing their term-project VPLs.) The subjects turned in the results of the during-benchmarks task two weeks after the assignment was made. For purposes of motivation, it too was a graded assignment, where the grade was based on the quality of their designs.

Grading this task raised the question of what set of grading criteria would define whether they had designed a "good" representation. We decided to follow the sample mapping from measurements to ratings shown in Appendix B. This meant that the grades would be determined by whether a design's benchmarks mapped into mostly "good" ratings. To avoid prejudicing the results by forcing design changes via these grading criteria, only ratings for *those benchmarks that were deemed important by the subject for that particular VPL* were included in the grading criteria. Any benchmark could be eliminated if the subject explained why it was not an important measure, given their language's goals.

The subjects turned in their completed representation design and the rating information. When they turned in this information, they were given time in class to list any problems they had using the benchmarks and to annotate their design pointing out which, if any, changes they made as a result of using the Representation Design Benchmarks, as distinguished from changes they made for other reasons.

4.6.3. Results And Discussion: Goal 1

All of the subjects were able to complete the before-benchmarks training task, but they all had trouble categorizing the viewable elements correctly into the three categories (static, navigational aids, and dynamic). We clarified the definition of navigable static representations to partially address this problem. In addition, however, we are inclined to infer from this evidence that isolating the navigable static representation from the rest of the VPL is an academic exercise that does not come naturally for interactive VPLs, and is one that might be omitted in the absence of the benchmarks. The poor track record of static representations for interactive VPLs lends some support to this conjecture. Since we believe that this isolation is important if the designer is to obtain a clear understanding of the representation's strengths and weaknesses, we view this as one advantage of using the benchmarks.

All of the subjects also completed the during-benchmarks task, and reported the problems they had in understanding how to obtain some of the measurements. The subjects were successful with the NI benchmarks, but had some difficulties with the S_p and S_c benchmarks. (At the time of the study, the benchmarks measuring NI and the benchmarks measuring S were not explicitly separated.) Also, the screen real estate benchmarks were based upon a test suite at that time, and none of the subjects were able to perform these benchmarks with any accuracy. The subjects also suggested that the benchmarks as a whole needed to better reflect the trade-offs between adding new features to the representation versus the space and navigational effort required by these additional features.

As a result of the usability issues the subjects found in the during-benchmarks task, we made the following changes, all of which are incorporated in the benchmarks as described in this paper. First, an explicit separation was made between the NI benchmarks versus the benchmarks measuring aspects of S (S_p and S_c). We also revised the screen real estate benchmarks to measure characteristics of the representation itself rather than characteristics of test programs, and to include a measure of general space characteristics (SRE1). Finally, we added several new NI benchmarks throughout the benchmark groups to be sure the trade-offs between adding features and navigational effort imposed by those additional features were well represented.

4.6.4. Results And Discussion: Goal 2

All of the subjects reported that the Representation Design Benchmarks were useful to them. Their subjective reports were that the benchmarks helped them to think through their design more precisely, thereby focusing on problems that they had overlooked prior to using the benchmarks. The Goal 2 hypothesis was verified—all the subjects were able to complete the during-benchmarks task, and all the subjects found problems and made additions and/or changes to their designs as a direct result of using the benchmarks. Since they had previously been given incentives and time to make the best design they could (without the benchmarks), we expected that these changes made in the during-benchmarks task were as a direct result of the benchmarks. This fact was verified by their annotations on their design documents, which identified the changes resulting from using the benchmarks. The problems they found with their designs and the changes they made are summarized in Table 4-3.

Benchmark group	**Problems found and changes made by the subjects**
Dependencies	One subject found that only half of the dependencies were explicitly visible in her representation. This was fixed in her final design.
Program logic	Two subjects made changes in the representation of program logic: One subject improved the representation to make all the program logic visible. Another subject found and corrected a misrepresentation of generality in his representation.
Display of results with program logic	One subject reduced number of steps required to display the results with program logic.
Secondary notation	Two subjects made changes to the secondary notational devices available: One subject was surprised to see that her original design omitted comments; she changed her design to allow textual comments. Another subject added more devices for secondary notation.
Abstraction gradient	Two subjects added more powerful navigational devices in order to reduce the number of steps required to navigate among the levels of abstraction they supported.
Accessibility of related information	One subject added navigational aids to reduce the number of steps to access related information.
Use of screen real estate	One subject reduced the number of on-screen windows to reduce non-semantic intersections.

Table 4-3. Problems found and corrections made that resulted from using the Representation Design Benchmarks, as reported by the subjects.

4.7. Discussion: Beyond Design Time?

We have discussed the usefulness of the benchmarks as a design time aid, and have shown how they can be used to evaluate a single design and to compare several alternative design ideas. Since the notion of using benchmarks as a design aid is somewhat unusual, a question that naturally arises is whether Representation Design Benchmarks can be used in a more conventional way, such as in objective evaluations and comparisons of the representation schemes of different post-implementation VPLs.

Although we have not experimented with them for such purposes, we suspect that certain features of the Representation Design Benchmarks, which are needed for usefulness as a design-time aid, are not compatible with the features needed for objective comparisons. Recall that using the benchmarks is a tailorable process, including not only the objective step of obtaining the actual measurements, but also subjective steps such as selecting benchmarks applicable to the particular language's goals, and interpreting the implications of the resulting scores in light of the language's goals. Even the objective step has tailorability, because designers must determine exactly which features of their particular VPLs pertain to each individual benchmark in order to calculate the measurements. These kinds of flexibility are necessary to be useful to a designer for tasks such as evaluating design ideas with respect to the designer's goals, but they may introduce too much subjectivity to allow truly objective comparisons among different languages.

Another observation relevant to this issue is timing. When the designers we observed used Representation Design Benchmarks to evaluate their representation schemes after implementation, they tended to be more interested in justifying past work (and manipulating the tailorable aspects to accomplish this) than in finding ways to improve the design. This is not surprising, because after the design is completed, a conflict of interest arises—if a designer considers a design finished, there are powerful disincentives to find anything wrong with it. This observation runs in the same vein as Winograd's observation mentioned earlier, that uncovering substantive problems is more likely to occur early in the design stages than later in the lifecycle. Winograd's observation pertained to users [Winograd 1995], and our experience was that it also pertained to the designers themselves. From this we surmise that, even if it is possible to use the Representation

Design Benchmarks for non-design-oriented purposes (by a language's designers or by others), the amount of useful information obtainable from the benchmarks is still likely to be greatest during the design stage.

5. FUTURE WORK

Our research into generalizing abstractions and representing the resulting programs in static representations is not complete. Some questions remain that require further research.

Time Optimizations: One optimization to the generalization algorithm would be to reduce the time to perform cycle detection on the graph. Currently, the cycle detection routine is performed twice whenever a formula is entered or modified: once to detect and prevent circular references in the formula and again to detect possible cycles to determine if generalization algorithm should be triggered. One possible improvement is to eliminate the second cycle detection by always generalizing whenever a new formula is entered. This does not add to the overall cost of the generalization algorithm because even if some formulas are not generalized right away, they are generalized when the form is saved and removed from the screen. Although we anticipate this constant saving should improve the performance, further empirical results are needed.

Applying Representation Design Benchmarks to other languages: The Representation Design Benchmarks we devised as part of this work may be applicable to other kinds of computer languages. For instance, web-based design with hypertext markup language (HTML) has some of the same characteristics as programming in a programming language. We would like to investigate applying Representation Design Benchmarks to computer languages such as these, that are not usually considered to be programming languages.

Extending Representation Design Benchmarks for dynamic representations: We chose to study only the static representation in this thesis. However, we believe one advantage of many visual programming languages is their dynamic, highly interactive interface. We would like to explore extensions to our work on Representation Design

Benchmarks that would help VPL designers in designing the dynamic representation of their languages.

6. CONCLUSION

This thesis describes generalized abstractions in form-based visual programming languages. Its two contributions are the generalization algorithm, which derives the generalized programs, and the development of the Representation Design Benchmarks, which enable VPL designers to evaluate static representations of visual programs at language design time.

The generalization algorithm presented in this thesis allows a fully general form-based program to be derived from one whose formulas were specified with concrete examples and direct manipulation. This is accomplished through recognizing and recording the logical relationships among the concrete data, from the perspective of the computational goals of the program fragment currently on the screen. The key benefits of the technique are:

(1) It supports a visual style of general-purpose programming that incorporates extensive use of concrete examples, direct manipulation, and responsiveness.

(2) It removes any order requirements from the user's program entry process. This frees the user to concentrate on problem-solving, rather than having to concentrate on providing information to the computer in the order the computer wants it. Since declarative languages strive to be solely dependent on definitions rather than on the order a program is specified, this is an especially important attribute for declarative VPLs.

(3) It is scalable, because its performance is bounded by the number of program entities currently on the screen, not by the number of entities in the program.

(4) It does not use guessing. Hence there is no restriction to domain-specific programming tasks, because there is no risk of "guessing wrong" in the generalization process.

Representation Design Benchmarks are the first approach devised specifically to help VPL designers to design the static representations of their languages. Extending the work on cognitive dimensions for programming systems, the benchmarks provide a concrete way for VPL designers to apply HCI principles on cognitive aspects of programming. Representation Design Benchmarks have been used both by experienced VPL designers in designing static representations for the interactive VPLs Forms/3 at Oregon State University and ICBE at Hewlett-Packard Laboratories, and by student subjects in a small empirical study. Indications from these uses are that the benchmarks were helpful for VPL designers in discovering problems with their designs.

ANNOTATED BIBLIOGRAPHY

[Almada et al. 1996] Antao Vaz Almada, Antonio Eduardo Dias, Joao Pedro Silva, Emanuel Marques dos Santos, Pedro Jose Pedrosa and Antonio Sousa Camara, "Exploring Virtual Ecosystems", Advanced Visual Interface, Gubbio, Italy, 166-174, 1996.
Keywords: Forms Virtual reality, pen-based input, BITS, time-based approach

[Ambler and Burnett 1989] Allen L. Ambler and Margaret M. Burnett, "Visual Languages and the Conflict Between Single Assignment and Iteration", 1989 IEEE Workshop on Visual Languages, Rome, Italy, 138-143, October 4-6, 1989.
Keywords: Forms/2, Forms-based, temporal assignment
Classification: VPL-II.A.4, VPL-III.B
Summary:
Describes single assignment languages and how iteration can be allowed without violating this concept. Surveys several approaches and details approach in Forms/2, which uses temporal assignment. Single assignment scaling up issues: claims aids understandability; claims aids efficiency by maximizing parallelism. Forms/2 addresses the following scaling up issues: Efficiency: lazy evaluation. Procedural abstraction: forms have interfaces and can be called. General purpose language: supports iteration through the use of temporal single assignment. Debugging: temporal assignment supports browsing of "older" intermediate values. File handling: random read/write handled using temporal assignment paradigm. Documentation: can integrate textual/graphic documentation as part of form. Information hiding: cells may be hidden, revealing only the public interface to the form.

[Ambler and Burnett 1990] Allen L. Ambler and Margaret M. Burnett, "Visual Forms of Iteration that Preserve Single Assignment", *Journal of Visual Languages and Computing* 1(2), 159-181, June 1990.
Keywords: Forms/2, Forms-based
Classification: VPL-II.A.4, VPL-III.B

[Ambler and Hsia 1993] Allen Ambler and Yen-Teh Hsia, "Generalizing Selection in By-Demonstration Programming", *Journal of Visual Languages and Computing* 4(3), 283-300, September 1993.
Keywords: Program-by-demonstration, generalized selection, PT
Classification: VPL-II.A.10
Summary:
Describes a by-demonstration system that does that use inference in determining the generality of the program. Programming in PT requires designing graphical objects, placing them into a context, called a picture, and manipulating these objects graphically. The process of manipulating a picture is recorded as a film. A film corresponds to a procedure and a picture to the combined collection of parameters and local variables used by the procedure. During filming or picture manipulation of objects, three types of manipulations are recorded: situation testing, action, and selection. A situation is a condition that is employed to condition action sequences. An action may be primitive actions like assignment of a value or the creation of an object, or an action may require playing an already recorded film. A selection identifies a set of target objects that may be used in other selections and in specifying object modifications, such as changing their attribute values. While the selection appears concrete in terms of sample data, any manipulation of the selected concrete data is abstracted as a manipulation of all objects satisfying the selection criteria. The feature of PT that makes this generalization possible without inference is the fact that generalized information is explicitly specified by the programmer as part of programming.

[Atwood 1996] John Atwood, "Tracking the Culprits: Making One-Way Constraints GUIs More Responsive", Technical Report 96-60-8, Department of Computer Science, Oregon State University, April 1996.

[Atwood et al. 1996] John Atwood, Margaret M. Burnett, Rebecca Walpole, Eric Wilcox and Sherry Yang, "Steering Programs via Time Travel", 1996 IEEE Symposium on Visual Languages, Boulder, Colorado, 4-11, September 3-6,1996.
Keywords: Time dimension, steering
Summary:
Understanding Forms/3 program behavior by steering through time dimension.

[Ball and Eick 1994] T. Ball and S.G. Eick, "Visualizing Program Slices", 1994 IEEE Symposium on Visual Languages, St. Louis, Missouri, 288-295, October 4-7, 1994.
Keywords: program slicing, program execution visualization

[Barr and Feigenbaum 1981] Avron Barr and Edward A. Feigenbaum, The Handbook of Artificial Intelligence, vol. I, HeurisTech Press, Stanford, California. 1981.
Keywords: Inference, reasoning
Summary:
Definition of inference and various reasoning methods.

[BellB et. al. 1991] Brigham Bell, John Rieman and Clayton Lewis, "Usability Testing of a Graphical Programming System: Things We Missed in a Programming Walkthrough", CHI '91, New Orleans, Louisiana, 7-12, April 27-May 2, 1991.
Keywords: Language facility, Expressiveness, doctrine, ChemTrains
Classification: VPL-VI.C.1
Summary:
Most programming languages focus on efficiency and expressiveness, with minimal attention to language facility, the ease with which a programmer can translate requirements of a task into statements in the language. The programming walkthrough is a method of evaluating two import aspects: Expressiveness- the ability to state solutions to hard problems simply and Facility- the ability to solve problems easily. Require 2 things: a representative set of tasks or problems that the system is intended for, and a document describing what a naive user needs to know about the system (Doctrine). Six subjects were used in the usability testing.

[BellB et al. 1994] B. Bell, W. Citrin, C. Lewis, J. Rieman, R. Weaver, N. Wilde, and B. Zorn, "Using The Programming Walkthrough To Aid In Programming Language Design", *Software—Practice and Experience* 24(1), 1-25, January 1994.

[BellM 1994] M. Bell, "Evaluation Of Visual Programming Languages & Environments", CTI Centre for Chemistry, University of Liverpool (PO Box 147, Liverpool, UK L69 3BX), Technical Report, August 1994.
Keywords: Cognitive dimensions, programming walkthrough
Summary:
Identified the two techniques used for design time evaluation of visual programming languages, cognitive dimensions and programming walkthrough.

[Bertin 1983] Jacques Bertin, Semiology of Graphics: Diagrams Networks Maps, The University of Wisconsin Press, Madison, Wisconsin, 1983
Keywords: Human perception rules, 8 graphic variables

[Blattner et al. 1988] Meera M. Blattner, Lawrence T. Kou, John W. Carlson, and Douglas W. Daniel, "A Visual Interface for Generic Message Translation", 1988 IEEE Workshop on Visual Languages, Pittsburgh, Pennsylvania, 121-126, October 10-12, 1988.
Classification: VPL-III.A.1,
Summary:
The translator accomplishes two tasks: (1) creates a mapping between fields in different message with similar semantic content, and (2) reformats or translates data specifications within these fields.

[Bolognesi and Latella 1989] Tommaso Bolognesi, and Diego Latella, "Techniques for the Formal Definition of the G-LOTOS Syntax", 1989 IEEE Workshop on Visual Languages, Rome, Italy, October 4-6, 1989.
Summary:
Using a grammar to specify a visual language provides a precise definition of the syntax of the language. A spatial parser is an algorithm for recovering the underlying structure of a visual program from the picture. The parser is grammar driven (like yacc). Claims to be more general and extensible with the use of grammar and a general-purpose graphical editor.

[Borning 1986a] Alan Borning, "Graphically Defining New Building Blocks in ThingLab", *Human-Computer Interaction* 2, 269-295, 1986.
Keywords: Constraints
Classification: VPL-II.A.2, VPL-III.A.2,
Summary:
ThingLab is a constraint-oriented, interactive graphical system for building simulations. Three different styles of definition of constraints are illustrated: constraint networks, constraint expressions, and purely graphical definitions. Constraint networks are "wired together" from more primitive elements. Constraint expressions look like standard two-dimensional algebraic notation. Purely graphical definitions are similar to the geometric constructions of Euclidean geometry. Event handling can be embedded in the constraint specification.

[Borning 1986b] Alan Borning, "Defining Constraints Graphically", CHI '86, 137-143, April 1986.
Keywords: Constraints
Classification: VPL-II.A.2

[Borning 1986c] Alan Borning, "Classes versus Prototypes in Object-Oriented Languages", 1986 Fall Joint Computer Conference, Dallas, Texas, 36-40, November 2-6, 1986.
Keywords: Constraints
Classification: VPL-II.A.2

[Borning et al. 1996] Alan Borning, Richard Anderson, Bjorn Freeman-Benson, "Indigo: A Local Propagation Algorithm for Inequality Constraints", ACM Symposium on User Interface Software and Technology, Seattle, Washington, 129-136, November 6-8, 1996.
Keywords: Constraints
Summary:
Adds ability to solve inequality constraints. Best paper at UIST.

[Bos 1992] Edwin Bos, "Some Virtues and Limitations of Action Inferring Interfaces", ACM Symposium on User Interface Software and Technology, Monterey, California, 79-88, November 15-18, 1992.
Keywords: Programming by example, demonstrational interfaces, multimodal interfaces.

[Brown and Gould 1987] P. Brown and J. Gould, "An Experimental Study of People Creating Spreadsheets", *ACM Transactions on Office Information Systems* 5(3), 258-272, July 1987.

[Brown and Najork 1996] M. Brown and M. Najork, "Collaborative Active Textbooks: A Web-Based Algorithm Animation System for an Electronic Classroom", 1996 IEEE Symposium on Visual Languages, Boulder, Colorado, 266-275, September 3-6,1996.
Keywords: algorithm animation, web-based

[Burnett 1991] Margaret M. Burnett, "Abstraction in the Demand-Driven, Temporal-Assignment, Visual Language Model", Ph.D. Thesis, University of Kansas Computer Science Department, Lawrence, Kansas, August 1991.
Keywords: Forms/3, Declarative, Demand-driven, temporal assignment, lazy evaluation
Classification: VPL-II.A.4, VPL-II.B.2, VPL-III.A, VPL-III.E

[Burnett and Ambler 1992a] Margaret M. Burnett and Allen L. Ambler, "A Declarative Approach to Event-Handling in Visual Programming Languages", 1992 IEEE Workshop on Visual Languages, Seattle, Washington, 34-40, September 15-18, 1992.
Keywords: Forms/3, event handling, temporal assignment, declarative
Classification: VPL-II.A.4, VPL-III.E
Summary:
Elegant approach to event handling and contributes towards scaling up problem. Events are fully declarative. Use of temporal assignment enables access to prior values of a cell; as events are values, this provides an event history. System, user interface and user-defined events supported.

[Burnett and Ambler 1992b] Margaret M. Burnett and Allen L. Ambler, "Generalizing Event Detection and Response in Visual Programming Languages", Advanced Visual Interfaces: an International Workshop, Rome, Italy, May 1992.
Keywords: Forms/3, event handling, temporal assignment, declarative
Classification: VPL-II.A.4, VPL-III.E
Summary:
Describes a general approach to event detection and handling in VPLs. Event detection and response are separated. Events are treated as data, and a new event is seen as a change in the data value. Allows events to be treated as abstract datatypes. Single notation, consistent with the host language; no special tools are needed. Approach applicable to various VPL types, e.g. stream-based, data flow, forms-based. User can define higher-level events using the primitives.

[Burnett et al. 1992] Margaret M. Burnett, Richard Hossli, Timothy Pulliam, Brian VanVoorst, and Xiaoyang Yang, "Toward Visual Programming Languages for Steering in Scientific Visualization: a Taxonomy", CS-TR 92-12, Michigan Technological University, December 1992.
Keywords: Scientific visualization
Classification: VPL, VPL-V.D

[Burnett 1993a] Margaret M. Burnett, "On Types and Type Inference in a Visual Programming Language", 1993 IEEE Symposium on Visual Languages, Bergen, Norway, August 24-27, 1993.
Keywords: Forms/3, Type checking, typing theory
Classification: VPL-II.A.4, VPL-VI.C.5
Summary:
This paper describes the Milner-style type system developed for Forms/3. The principal goal of the type system is to provide immediate feedback on type errors while minimizing language restrictions, new concepts to be learned, or additional information to be manually specified.

[Burnett 1993b] Margaret M. Burnett, "Toward Scaling Up: Visual Data Abstraction For Declarative Visual Programming Languages", NSF Grant Proposal, 1993.
Keywords: Forms/3, generalization, event handling, temporal assignment, declarative
Classification: VPL-II.A.4, VPL-III.E

[Burnett and Ambler 1994] Margaret M. Burnett and Allen L. Ambler, "Interactive Visual Data Abstraction in a Declarative Visual Programming Language", *Journal of Visual Languages and Computing* 5(1), 29-60, March 1994.
Keywords: data abstraction, Forms/3

[Burnett et. al. 1994] Margaret M. Burnett, R. Hossli, T. Pulliam, B. VanVoorst, and X. Yang, "Toward Visual Programming Languages for Steering in Scientific Visualization: a Taxonomy", *IEEE Computational Science & Engineering*, December 1994.

[Burnett et al. 1995a] Margaret M. Burnett, Adele Goldberg and Ted G. Lewis, Visual Object-Oriented Programming: Concepts and Environments, Prentice-Hall/Manning Publications, Englewood Cliffs, NJ, 1995.

[Burnett et al. 1995b] Margaret M. Burnett, Marla Baker, Carisa Bohus, Paul Carlson, Sherry Yang, and Piet van Zee, "Scaling Up Visual Programming Languages", *Computer*, 45-54, March 1995.

[Card et al. 1983] S. Card, T. Moran, and A. Newell, The Psychology of Human-Computer Interaction, Erlbaum, Hillsdale, NJ, 1983.

[Card et al. 1991] Stuart K. Card, George G. Robertson and Jock D. Mackinlay, "The Information Visualizer, An Information Workspace", CHI '91, New Orleans, Louisiana, 181-188, April 27- May 2, 1991.
Keywords: interface metaphors, screen layout

[Chalmers et al. 1996] Matthew Chalmers, Robert Ingram, Christopher Pfranger, "Adding Imageability Features to Information Displays", ACM Symposium on User Interface Software and Technology, Seattle, Washington, 33-39, November 6-8, 1996.

Keywords: data visualization

Summary:

Better data visualization techniques. Use of clustering. Size of a data point determined by the number of hits the item had and size changes as the user modifies the criteria.

[Chung and Dewan 1996] Goopeel Chung and Prasum Dewan, "A Mechanism for Supporting Client Migration in a Shared Window System", ACM Symposium on User Interface Software and Technology, Seattle, Washington, 11-20, November 6-8, 1996.

Keywords: Optimizations, distributed visual language

Summary:

Migration of a client process from one machine to another while still maintaining good response time.

[Citrin et. al. 1995] W. Citrin, R. Hall and B. Zorn, "Programming with Visual Expressions", 1995 IEEE Symposium on Visual Languages, Darmstadt, Germany, 294-301, September 5-9, 1995.

Keywords: VIPR

[Citrin and Santiago 1996] W. Citrin and C. Santiago, "Incorporating Fisheyeing into a Visual Programming Environment", 1996 IEEE Symposium on Visual Languages, Boulder, Colorado, 20-27, September 3-6, 1996.

Keywords: VIPR, zooming, fisheyeing

[Cohen 1992] Cohen, Philip R., "The Role of Natural Language in a Multimodal Interface", ACM Symposium on User Interface Software and Technology, Monterey, California, 143-149, November 15-18, 1992.

Keywords: Direct Manipulation, natural language, communication/interaction modalities

Summary:

Strengths and weaknesses of direct manipulation and natural languages examined.

[Coote et al. 1988] Susan Coote, John Gallagher, John Mariani, Thomas Rodden, Andrew Scott, and Doug Shepherd, "Graphical and Iconic Programming Languages for Distributed Process Control: An Object Oriented Approach", 1988 IEEE Workshop on Visual Languages, Pittsburgh, Pennsylvania, 183-190, October 10-12, 1988.

Summary:

A Graphical System Description Language (GSDL), based on an extended GRAFCET notation, is described. A higher level iconic approach to the description of control strategies is also described. GRAFCET uses the state transition diagram to describe control flow. GSDL abstracts the control flow in two levels, the behavior of objects and the interaction of objects.

[Cordy and Graham 1992] James Cordy and T.C. Nicholas Graham, "GVL: Visual Specification of Graphical Output", Journal *of Visual Languages and Computer* 3(1), 25-47, March 1992.

Keywords: Graphical Output, Output specification

Summary:

GVL is a functional language that allows you to specify graphical output.

[Cox et al. 1989] P. T. Cox, F. R. Giles, T. Pietrzykowski, "Prograph: A Step Towards Liberating Programming from Textual Conditioning", 1989 IEEE Workshop on Visual Languages, Rome, Italy, 150-156, October 4-6, 1989.

Classification: VPL-1, VPL-II.A.3, VPL-II.B.1, VPL-III.A, VPL-III.E

Summary:

This paper describes how Prograph represents the main concepts of object oriented programming, classes, attributes and methods and how a method's representation defines the semantics of its dataflow, data driven execution. In addition, the paper describes how parallelism, sequencing, iteration and conditional constructs are modeled in Prograph. Data values in Prograph are untyped. Values and types are checked at run time. Prograph provides a rich set of iteration and parallelism mechanisms through multiplexes. A comment may be attached to any icon or program element and serves only to provide additional information for the viewer. The Prograph environment has three main components, the editor, interpreter, and application builder. The editor is used for program design and construction; the interpreter executes a program and provides debugging facilities; the application builder simplifies the task of constructing a graphical user interface for a program. The interpreter of Prograph provides a stack window for tracing the execution of an application.

[Cox and Pietrzykowski 1992] P. T. Cox and T. Pietrzykowski, "Visual Message Flow Language MFL and its Interface", Advanced Visual Interfaces : an International Workshop, Rome, Italy, May 27-29, 1992.

Summary:

Computations are specified in terms of message passing: 1. The control of message propagation through the network of objects is separated from the behavior of the objects themselves. 2. The system provides "hyperpictoriality" which adds a third dimension to pictures in a way analogous to hypertext.

[Cunniff and Taylor 1987] Nancy Cunniff and Robert P. Taylor, "Graphical vs. Textual Representation: An Empirical Study of Novice's Program Comprehension", Empirical Studies of Programmers: Second Workshop, Washington D.C., 114-131, December 7-8, 1987.

Classification: VPL-VI.C.1

[Cypher et al. 1993] A. Cypher, D. Kosbie, and D. Maulsby, "Characterizing PBD Systems", in Watch What I Do: Programming by Demonstration, (A. Cypher, ed.), MIT Press, Cambridge, MA, 1993.

[Darlington and While 1987] John Darlington and Lyndon While, "Controlling the Behavior of Functional Language Systems", Functional Programming Languages and Computer Architecture, Portland, Oregon, 278-300, 1987.

Keywords: Temporal Constraints, Declarative, Event handling.

[Del Bimbo at al. 1994a] A. Del Bimbo, E. Vicario and D. Zingoni, "An Interactive Environment for the Visual Programming of Virtual Agents", 1994 IEEE Symposium on Visual Languages, St. Louis, Missouri, 145-152, October 4-7, 1994.

[Del Bimbo at al. 1994b] A. Del Bimbo, P. Pala and S. Santini, "Visual Image Retrieval by Elastic Deformation", 1994 IEEE Symposium on Visual Languages, St. Louis, Missouri, 216-223, October 4-7, 1994.

[Douglas et. al 1995] S. Douglas, C. Hundhausen and D. McKeown, "Toward Empirically-Based Software Visualization languages", 1995 IEEE Symposium on Visual Languages, Darmstadt, Germany, 342-350, September 5-9, 1995.

[Du and Wadge 1990] Weichang Du and William Wadge, "A 3D Spreadsheet Based on Intensional Logic", *IEEE Software*, 78-89, May 1990.
Keywords: Spreadsheet
Classification: VPL-II.A.4
Summary:
Three-dimensional (two spatial, one temporal) spread-sheet-based system. The system is graphical in the sense that conventional spreadsheets are graphical. The intensional logic refers to a set of composable operators providing a form of relative addressing of cells. Multiple named spreadsheets can be created and linked to one another via references in cell formulas. In addition to cells, a spreadsheet can have a set of global variables and a set of named function definitions written in Plane Lucid, a text-based dataflow programming language. The system addresses the scaling-up problem for spreadsheets by providing user-definable functions in a reasonable text-based programming language, by providing named global variables, and by allowing spreadsheets to be linked. If the user-defined functions and global variables are local to the spreadsheet to which they belong, they could be used to provide a form of data abstraction. The system still relies heavily on the use of relative addressing on a system-defined grid, which does not scale up well. The provision of multiple linkable spreadsheets ameliorates this to some degree, but it adds a new level of complexity. There are no provisions for graphical extensibility.

[Egenhofer 1996] Max Egenhofer, "Spatial Query-by-Sketch", 1996 IEEE Symposium on Visual Languages, Boulder, Colorado, 60-67, September 3-6, 1996.
Keywords: visual query

[Fisher and Busse 1992] Gene L. Fisher and Dale E. Busse, "Adding Rule-Based Reasoning to a Demonstrational Interface Builder", ACM Symposium on User Interface Software and Technology, Monterey, California, 89-97, November 15-18, 1992.
Keywords: UIMSs, Interface Builders, Programming by demonstration, direct manipulation

[Ford 1993] Lindsey Ford, "How Programmers Visualize Programs", Draft paper, March 1993.
Keywords: Visualization, algorithm animation
Classification: VPL-VI.C.1
Summary:
Exploration of graphical and animated representations has implications for: 1) How programming is taught, learned and empirically studied, and 2) Program visualization and visual programming. Use graphics to teach algorithms and program constructs. Recognition that an intrinsic difficulty in programming arises out of the need to visualize a program in two dimensions - space and time. These dimensions are apparent during a program's construction (the source program is a spatial object, the programmer mentally simulates its execution over time), and during its execution (functions and data are in the spatial dimension, the program executes in real time). Programming is made difficult by the programmer having to use inappropriate tools to reconcile these two dimensions: text-based editors present the program only a spatial dimension and limit expression of dynamic concepts in it, such as looping and inheritance, to the textual medium; and similarly, text-based debuggers focus on time-based dynamics through textual changes, and largely ignore the spatial dimension. 46 students used.

[Freeman et al. 1995] Elisabeth Freeman, David Gelernter and Suresh Jagannathan, "In Search of a Simple Visual Vocabulary", IEEE Symposium on Visual Languages, Darmstadt, Germany, 302-309, September 5-9, 1995.
Keywords: Time dimension, map

[Freeman et al. 1996] Elisabeth Freeman, D. Gelernter, S. Jagannathan, "Uniformity of Environment and Computation in MAP", 1996 IEEE Symposium on Visual Languages, Boulder, Colorado, 130-137, September 3-6, 1996.
Keywords: MAP
Summary:
Map is a system to visualize data and executions. Visualizes execution history

[Freeman-Benson 1989] Bjorn Freeman-Benson, "A Module Mechanism for Constraints in Smalltalk", OOPSLA '89, 389-396, October 1989.

[Gautier and Le Guernic 1987] Thierry Gautier and Paul Le Guernic, "SIGNAL: A Declarative Language for Synchronous Programming of Real-time Systems", Functional Programming Languages and Computer Architecture, Portland, Oregon, 257-277, 1987.
Keywords: Declarative Languages, Event handling

[Ginsburg et al. 1996] Adam Ginsburg, Joe Marks, Stuart Shieber, "A Viewer for PostScript Documents", ACM Symposium on User Interface Software and Technology, Seattle, Washington, 31-32, November 6-8, 1996.
Summary:
How to view a large PostScript document by scanning through the pages. Like a scroll bar, the position of the handler indicate the approximate position of the current page in the document.

[Glinert et al. 1990] Ephraim P. Glinert, Mark E. Kopache and David W. McIntryre, "Exploring the General-Purpose Visual Alternative", *Journal of Visual Languages and Computing* 1(1), 3-39, 1990.

[Glinert and Tanimoto 1984] Ephraim Glinert and Steven Tanimoto, "Pict: An Interactive Graphical Programming Environment", *Computer* 17(11), 7-25, November 1984.

[Glinert 1989] E. Glinert, "Towards a Software Metrics for Visual Programming", *International Journal of Man-Machine Studies* 30, 425-445, 1989.
Summary:
A framework for software metrics which measures the attractiveness of a visual computing environment.

[Golin and Reiss 1989] Eric J. Golin and Steven P. Reiss, "The Specification of Visual Language Syntax", 1989 IEEE Workshop on Visual Languages, Rome, Italy, October 4-6, 1989.

[Gorlick and Quilici 1994] Michael Gorlick and Alex Quilici, "Visual Programming-in-the-Large versus Visual Programming-in-the-Small", 1994 IEEE Symposium on Visual Languages, St. Louis, Missouri, 137-144, October 4-7, 1994.

Keyword: semantic zooming, representation, scaling up.

Summary:

This paper describes a visual software composition and integration environment which addresses the scaling problem of date-flow representations. The problems of visual cluttering, difficulty of locating components and non-visual response to user errors are discussed. Semantic zooming is used to address the cluttering problem. Formal object-oriented representation of components is used to deal with the component discovery problem. Finally, efforts are made to make it hard for users to make mistakes.

[Gottfried and Burnett 1996] Herkimer J. Gottfried and Margaret M. Burnett, "Graphical Definitions: Expanding Spreadsheet Languages through Direct Manipulation and Gestures", Technical Report 96-60-3, Department of Computer Science, Oregon State University, April 1996.

Keyword: spreadsheets, gestures, direct manipulation, visual programming, Forms/3

Summary:

This paper describes using gestures and direct manipulation to define complex objects within the spreadsheet paradigm.

[Graf 1987] Mike Graf, "A Visual Environment for the Design of Distributed Systems", 1987 IEEE Workshop on Visual Languages, 330-344, August 1987.

Summary:

A workspace and tools for creation, simulation, and analysis of distributed system design. Difficulties of using a linear technology (text) to describe nonlinear phenomena (distributed and concurrent computation) is discussed.

[Graf 1990] Mike Graf, "Visual Programming and Visual Languages: Lessons Learned in the Trenches", in Visual Programming Environments: Applications and Issues (ed. E.P. Glinert), 452-454, 1990.

[Green 1991] T. R. Green, "Describing Information Artifacts With Cognitive Dimensions and Structure Maps", in People and Computers VI, (D. Diaper and N. Hammond, eds.), Cambridge University Press, 1991.

[Green et al. 1991] T.R. Green, M. Petre and R.K.E. Bellamy, "Comprehensibility of Visual and Textual Programs: A Test of Superlativism Against the "Match-Mismatch" conjecture", Empirical Studies of Programmers: Fourth Workshop, New Brunswick, NJ, 121-146, December 7-9, 1991.

Keywords: Superlativist, information accessibility, match-mismatch conjuncture
Classification: VPL-VI.C.1
Summary:

Studies the readability of textual and graphical programs. VPL claimed advantages in both levels of comprehension - low level micro-structure and high-level reasoning, but every empirical studies explicitly comparing the claims against textual PLs. The study contrast two major hypotheses: superlativism (graphical is naturally best, VPLs are just good in general because of the 2 dimensional visual perception is more natural, more efficient, etc. than reading text.) and information accessibility (the structure of any given VPL, in combination with the reader's experience, will mean that certain tasks are easy and others are hard; we can pin down the contribution of visual notations more exactly: specific ways of designing the notation will result in making specific information easier to access and will therefore make specific tasks easier, other tasks, however, may not benefit, or may even suffer). Short conditional expressed in 4 notations (text or graphic cross with sequential and circumstantial). Graphical programs took longer than textual ones. 5 subjects

[Green and Petre 1994] T.R.G. Green and M. Petre, "Cognitive Dimensions as Discussion Tools for Programming Language Design", In submission, January 1994.

Keyword: cognitive dimensions
Summary:

This paper describes the use of cognitive dimensions as a framework for analyzing the design of visual programming languages. Cognitive dimensions is a set of terms that describe the cognitively important aspect of a language. Some of the dimensions include hidden dependencies, consistency, closeness of mapping to the domain, imposed guess-ahead, etc. Five languages are compared in the paper - Pascal, Basic, Prograph, LabView and spreadsheets. Spreadsheets, for examples, are very poor in hidden dependencies and visibility but are doing quite well in closeness of mapping to the problem domain and have no difficulties in imposed guess-ahead.

[Green and Petre 1995] T. Green and M. Petre, "Usability Analysis of Visual Programming Environments: a 'Cognitive Dimensions' Framework", Technical Report, MRC Applied Psychology Unit, 1995.

[Green and Petre 1996] T.R.G. Green and M. Petre, "Usability Analysis of Visual Programming Environments: A 'Cognitive Dimensions' Framework", *Journal of Visual Languages and Computing* 7 (2), 131-174, June 1996.

[Griebel et al. 1996] P. Griebel, G. Lehrenfeld, W. Mueller, C. Tahedl, H. Uhr, "Integrating a Constraint Solver into a Real-time Animation Environment", 1996 IEEE Symposium on Visual Languages, Boulder Colorado, 12-19, September 3-6, 1996.

Keywords: pictorial Janus

[Gross and Do 1996] Mark D. Gross and Ellen Yi-Luen Do, "Ambiguous Intentions: A Paper-Like Interface for Creative Design", ACM Symposium on User Interface Software and Technology, Seattle, Washington, 183-192, November 6-8, 1996.
Keywords: Programming by demonstration
Summary:
Cocktail Napkin- pen-based environment for creative visual design.

[Guimaraes et al. 1992] Nuno M. Guimaraes, Nuno M. Correia and Telmo A. Carmo, "Programming Time in Multimedia User Interfaces", ACM Symposium on User Interface Systems and Technology, 125-134, November 15-18, 1992.
Keywords: time-based approach

[Gurka and Citrin 1996] J. Gurka and W. Citrin, "Testing Effectiveness of Algorithm Animation", 1996 IEEE Symposium on Visual Languages, Boulder Colorado, 182-189, September 3-6, 1996.
Keywords: algorithm animation, empirical studies

[Haeberli 1988] Paul Haeberli, "ConMan: A Visual Programming Language for Interactive Graphics", *Computer Graphics* 22(4), 103-111, August 1988.
Keywords: Dataflow languages, Event Handling related

[Harbert et al. 1990] Andrew Harbert and William Lively and Sallie Sheppard, "A Graphical Specification System for User-Interface Design", *IEEE Software* 7, 12-20, July 1990.

[Hashimoto and Myers 1992] Osamu Hashimoto and Brad A. Myers, "Graphical Styles for Building User Interfaces by Demonstration", ACM Symposium on User Interface Software and Technology, Monterey, California, 117-124, November 15-18, 1992.
Keywords: User Interface Builder, UIMS, Demonstrational Interfaces, Styles, Tabs, Garnet, Direct Manipulation, Inferencing.

[Hays and Burnett 1995] Judith G. Hays and Margaret M. Burnett, "A Guided Tour of Forms/3", TR 95-60-6, Department of Computer Science, Oregon State University, June 1995.
Summary:
A guided tour to what it's like to use our language. Intended to simulate what it feels like to sit in front of a workstation and use it to create several simple programs.

[Henderson and Card 1990] D. Austin Henderson, Jr. and Stuart K. Card, "Rooms: The Use of Multiple Virtual Workspaces to Reduce Space Contention in a Window-Based Graphical User Interface", in Visual Programming Environments: Applications and Issues (ed. E.P. Glinert), 369-401, 1990.
Keywords: screen real estate

[Hendry and Green 1993] D. G. Hendry and T.R. G. Green, "CogMap: a Visual Description Language for Spreadsheets", *Journal of Visual Languages and Computing* 4(1), 35-54, March 1993.
Keywords: spreadsheet, cognitive, user interface

[Hendry 1995] D. Hendry, "Display-based Problems in Spreadsheets: A Critical Incident and a Design Remedy", 1995 IEEE Symposium on Visual Languages, Darmstadt, Germany, September 5-9, 1995.

[Hils 1992] Daniel D. Hils, "Visual Languages and Computing Survey: Data Flow Visual Programming Languages", *Journal of Visual Languages and Computing* 3(1), 69-101, March 1992.
Summary:
This paper is a survey of a number of visual data flow programming languages. The discussion is ordered by domain of application of the language. For each language, the paper gives some small examples of its use and discusses some of the characteristics of the language. The characteristics chosen for discussion are drawn from a table of design alternatives given at the beginning of the paper. Of these, only two, procedural abstraction and type-checking, are among those we have identified as relevant to scaling up.

[Hirakawa et al. 1990] M. Hirakawa, M. Tanaka, and T. Ichikawa, "An Iconic Programming System, HI-VISUAL", *IEEE Transactions on Software Engineering* 16(10), 1178-1184, October 1990.

[Hix 1990] Deborah Hix, "Generations of User-Interface Management Systems", *IEEE Software* 7, 77-87, September 1990.
Keywords: UIMS, UI
Summary:
This paper survey the generation of UIMS. Refer to MS paper for more details.

[Hosking 1996] John Hasking, "Visualization of Object-Oriented Program Execution", 1996 IEEE Symposium on Visual Languages, Boulder, Colorado, 190-191, September 3-6, 1996.

[Hsia 1989] Yen-Teh Hsia, "Programming Through Pictorial Transformations", Ph.D. Thesis, University of Kansas Computer Science Department, 1989.
Keywords: Non-inference, general purpose visual programming language
Summary:
This paper describes the PT system. PT does not use inference.

[Hudson and Smith 1996] Scott E. Hudson and Ian Smith, "Ultra-Lightweight Constraints", ACM Symposium on User Interface Software and Technology, Seattle, Washington, 147-155, November 6-8, 1996.
Keywords: Constraints
Summary:
Space optimizations that work for layout constraints. Lazy evaluation with eager marking.

[Hughes and Moshell 1990] C. E. Hughes and J. M. Moshell, "Action Graphics: A Spreadsheet-Based Language for Animated Simulation", in Visual Languages and Applications (T. Ichikawa, E. Jungert, R. Korfhage, eds.), Plenum Press, NY, 203-236, 1990.

Summary:

Action Graphics is a VPL for graphics animation with some general purpose features. It combines elements of free-form spreadsheets and object-oriented programming in a nice programming environment. Execution is undoable for many operations, allowing the user to walk backwards through execution. Execution can be run in single step mode. Cell evaluation can be disabled, and disabling/enabling can be performed by hand or via program--this is useful as a form of "conditional compilation" and also for debugging. Changes can be propagated to all cells (default) or propagation can be restricted to certain cells of interest, so that only changes affecting those cells are propagated. Evaluation of any given cell can be requested.

[Ingalls et al. 1988] Dan Ingalls, Scott Wallace, Yu-Ying Chow, Frank Ludolph, Ken Doyle, "Fabrik, A Visual Programming Environment", OOPSLA '88, San Diego, September 1988.

Keywords: Dataflow

Summary:

A well-rounded and well implemented VPL using bi-directional dataflow. Makes a number of contributions to scaling up. Liveness is level 4. Bi-directional data flow reduces number of components and connections necessary; icons in browser are minified until extracted for editing. Static type checking utilizes type propagation to type untyped pins. User interface events detected using a "sensor", system events(clock) detected by change of value. Aggregate types, array, enumerated supported. Supports procedural abstraction, parameters may be polymorphic in type. Routines are compiled dynamically upon creation (uses Smalltalk). Each pin displays pin name and type when mouse is over it. Cannot connect pins of incompatible types. Status line always shows component name and pins. Documentation display can be always or on-demand. User can add new low-level component kits.

[Kado et al. 1992] M. Kado, M. Hirakawa, and T. Ichikawa, "HI-VISUAL for Hierarchical Development of Large Programs", 1992 IEEE Workshop on Visual Languages, Seattle, Washington, 48-54, September 15-18, 1992.

[Kahn 1996a] Ken Kahn, "ToonTalk–An Animated Programming Environment for Children", *Journal of Visual Languages and Computing* 7 (2), 197-217, June 1996.

[Kahn 1996b] Ken Kahn, "Seeing Systolic Computations in a Video Game World", 1996 IEEE Symposium on Visual Languages, Boulder Colorado, 95-101, September 3-6, 1996.

Keywords: Toon Talk

[Kamin 1990] Samuel N. Kamin, Programming Languages An Interpreter-Based Approach, Addison-Wesley Publishing Company, Inc. 1990.

[Kimura et al. 1990] Takayuki Dan Kimura, Julie W. Choi, Jane M. Mack, "Show and Tell", in Visual Computing Environments (E. P. Glinert ed.), IEEE Computer Society Press, Washington, D.C., 1990.

Summary:

VPL- boxes and arrows. Subroutine, iteration, recursion and concurrency functions are all represented by 2 dimensional graphic patterns.

[Kimura 1988] Takayuki Dan Kimura, "Visual Programming by Transaction Network", 21st Hawaii International Conference on System Sciences, Kona, Hawaii, 648-654, 1988.
Summary:
This paper introduces a new parallel computation model that is suitable for pursuit of large scale concurrency and fine-grained parallelism. Demote the notion of process as the key concept in organizing large scale parallel computation. Instead, author promotes the notion of transaction, an anonymous atomic action void of internal state, as the basic element of computation. In transaction net, database are connected by transactions. A transaction is an atomic action with constraints.

[Kodosky et al. 1991] J. Kodosky, J. MacCrisken, G. Rymar, "Visual Programming Using Structured Data Flow", 1991 IEEE Workshop on Visual Languages, Kobe, Japan, October 1991.

[Koike and Yoshihara 1993] Hideki Koike and Hirotaka Yoshihara, "Fractal Approaches for Visualizing Huge Hierarchies", IEEE Symposium on Visual Languages, Bergen, Norway, August 24-27, 1993.
Keywords: cone tree, screen real estate

[Koike et al. 1996] Y. Koike, Y. Maeda, Y. Koseki,, "Enhancing Iconic Program Reusability with Object Sharing", 1996 IEEE Symposium on Visual Languages, Boulder, Colorado, 288-295, September 3-6, 1996.

[Krishnamurthy and Zloof 1995] R. Krishnamurthy and M. Zloof, "RBE: Rendering by example", Eleventh International Conference on Data Engineering, Taipei, Taiwan, 288-297, March 6-10, 1995.
Keywords: ICBE
Summary:
Describes how rendering is done in ICBE.

[Kurlander and Feiner 1992] David Kurlander, Steven Feiner, "A History Based Macro By Example System", ACM Symposium on User Interface Software and Technology, Monterey, California, 99-106, November 15-18, 1992.
Keywords: Macros, demonstrational techniques, histories, graphical representations, programming by example
Summary:
Describes how macros are defined, how the user can edit previous work to create new macros, and how generalization from examples is done. An extension to concept of editable graphical histories [Kurlander and Feiner 1990]. Static representation: a)novel approach to static representation of programs developed in a programming-by-example system; b)uses the same language for the static representation as the programming language itself. Allows user to interactively select those elements of a completed programming-by-demonstration sequence for use as procedure parameters. Allows invocation of procedures by name. Once converted into a macro, the original graphical history which was its origin is lost. Graphical history of operations is constantly being acquired--user doesn't have to enter a special mode to capture a macro. Objects in the scene that provide context are dimmed; active objects are highlighted. All macros use sample values to demonstrate their operation. To generalize macros to work in new contexts, an inference engine is used to determine the necessary attributes and arguments of each object manipulation. Generalization can either be done by the system using inference or explicitly specified by the user.

[Kurlander and Feiner 1990] David Kurlander, Steven Feiner, "A Visual Language for Browsing, Undoing, and Redoing Graphical Interface Commands", in Visual Languages and Visual Programming (S.K. Chang, ed.), 257-275, Plenum Press, New York, 1990.

Keywords: Macros, demonstrational techniques, histories, graphical representations, programming by example

Summary:

Presents an implemented graphical history editor for a drawing tool which supports visual presentation of prior actions as well as their undoing, modification, and redoing. Uses comic-strip metaphor (panels depicting states) to present the history. Addresses several scaling up issues. Uses context and heuristics to coalesce a sequence of actions of lesser importance into one panel. Hierarchical on-demand decomposition of composite actions into their components and vice versa. Objects of interest are emphasized in panels (even magnified); objects of unimportance are de-emphasized or left out. Interactive undo/redo allows study of drawing composition. Uses "Landmarks", which are shapes that are part of the scene that give context information.

[Ladret and Rueher 1991] Didier Ladret and Michel Rueher, "VLP: a Visual Logic Programming Language", *Journal of Visual Languages and Computing* 2(2), 163-188, June 1991.

[Larson 1992] James A. Larson, Interactive Software: Tools for Building Interactive User Interfaces, Prentice-Hall, Englewood Cliffs, New Jersey, 1992.

[Lau-Kee et al. 1991] David Lau-Kee, Adam Billyard, Robin Raichney, Yasuo Kozato, Paul Otto, Mark Smith, Ian Wilkinson, "VPL: An Active, Declarative Visual Programming System", 1991 IEEE Workshop on Visual Languages, Kobe, Japan, 40-46, October 1991.

Summary:

A well designed and implemented visual language applied to domain of image processing. Addresses a number of scaling up issues. Procedural abstraction provided. Language extension through use of 1st class functions (example given: build own repeat..until construct). Responsiveness maintained through use of client-server model for user interface and processing. Efficiency is good by using lazy evaluation. Documentation viewed by drag/drop on to "Book". New data objects automatically documented. Liveness level 4 contributes to understanding and debugging programs. Producer/Consumer/Probe triad is simple conceptual model for understanding. General purpose programming language (recursion, selection, abstraction). Programming environment has component (procedure) and data browsers.

[Lewis 1987] Clayton Lewis, "NoPumpG: Creating Interactive Graphics with Spreadsheet Machinery", University of Colorado at Boulder, Department of Computer Science Technical Report CU-CS-372-87, August 1987. Also revised and reprinted in Visual Programming Environments: Paradigms and Systems (E. Glinert, editor), IEEE CS Press, 1990.

Keywords: Event handling related, spreadsheet

Summary:

NoPumpG is an extension of the spreadsheet model. It does not use the standard two-dimensional grid of cells, instead allowing named cells containing a value and an optional formula to be placed anywhere on the screen. It provides a limited bi-directional linkage between graphical objects and control cells, allowing values or formulas placed in the cells to change aspects of the graphical objects, and manipulations on the graphical objects to change the values in control cells without formulas. The system has a built-in capability to respond to two kinds of events, mouse-dragging and clock ticks, but this facility is not user-programmable. Collections of cells and graphical objects that depend on the clock function as processes. Cells can be hidden in NoPumpG, providing a limited abstraction mechanism. Saved models can be copied into other models, providing limited code-reuse capability. However, modifications made to the original are not propagated to copies.

[Linton et al. 1989] Mark Linton, John Vlissides and Paul Calder, "Composing User Interfaces with InterViews", *Computer* 22(2), 8-22, February 1989.

[Ludolph et al. 1988] F. Ludolph, Y.-Y. Chow, D. Ingalls, S. Wallace, K. Doyle, "The Fabrik Programming Environment", 1988 IEEE Workshop on Visual Languages, Pittsburgh, Pennsylvania, 222-230, October 10-12, 1988.

Keywords: event handling

Summary:

More in-depth look at the Fabrik environment. Needs [Ingalls, et. al 1988] for completeness. Spreadsheets successful for understandability because of use of familiar representations and immediate feedback. Bi-directional dataflow reduces components and connections needed through two-way constraints. Structural dataflow model (data is not absorbed, but remains present for lifetime of the execution) aids understanding through timelessness and simplified model of iteration. Algebraic expressions are entered textually; variables map to pins. Reports results of experiments on mouse/keyboard commands: a) designed complete, visual, and efficient one-handed interface (use gestures to enhance mouse bandwidth) b)augment it with two-handed alternatives for increased user performance and continuous use. Environment: components browser supports retrieval by search on partial name, keyword, and content; content search a significant contribution. Persistence of values after execution is beneficial here.

[Mackinlay, Robertson and Card 1991] Jock D. Mackinlay, George G. Robertson and Stuart K. Card, "The Perspective Wall: Detail and Context Smoothly Integrated", CHI '91, New Orleans, Louisiana, 173-179, April 27- May 2, 1991.

Keywords: interface metaphors, screen layout, screen real estate

[Miyashita et al. 1992] Ken Miyashita, Satoshi Masuoka, Shin Takahashi and Akinori Yonezawa, "Declarative Programming of Garphical Interfaces by Visual Examples", ACM Symposium on User Interface Software and Technology, Monterey, California, 107-116, November 15-18, 1992.

Keywords: graphical user interface, direct manipulation, constraints, programming by example, layouts, visualizations.

[Modugno and Myers 1993a] Francesmary Modugno and Brad Myers, "Visual Representations as Feedback in a Programmable Visual Shell", Technical Report CMU-CS-93-133, School of Computer Science, Carnegie Mellon University, 1993.

[Modugno and Myers 1993b] Francesmary Modugno and Brad Myers, "Typed Output and Programming in the Interface", Technical Report CMU-CS-93-134, School of Computer Science, Carnegie Mellon University, 1993.

[Modugno et al. 1994] F. Modugno, T. Green, and B. Myers, "Visual Programming in a Visual Domain: a Case Study of Cognitive Dimensions", People and Computers IX (G. Cockton, S. Draper, and G. Weir, eds.), Cambridge University Press, Cambridge, UK, 1994.

[Modugno and Myers 1994] F. Modugno and B.A. Myers, "A State-Based Visual Language for a Demonstrational Visual Shell", 1994 IEEE Symposium on Visual Languages, St. Louis, Missouri, 304-313, October 4-7, 1994.

[Moher et al. 1993] Thomas G. Moher, David C. Mak, Brad Blumenthal and Laura M. Leventhal, "Comparing the Comprehensibility of Textual and Graphical Programs: The Case of Petri Nets", Draft paper, March 1993.
Keywords: superlativist view, circumstantial vs. sequential
Classification: VPL-VI.C.1
Summary:
Find criteria beyond the sequential/circumstantial dichotomy which could account for differences in the relative comprehensibility of graphical program representations. Forward vs. backward questions. Found no evidence to support the superlativist claims regarding graphical programming, similarly for superlativist claims regarding textual representation. Petri-net is used as the graphical representation, some problems were discussed. Found that "Not only is no single representation best for all kinds of programs, no one representation is even best for tasks involving the same program". Total of 12 subjects were used with 9 used in results.

[Moher et al. 1993] T. Moher, D. Mak, B. Blumenthal, and L. Leventhal, "Comparing the Comprehensibility of Textual and Graphical Programs: the Case of Petri Nets", Empirical Studies of Programmers: Fifth Workshop, Palo Alto, California, 1993.

[Myers 1983] Brad A. Myers, "Incense: a System for Displaying Data Structures", *Computer Graphics* 17 (3), 115-125, 1983.

[Myers 1989a] Brad A. Myers, "User-Interface Tools: Introduction and Survey", *IEEE Software*, 15-23, January 1989, .
Keywords: UIMS, UI toolkits, direct manipulation
Summary:
Difference between UI toolkit and UIMS. Explained different approaches.

[Myers 1989b] Brad A. Myers, "Encapsulating Interactive Behaviors", CHI '89, Austin, TX, 319-324, May 1989.

[Myers et. al 1990] Brad Myers, Dario Guise, Roger Dannenberg, Brad Vander Zanden, David Kosbie, Edward Pervin, Andrew Mickish, Philippe Marchal, "Garnet: Comprehensive Support for Graphical, Highly Interactive User Interfaces", *Computer*, 71-85, November 1990.

[Myers 1990] Brad A. Myers, "Taxonomies of Visual Programming Languages and Program Visualization", *Journal of Visual Languages and Computing* 1(1), 97-123, March 1990.

Summary:

Covering lots of visual languages and other visual aspects of non-visual languages. Plus some words on motivations and current problems. VPL-specific problems: lack of evidence of their worth, lack of portability of programs, difficulty in specifying the display, timing on refresh rates, and being unstructured in the software engineering sense.

[Myers 1991] Brad Myers, "Graphical Techniques in a Spreadsheet for Specifying User Interfaces", 1991, CHI '91, New Orleans, Louisiana, 243-249, April 27-May 2, 1991.

Keywords: constraints, spreadsheets, generalization

Summary:

Discusses C32 VPL, a Lisp-based spreadsheet using constraints. Two contributions to scaling up issues: 1. Debugging: can change a cell value to an arbitrary value to observe its effect, regardless of the formula attached to the cell; formula remains unchanged. 2. Documentation/Understanding: arrows can be drawn showing the dependencies of a cell's formula on other cells. Generalized abstractions are written in LISP. The system substitutes formal parameters for concrete values used in the LISP code.

[Myers 1993] Brad Myers, "Peridot: Creating User Interfaces by Demonstration," Watch What I Do: Programming by Demonstration (A. Cypher, editor), MIT Press, Cambridge, Massachusetts, 125-154, 1993.

[Myers et al. 1996] Brad A. Myers, Robert C. Miller, Rich McDaniel, Alan Ferrency, "Easily Adding Animations to Interfaces Using Constraints", ACM Symposium on User Interface Software and Technology, Seattle, Washington, 119-128, November 6-8, 1996.

Keywords: constraints, animation, Amulet

Summary:

Amulet supports animations through an animation constraint. It demonstrates nicely and supports things like fade-in/fade-out, slow-in/slow-out. Own constraint solver.

[Najork and Golin 1990] Marc Najork and Eric Golin, "Enhancing Show-and-Tell with a Polymorphic Type System and Higher-order Functions", 1990 IEEE Workshop on Visual Languages, Skokie, Illinois, 215-220, October 1990.

[Najork and Kaplan 1993] Marc A. Najork and Simon M. Kaplan, "Cube or Programming in Three Dimensions", submitted to *Journal of Visual Languages and Computing*.

Summary:

A graphical, polymorphic, statically typed logical programming language with a three-dimensional syntax of cubes connected by pipes. Does not have Prolog's non-declarative forms cut and assert, but allows predicates to be passed as data and applied to arguments. Uses a Hindley-Milner type system, so explicit declaration of variable types is not needed. Comprises two sublanguages, one for type definition and one for predicate definition, that use the same syntactic forms. Colors are used to distinguish between classes of objects. The three-dimensional constructs are readily flattened out using slicing and unstacking.

[Najork 1996] Marc A. Najork, "Programming in Three Dimensions", *Journal of Visual Languages and Computing* 7 (2), 219-242, June 1996.

[Nardi and Zarmer 1993] Bonnie A. Nardi and Craig L. Zarmer, "Beyond Models and Metaphors: Visual Formalisms in User Interface Design", *Journal of Visual Languages and Computing* 4(1), 5-33, March 1993.
Keywords: visual Formalism, application framework, user interface design, spreadsheet
Summary:
ACE work by HP lab. Compared models and metaphors, felt they were both inadequate for design information-intensive interfaces.

[Nardi 1993] B. Nardi, A Small Matter of Programming: Perspectives on End User Computing, MIT Press, Cambridge, Massachusetts, 1993.

[Newbery 88] Frances J. Newbery, "An Interface Description Language for Graph Editor", 1988 IEEE Workshop on Visual Languages, Pittsburgh, Pennsylvania, 144-149, October 10-12, 1988.
Keywords: screen real estate, automatic layout
Summary:
Defines a language which describes the interface between an application and a graph editor. The language describes attributes of the graph. It also specifies the manner in which the graph is to be displayed (layout algorithm, layout constraints) and the set of editing commands.

[Newman and Lamming 1995] William M. Newman and Michael G. Lamming, Interactive System Design, Addison-Wesley, 1996.

[Nielsen and Molich 1990] J. Nielsen and R. Molich, "Heuristic Evaluation of User Interfaces", CHI '90, Seattle, Washington, 249-256, April 1-5, 1990.

[Nielsen 1992] J. Nielsen, "Finding Usability Problems Through Heuristic Evaluation", *CHI '92*, 373-380, 1992.

[Olsen 1996] Dan R. Olsen Jr., "Inductive Groups", ACM Symposium on User Interface Software and Technology, Seattle, Washington, 193-199, November 6-8, 1996.
Keywords: programming by demonstration
Summary:
Manipulating sets of related objects through inductive relationships among objects. Generalization on the idea of groups in drawing packages.

[Palmiter 1993] Susan Palmiter, "The Effectiveness of Animated Demonstrations for Computer-based Tasks: a Summary, Model and Future Research", *Journal of Visual Languages and Computing* 4(1), 71-89, March 1993.

[Pandey and Burnett 1993] Rajeev K. Pandey and Margaret M Burnett, "Is It Easier to Write Matrix Manipulation Programs Visually or Textually? An Empirical Study", 1993 IEEE Symposium on Visual Languages, Bergen, Norway, August 24-27, 1993.
Keywords: Forms/3, visual program comprehension
Classification: VPL-VI.C.1
Summary:
Experiment conducted on programmers solving vector and matrix manipulation tasks using the visual language Forms/3, the textual language Pascal and a textual matrix manipulation language, an alternate syntax version of APL. Program construction.

[Pau and Olason 1991] L. F. Pau and H. Olason, "Visual Logic Programming", *Journal of Visual Languages and Computing* 2(1), 3-15, March 1991.
Summary:
Describes a Prolog system in which the knowledge base can be represented and edited either as a graph or in conventional text form. The system provides facilities for converting from each form to the other. The visual complexity of the graph representation can be controlled by combining nodes, and there are facilities for zooming in on a part of the graph. The system also provides an automatic layout mechanism, which provides potential for improving readability.

[Pane and Myers 1996] John F. Pane and Brad A. Myers, "Usability Issues in the Design of Novice Programming Systems", Technical Report CMU-HCII-96-101, School of Computer Science, Carnegie Mellon University, August 1996.
Keywords: HCI, usability
Summary:
This report summarizes research about novice programming.

[Penz 1991] Franz Penz, "Visual Programming in the Object World", *Journal of Visual Languages and Computing* 2(1), 17-41, March 1991.
Summary:
Software development environment that combines object-oriented programming and visual programming to enable software reuse is presented.

[Petre and Green 1993] M. Petre and T.R. G. Green, "Learning to Read Graphics: Some Evidence that "Seeing" an Information Display is an Acquired Skill", *Journal of Visual Languages and Computing* 4(1), 55-70, March 1993.
Keywords: graphical representation

[Petre 1995] M. Petre, "Why Looking Isn't Always Seeing: Readership Skills and Graphical Programming", *Communications of the ACM* 38(6), 33-44, June 1995.

[Poswig et al. 1992] Jorg Poswig, Klaus Teves, Guido Vrankar, and Claudio Moraga, "Visa Vis - Contributions to Practice and Theory of Highly Interactive Visual Languages", 1992 IEEE Workshop on Visual Languages, Seattle, Washington, 155-161, September 15-18 1992.

Summary:

Scaling up issues addressed are procedural abstraction, compilation and execution efficiency, type checking, and screen real-estate. Procedural abstraction is performed by dragging the icon of a function from the editor into the workspace. Efficiency is addressed by linear time compilation as well as expression transformation. Usage of screen real-estate is reduced by using implicit typing and type inference, which eliminates need for on-screen type representations. Language itself is functional and based on the single and simple concept of substitution. Visual feedback during editing conveys to user the acceptable destinations for a selected function. Language is compiled from visual representation into FFP [Backus 1978].

[Poswig and Moraga 1993] Jorg Poswig and Claudio Moraga, "Incremental Type Systems and Implicit Parametric Overloading in Visual Languages", 1993 IEEE Symposium on Visual Languages, Bergen, Norway, August 24-27 1993.

Summary:

Discusses two aspects of the type system for the language VisaVis, a declarative visual language based on Backus's FFP and using a Milner-style type system. The two aspects of the type system discussed are the implementation of the system's incremental type checking, and a facility called "implicit parametric overloading". See [Burnett 1993a] on how incremental type checking relates to scaling up. Implicit parametric overloading addresses the problem of specifying that an otherwise free type must have certain operations defined on it.

[Potosnak 1988] Kathleen Potosnak, "Do Icons Make User Interfaces Easier to Use?", *IEEE Software Human Factors*, 97-99, May 1988.

Summary:

Results showed that new users performed better on command and menu systems than they did on the iconic systems. Useless help systems, obscure icons, and confusing modes hindered users.

[Purchase et al. 1995] Helen C. Purchase, Robert F. Cohen and Murray James, "Validating Graph Drawing Aesthetics", 1995 Graph Drawing Conference. Also in Lecture Notes in Computer Science (F. Brandenburg. ed.), Springer Verdag, 1995.

Keywords: graph drawing

Summary:

Line crossing is bad in representation of graphs

[Rasure and Williams 1991] John R. Rasure and Carla S. Williams, "An Integrated Data Flow Visual Language and Software Development Environment", *Journal of Visual Languages and Computing* 2(2), 217-246, June 1991.
Summary:
Describes cantata, a large-scale graphical dataflow language embedded in the development environment Khoros. Individual operations associated with the nodes of the graph are routines programmed in a textual language (FORTRAN or C) and stored in libraries. Cantata is not a pure dataflow language, having loop constructs with termination conditions that can be dependent on global variables. Variables and arithmetic and Boolean expressions are entered and represented textually for ease of entry and to reduce visual clutter. Cantata addresses name-space partitioning by providing a three-level hierarchy for accessing library routines. The language provides procedural abstraction via nested dataflow graphs representing procedure definitions. These nested procedure definitions can also have variable declarations, and the language uses scoping rules like those of Pascal. Debugging facilities include highlighting the currently executing node in a graph, ability to interrupt and resume execution, ability to change state information in a halted program, and a single-step mode.

[Repenning 1994] Alex Repenning, "Bending Icons: Syntactic and Semantic Transformations of Icons", 1994 IEEE Symposium on Visual Languages, St. Louis, Missouri, 296-303, Oct. 4-7, 1994.

[Repenning 1995] Alex Repenning, "Bending the Rules: Steps Toward Semantically Enriched Graphical Rewrite Rules", 1995 IEEE Symposium on Visual Languages, Darmstadt, Germany, 226-233, Sept. 5-9, 1995.
Keywords: Agentsheets

[Repenning and Ambach 1996] Alex Repenning and J. Ambach, "Tactile Programming: A Unified Manipulation Paradigm Supporting Program Comprehension, Composition and Sharing", 1996 IEEE Symposium on Visual Languages, Boulder, Colorado, 102-109, September 3-6, 1996.
Keywords: Visual Agentsheets

[Riecken 1993] Douglas Rieken, "End-User Programming Paradigms: Visual Tool-based Programming Versus Scripting", Draft paper, March 1993.
Keywords: Direct manipulative tool-based language
Classification: VPL-VI.C.1
Summary:
A study to empirically compare two end-user programming paradigms: a visual direct manipulative tool-based programming language versus an end-user programming scripting language. Three characteristics were used to evaluate the subjects' performance: 1) Ability to complete a given task, 2) Correctly implement the program to perform the task and 3) the time that takes to complete the exercise. Motivation: Interest in the application of a visual language whose language constructs are small simple software tools which can be combined to perform a specific action. Assumed that tool-based VPLs might yield significant results in end-user programming performance. Expected the subjects (40) to do better visually over scripting. The objective was to observe and quantify subject performance to solve programming problems related to end-user design and implementation of computer programs. Findings suggest end-user programming performance was significantly better for those who use visual software tools. Visual software tools won over Unix script language.

[Robertson et al. 1991] George G. Robertson, Jock D. Mackinlay and Stuart K. Card, "Cone Trees: Animated 3d Visualizations of Hierarchical Information", CHI '91, New Orleans, Louisiana, 189-194, April 27- May 2, 1991.

Keywords: interface metaphors, screen layout

[Rogers 1988] Greg Rogers, "Visual Programming with Objects and Relations," 1988 IEEE Workshop on Visual Languages, Pittsburgh, Pennsylvania, 29-36, October 10-12, 1988.

Summary:

Discuss a VPL based on object-oriented programming languages and relational data bases. The essence of programming is editing data structures. The edit operations performed on these data structures is specified through programming-by-example. This is done by laying out a sequence of message passings, conditional and looping structures, precondition expressions and edit actions on the data object. Complex objects are built by combining objects with relation.

[Rogers 1990] Greg Rogers, "The GRClass Visual Programming System", 1990 IEEE Workshop on Visual Languages, Skokie, Illinois, 48-53, October 1990.

[Schafer 1988] Alice L. Schafer, "Graphical Interactions with an Automatic Programming System", *IEEE Transactions on Systems, Man, and Cybernetics* 18(4), 575-591, July/August 1988.

Summary:

Describes a graphical interface, IGI, to an automatic programming system, ISFI. IGI graphically represents and permits graphical editing of constraint networks constructed "cooperatively" by the computer and the programmer. IGI provides graphical views of the ISFI object hierarchy and the state-transition diagram representation of the constraint network, but these views cannot be edited. The system supports queries on the relationships between generated code and aspects of the constraint network. IGI is intended to be used by programmers who are knowledgeable about the structure and operation of ISFI.

[Schoberth 1990] Andreas Schoberth, "Event Handling in a Demand-driven Visual Language Preserving Single Assignment", Master's Thesis, Department of Computer Science, University of Kansas, 1990.

[Sears 1993] A. Sears, "Layout Appropriateness: A Metric for Evaluating User Interface Widget Layout", *IEEE Transactions on Software Engineering*, 19(7), 707-719, July 1993.

[Sebesta 1993] Robert Sebesta, Concepts of Programming Languages, Second Edition, Benjamin/Cummings, Redwood City, California, 1993.

[Selker and Koved 1988] Ted Selker and Larry Koved, "Elements of Visual Language", 1988 IEEE Workshop on Visual Languages, Pittsburgh, Pennsylvania, 38-44, October 10-12, 1988.

Summary:

Provides a definition of the elements of visual language.

[Shaw et al. 1983] Mary Shaw, Ellen Borlson, Michael Horowitz, Tom Lane, David Nichls, Randy Pausch, "Descartes: A Programming-Language Approach to Interactive Display Interfaces", SIGPLAN Notices 18(6), 100-111.

[Shneiderman 1996] Ben Shneiderman, "The Eyes Have It: A Task By Data Type Taxonomy for Information Visualization", 1996 IEEE Symposium on Visual Languages, Boulder, Colorado, 336-343, September 3-6, 1996.

[Shu 1985] Nan C. Shu, "FORMAL: A Forms-Oriented, Visual-Directed Application Development System", *Computer* 18(8), 38-49, August 1985.
Summary:
Early paper on declarative form-based development system exhibiting good ideas for scaling up: Data abstraction, open ended application size, compact screen representation, and some error handling. Other good ideas, not necessarily pertaining to scaling up are: automatic data restructuring, extensibility via libraries, familiar form paradigm.

[Singh and Green 1989] Germinder Singh and Mark Green, "A High-level User Interface Management System", CHI '89, Austin, Texas, 133-138, April 30 - May 4, 1989.

[Siochi and Hix 1991] A. Siochi and D. Hix, "A Study of Computer-Supported User Interface Evaluation Using Maximal Repeating Pattern Analysis", CHI '91, New Orleans, Louisiana, 301-305, April 27-May 2, 1991.

[Smith 1990] David N. Smith "The Interface Construction Set," in Visual Languages and Applications, (T. Ichikawa, E. Jungert, R. Korfhage, eds.), 31-51, Plenum Publishing Corp., New York, 1990.
Keywords: visual dataflow language, direct manipulation, event handling

[Spenke et al. 1996] Michael Spenke, Christian Beilken, Thomas Berlage, "FOCUS: The Interactive Table for Product Comparison and Selection", ACM Symposium on User Interface Software and Technology, Seattle, Washington, 41-50, November 6-8, 1996.
Keywords: data visualization in a table
Summary:
Interactive table viewer. It combines a dynamic query mechanism with fisheye. It displays the whole catalog and all of the data in a table and allows the user to focus on parts of interest.

[Stasko 1991] John T. Stasko, "Using Direct Manipulation to Build Algorithm Animations by Demonstration", CHI '91, New Orleans, Louisiana, April 27 - May 2, 1991.
Keywords: program visualization, program animation

[Stasko et al. 1993] John T. Stasko, Albert Badre and Clayton Lewis, "Do Algorithm Animations Assist Learning? An Empirical Study and Analysis", InterCHI '93, 1993.
Summary:
Conducted an empirical study of a priority queue algorithm animation and the study results indicate that the animation only slightly assisted student understanding. Explained why algorithm animations may not be as helpful as was initially hoped.

[Stasko and Muthukumarasamy 1996] J. Stasko and J. Muthukumarasamy, "Visualizing Program Executions on Large Data Sets", 1996 IEEE Symposium on Visual Languages, Boulder , Colorado, 166-173, September 3-6, 1996.
Keywords: data visualization
Summary:
Technique for visualizing large data sets.

[Stefik et al. 1986] Mark J. Stefik, Daniel G. Bobrow and Kenneth Kahn, "Integrating Access-oriented programming into a multiparadigm environment", *IEEE Software* 3(1), 10-18, January 1986.

[Sugiura and Koseki 1996] Atsushi Sugiura and Yoshiyuki Koseki, "Simplifying Macro Definition in Programming by Demonstration", ACM Symposium on User Interface Software and Technology, Seattle, Washington, 173-182, November 6-8, 1996.
Keywords: programming by demonstration
Summary:
Generalization very limited. Two techniques are described: 1. action slicing extracts from history the user actions that affected the creation of specific data. 2. Macro auto-definition detects user actions that are expected to be performed again in the future and converts them automatically into macro.

[Tanimoto 1990] Steven Tanimoto, "VIVA: A Visual Language for Image Processing", *Journal of Visual Languages and Computing* 1(2), 127-139, June 1990.
Keywords: liveness levels

[Tanimoto and Glinert 1990] Steven Tanimoto and Ephraim P. Glinert, "Designing Iconic Programming Systems: Representation and Learnability," in Visual Programming Environments: Applications and Issues (ed. E.P. Glinert), 330-336, 1990.

[Tolmach and Appel 1993] Andrew Tolmack and Andrew W. Appel, "A Debugger for Standard ML", *Journal of Functional Programming*, 1(1), 1-47, January 1993.
Keywords: reverse execution, explicit approach to time

[Tufte 1983] Edward R. Tufte, The Visual Display Of Quantitative Information, Graphics Press, Cheshire, Connecticut, 1983.
Keywords: perception rules

[Ungar and Smith 1987] David Ungar and Randall Smith, "Self: The Power of Simplicity," OOPSLA '87, 227-242, October 4-8, 1987.

[Vander Zanden and Venckus 1996] Brad Vander Zanden and Scott Venckus, "An Empirical Study of Constraint Usage in Graphical Applications", ACM Symposium on User Interface Software and Technology, Seattle, Washington, 137-146, November 6-8, 1996.
Keywords: Constraints
Summary:
The study of one-way constraints used in various Amulet applications. They found the constraints to be highly modular. They also found evidence to support lazy evaluation and memoization because of many repeated evaluations of the same constraints.

[Viehstaedt and Ambler 1992] Gerhard Viehstaedt and Allen L. Ambler, "Visual Representation and Manipulation of Matrices", *Journal of Visual Languages and Computing* (3)3, 273-298, September 1992.
Summary:
Uses features of Forms/3 to show matrix manipulations in a visual representation. Screen navigation vocabulary (view, regions, etc.)

[Vion-Dury and Santana 1994] Jean-Yves Vion-Dury and Miguel Santana, "Interactive Visualization of Distributed Object-Oriented Systems", OOPSLA '94, Portland, Oregon, 65-84, October 23-27, 1994.

Keyword: virtual images, information visualization, 3D animation.

Summary:

Describes a graphical user interface to visualize a large, complex set of objects. A 3D animation technique is used to represent large numbers of objects, complex relationships and dynamic execution of concurrent activities. It claims that 3D workspaces are intrinsically better for keeping good locality of objects and animation is used to preserve the perceptual continuity

[Virzi 1992] R. Virzi, "Refining The Test Phase Of Usability Evaluation: How Many Subjects Is Enough?", *Human Factors* 34(4), 457-468, 1992.

Keyword: usability studies, subject size

Summary:

5 subjects is sufficient to uncover 80% of usability problems. Adding more subjects are less likely to uncover more and most severe problems are found in the first few subjects.

[Wadge and Ashcroft 1985] W. Wadge and E. Ashcroft, Lucid, the Dataflow Programming Language, Academic Press, London, 1985.

[Waite 1989] Kevin W. Waite, "A Graphical Environment for Formally Specifying Abstract Data Types", Graphics Tools for Software Engineers (A. Kilgour and R. Earnshaw, eds.), Cambridge University Press, 71-92, 1989.

[Wang and Ambler 1996] G. Wang and A. Ambler, "Solving Display-Based Problems", 1996 IEEE Symposium on Visual Languages, Boulder, Colorado, 122-129, Sept. 3-6, 1996.

Keywords: Formulate

[Ware 1993] Colin Ware, "The Foundations of Experimental Semiotics: a Theory of Sensory and Conventional Representation", *Journal of Visual Languages and Computing* 4(1), 91-100, March 1993.

[Wilde and Lewis 1990] Nicholas Wilde and Clayton Lewis, "Spreadsheet-based Interactive Graphics: From Prototype to Tool," CHI '90, Seattle, Washington, 153-159, April 1990.

Keywords: spreadsheet computational model, programming environments, event handling related

Summary:

An extension to spreadsheet for supporting interactive graphics. No facilities for generalized abstractions.

[Williams and Rasure 1990] Carla Williams and John Rasure, "A Visual Language for Image Processing", 1990 IEEE Workshop on Visual Languages, Skokie, Illinois, 86-91, October 1990.

[Winograd 1995] T. Winograd, "From Programming Environments to Environments For Designing", *Communications of the ACM* 38(6), 65-74, June 1995.

[Yang and Burnett 1994] Sherry Yang and Margaret M. Burnett, "From Concrete Forms to Generalized Abstractions Through Perspective-Oriented Analysis of Logical Relationships," Oregon State University Computer Science Technical Report 94-60-3, April 1994, revised July 1994. Also in 1994 IEEE Symposium on Visual Languages, St. Louis, Missouri, 6-14, October 4-7, 1994.
Keywords: generalization, concreteness, direct manipulation
Summary:
This paper describes the generalization algorithm in Forms/3.

[Yang et al. 1995] Sherry Yang, Margaret M. Burnett, Elyon DeKoven and Moshé Zloof, "Representation Design Benchmarks: a Design-Time Aid for VPL Navigable Static Representations", Oregon State University Computer Science Technical Report 95-60-3, August 1995.
Keywords: static representation, ICBE, Forms/3, cognitive dimensions

[Yang et al. 1996] Sherry Yang, Elyon DeKoven and Moshé Zloof, "Design Benchmarks for VPL Static Representations", 1996 IEEE Symposium on Visual Languages, Boulder, Colorado, 263-264, September 3-6, 1996.
Keywords: static representation, ICBE, Forms/3, cognitive dimensions

[Zarmer and Chow 1992] Craig L. Zarmer and Chee Chow, "Frameworks for Interactive, Extensible, Information-intensive Applications", ACM Symposium on User Interface Software and Technology, Monterey, California, 33-41, November 15-18, 1992.
Keywords: Application frameworks, UI toolkits, UIMS, builders, end user programming
Summary:
HP Lab's ACE work. Visual formalism.

[Zloof 1977] M. Zloof, "Query By Example: A Data Base Language", *IBM Systems Journal* 16 (4), 324-343, 1977.

[Zloof 1981] M. Zloof, "QBE/OBE: A Language For Office And Business Automation", *Computer*, 13-22, May 1981.

[Zloof and Krishnamurthy 1994] M. Zloof and R. Krishnamurthy, "IC By Example: Empowering The Uninitiated To Construct Database Applications", Technical Report, Hewlett Packard Laboratories, June 1994.

[van Zee et al. 1996] P. van Zee, M. Burnett and M. Chesire, "Retiring Superman: Handling Exceptions Seamlessly in a Declarative Visual Programming Language", 1996 IEEE Symposium on Visual Languages, Boulder, Colorado, 222-230, September 3-6, 1996.
Keywords: exception handling

APPENDICES

APPENDIX A COGNITIVE DIMENSIONS

Table A1 lists the dimensions, along with a thumb-nail description of each, and Figure A1 shows an example of using CDs to contrast the VPLs Prograph and LabVIEW. The relation of each dimension to a number of empirical studies and psychological principles is given in [Green and Petre 1995], but the authors also carefully point out the gaps in this body of underlying evidence. In their words, "The framework of cognitive dimensions consists of a small number of terms which have been chosen to be easy for non-specialists to comprehend, while yet capturing a significant amount of the psychology and HCI of programming."

Abstraction gradient	What are the minimum and maximum levels of abstraction? Can fragments be encapsulated?
Closeness of mapping	What 'programming games' need to be learned?
Consistency	When some of the language has been learnt, how much of the rest can be inferred?
Diffuseness	How many symbols or graphic entities are required to express a meaning?
Error-proneness	Does the design of the notation induce 'careless mistakes'?
Hard mental operations	Are there places where the user needs to resort to fingers or penciled annotation to keep track of what's happening?
Hidden dependencies	Is every dependency overtly indicated in both directions? Is the indication perceptual or only symbolic?
Premature commitment	Do programmers have to make decisions before they have the information they need?
Progressive evaluation	Can a partially-complete program be executed to obtain feedback on "How am I doing"?
Role-expressiveness	Can the reader see how each component of a program relates to the whole?
Secondary notation	Can programmers use layout, color, or other cues to convey extra meaning, above and beyond the 'official' semantics of the language?
Viscosity	How much effort is required to perform a single change?
Visibility	Is every part of the code simultaneously visible (assuming a large enough display), or is it at least possible to compare any two parts side-by-side at will? If the code is dispersed, is it at least possible to know in what order to read it?

Table A1. The Cognitive Dimensions (extracted from [Green and Petre 1995]).

"**Application:** In contrast to text languages, the box-and-line representation of data flow [does] really well at a local level the lines making the local data dependencies clearly visible. Both LabVIEW and Prograph therefore do well in avoiding the problem. LabVIEW uses virtually no variables at all, whereas Prograph has persistents which can act as global variables. These are different positions in the 'design space'. The Prograph position is presumably that if no globals at all are allowed, the program will get cluttered with too many lines.

But although local dependencies are made visible, long-range data dependencies are a different issue. Prograph has an extraordinarily large number of long-range hidden dependencies, created by the combination of a deep nesting with the lack of an overview of the nesting structure. Although the programmer can quickly navigate down the call graph by clicking on method icons to open their window, then clicking on the icons found there, etc., there is no way to proceed *up* the call graph in the same way. In general, to discover which method calls a given method, and thereby to determine its preconditions, can require an extensive search. To alleviate the difficulty, a searching tool is provided; it would be interesting to know how successful the tool is with expert users."

Figure A1. CDs are geared toward high-level discussion of the cognitive aspects of VPLs. In this example, the Hidden Dependencies dimension is being used to evaluate Prograph and LabVIEW (extracted from [Green and Petre 1995]).

APPENDIX B SAMPLE INTERPRETATION OF BENCHMARK RESULTS

Each designer interprets the benchmark results according to their particular design goals. A useful way to go about this is to devise a table of interpretation schemes such as Table A1, to use with the results. With such a table, tracking the improvements that come from different design alternatives is straightforward.

Table B1. One designer's mapping from benchmark results to subjective ratings.

Benchmark	S_c	S_p	NI	Aspect of the Representation	Example Rating Scale
D1		x		Visibility of dependencies	Ratio = Poor Fair Good 0.0 0.5 1.0
D2			x		# steps = Poor Fair Good >n n 0
PS2			x	Visibility of program structure	# steps = Poor Fair Good >n n 0
L2			x	Visibility of program logic	# steps = Poor Fair Good >n n 0
L3	x				# = Poor Good ≥1 0
R2			x	Display of results with program logic	# steps = Poor Fair Good >n n 0
SN1		x		Secondary notation: non-semantic devices	Ratio = Poor Fair Good 0.0 0.5 1.0
SN2			x		# steps = Poor Fair Good >n n 0

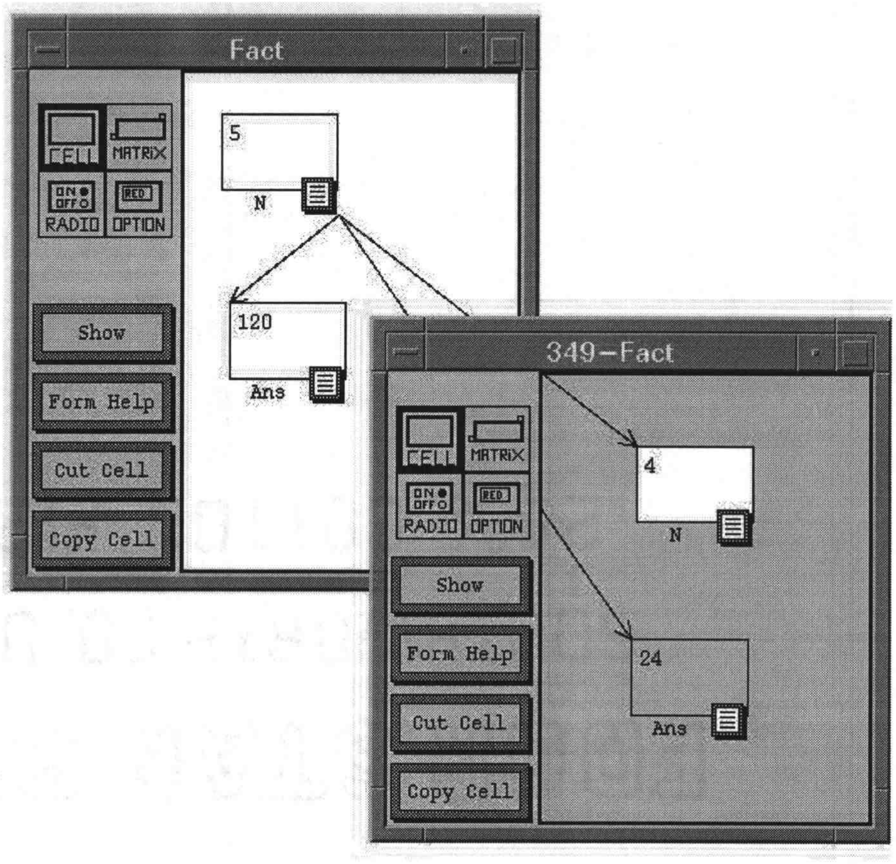
Table B1. Continued.

Benchmark	S_c	S_p	NI	Aspect of the Representation	Example Rating Scale
AG1		x		Abstraction gradient	Ratio = Poor |____ Fair |____ Good | ; 0.0 – 0.5 – 1.0
AG2			x		# steps = Poor |____ Fair |____ Good | ; >n – n – 0
RI2			x	Accessibility of related information	# steps = Poor |____ Fair |____ Good | ; >n – n – 0

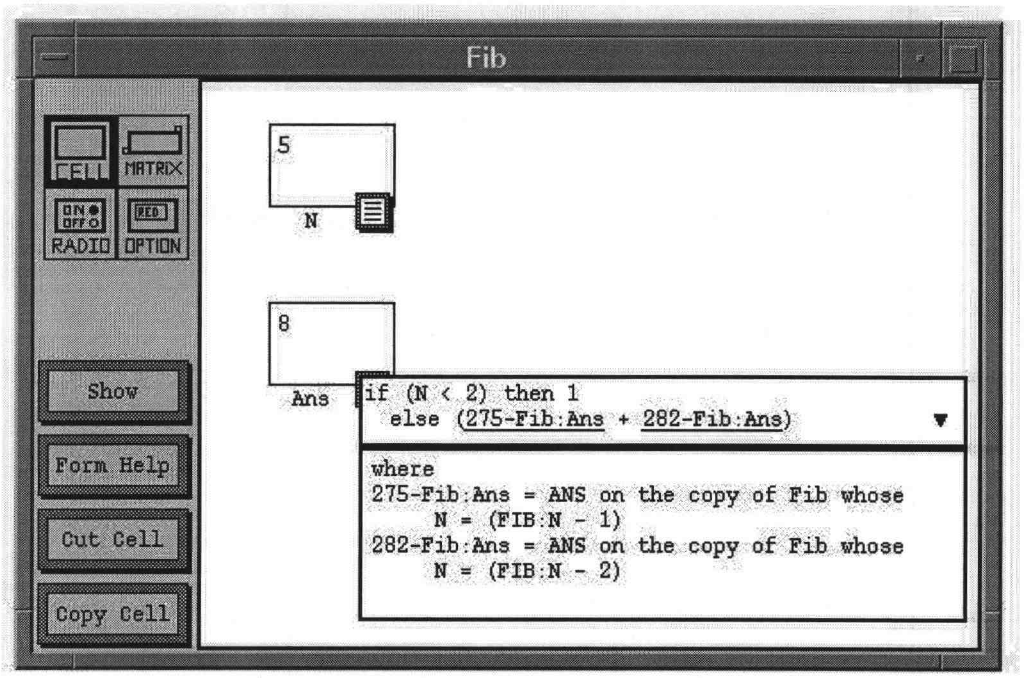
Table B1. One designer's mapping from benchmark results to subjective ratings. Not all benchmarks were rated by this designer, because some simply provide data points for comparison with other data points and have no natural mapping to subjective ratings.

APPENDIX C PROGRAM LISTINGS

Factorial



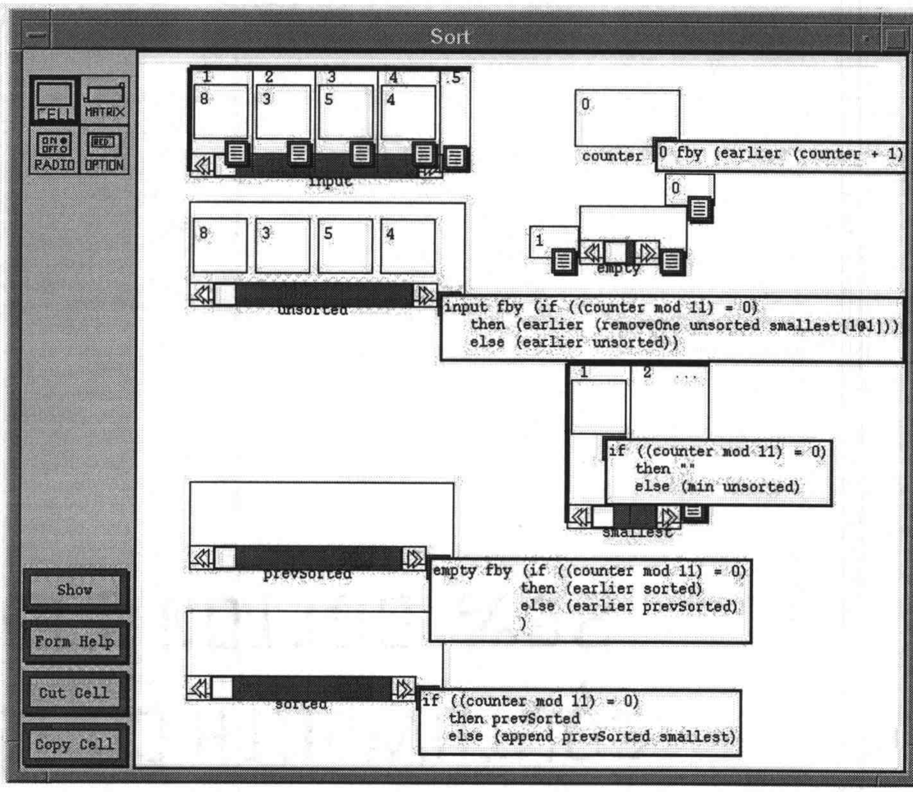
Fibonacci



```
if (N < 2) then 1
  else (275-Fib:Ans + 282-Fib:Ans)
```

```
where
275-Fib:Ans = ANS on the copy of Fib whose
      N = (FIB:N - 1)
282-Fib:Ans = ANS on the copy of Fib whose
      N = (FIB:N - 2)
```

<u>Animated Matrix Sort with Color Boxes</u>



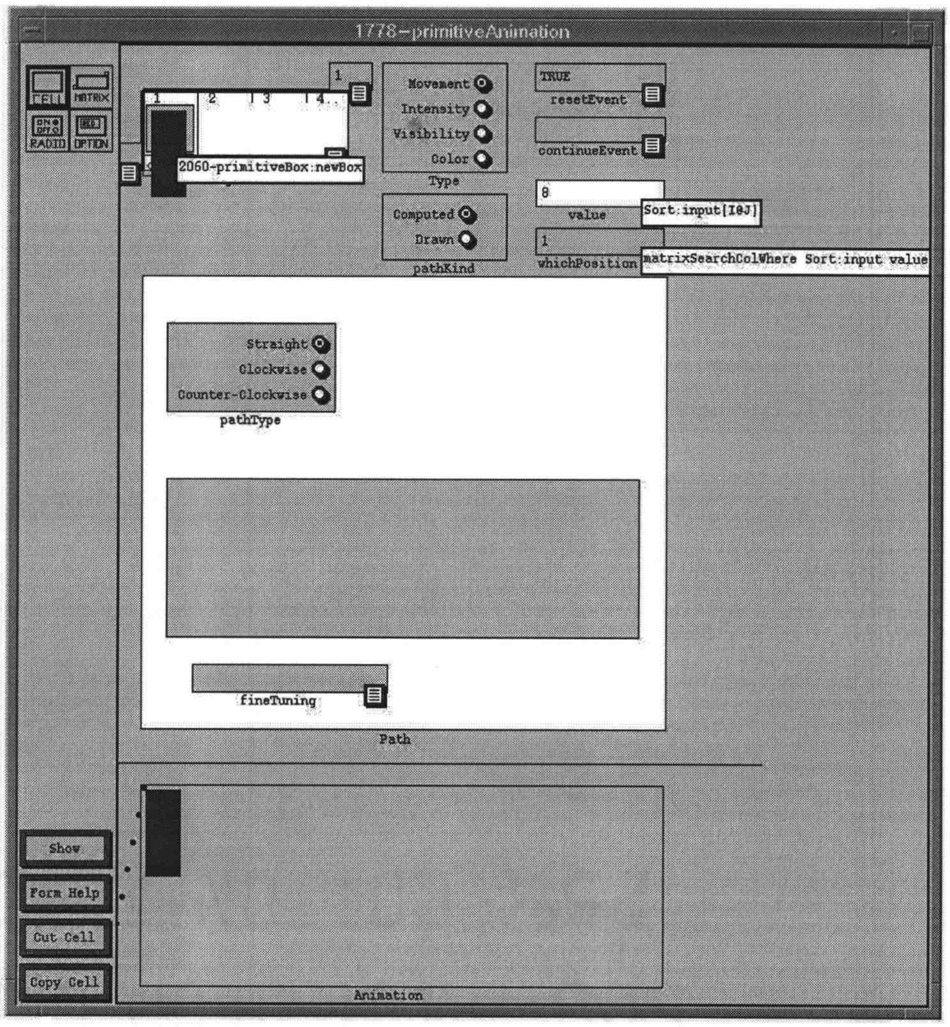
```
0 fby (earlier (counter + 1)
```

```
input fby (if ((counter mod 11) = 0)
   then (earlier (removeOne unsorted smallest[1@1]))
   else (earlier unsorted))
```

```
if ((counter mod 11) = 0)
   then ""
   else (min unsorted)
```

```
empty fby (if ((counter mod 11) = 0)
            then (earlier sorted)
            else (earlier prevSorted)
            )
```

```
if ((counter mod 11) = 0)
   then prevSorted
   else (append prevSorted smallest)
```



1 2 3 4..

1

2060-primitiveBox:newBox

Type: [✓ Movement] Intensity Visibility Color

pathKind: [✓ Computed] Drawn

resetEvent: TRUE

continueEvent:

value: 8

Sort:input[I@J]

whichPosition: 1

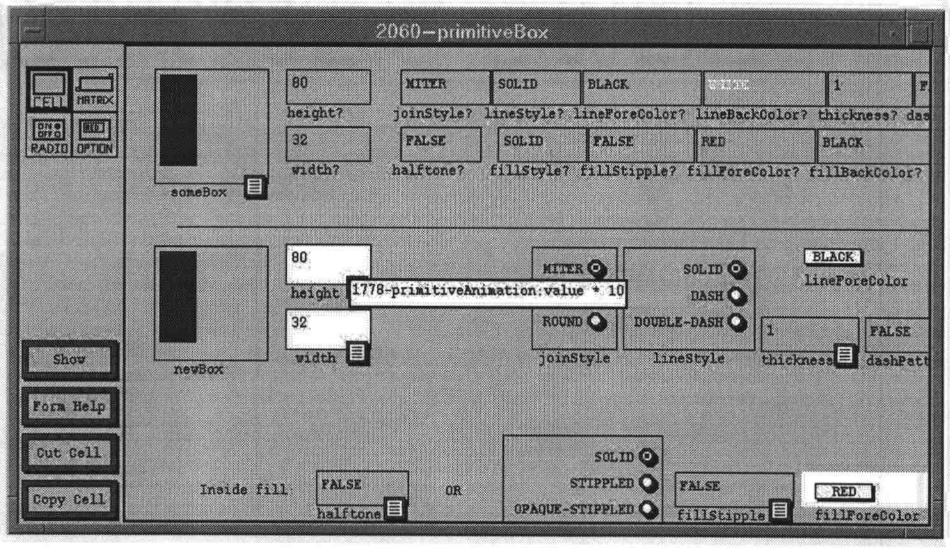
matrixSearchColWhere Sort:input value

pathType: [✓ Straight] Clockwise Counter-Clockwise

fineTuning

Path

Animation



someBox

80	MITER	SOLID	BLACK	WHITE	1	F
height?	joinStyle?	lineStyle?	lineForeColor?	lineBackColor?	thickness?	das
32	FALSE	SOLID	FALSE	RED	BLACK	
width?	halftone?	fillStyle?	fillStipple?	fillForeColor?	fillBackColor?	

newBox

height: 80

1778-primitiveAnimation:value * 10

width: 32

joinStyle: [✓ MITER] ROUND

lineStyle: [✓ SOLID] DASH DOUBLE-DASH

lineForeColor: BLACK

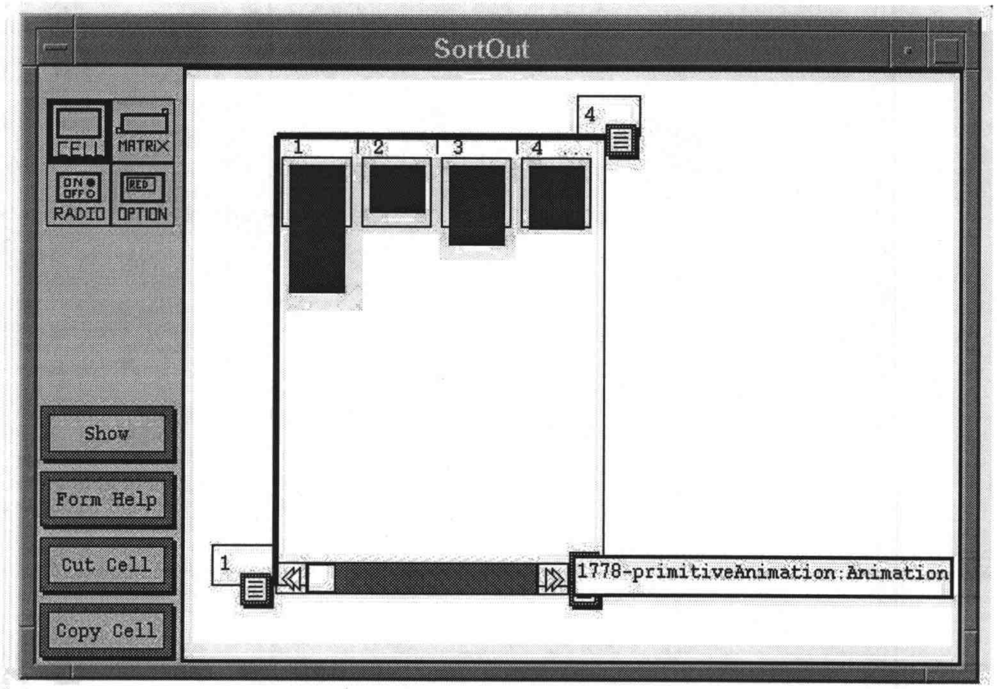
thickness: 1

dashPatt: FALSE

Inside fill halftone: FALSE OR [✓ SOLID] STIPPLED OPAQUE-STIPPLED

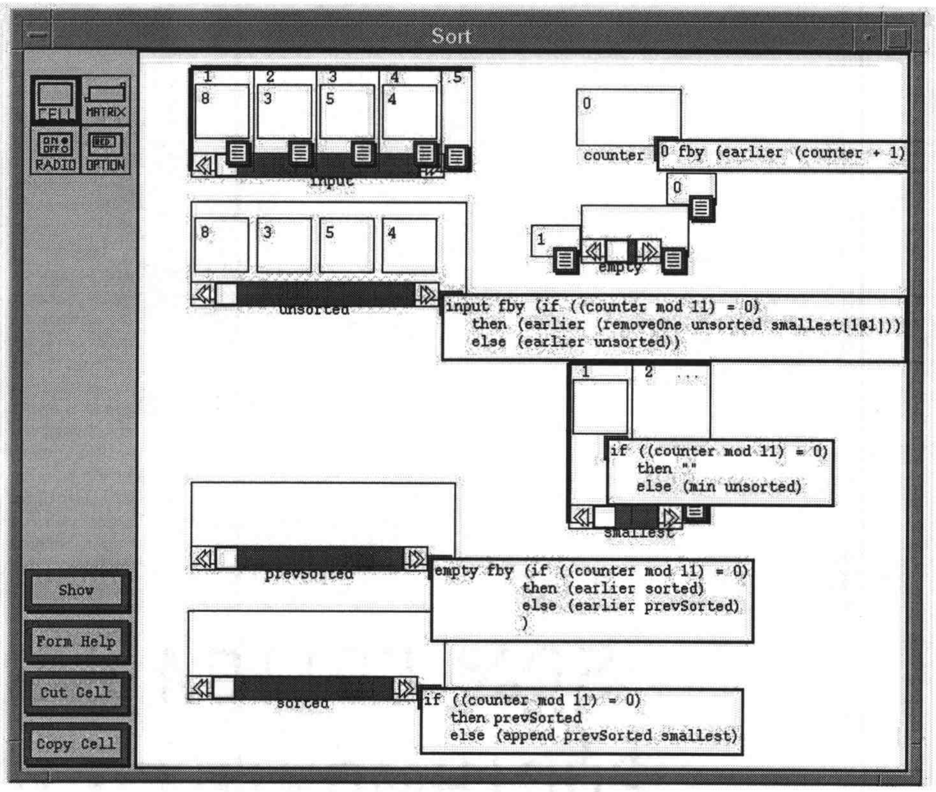
fillStipple: FALSE

fillForeColor: RED



1778-primitiveAnimation:Animation

Animated Matrix Sort without Color Boxes



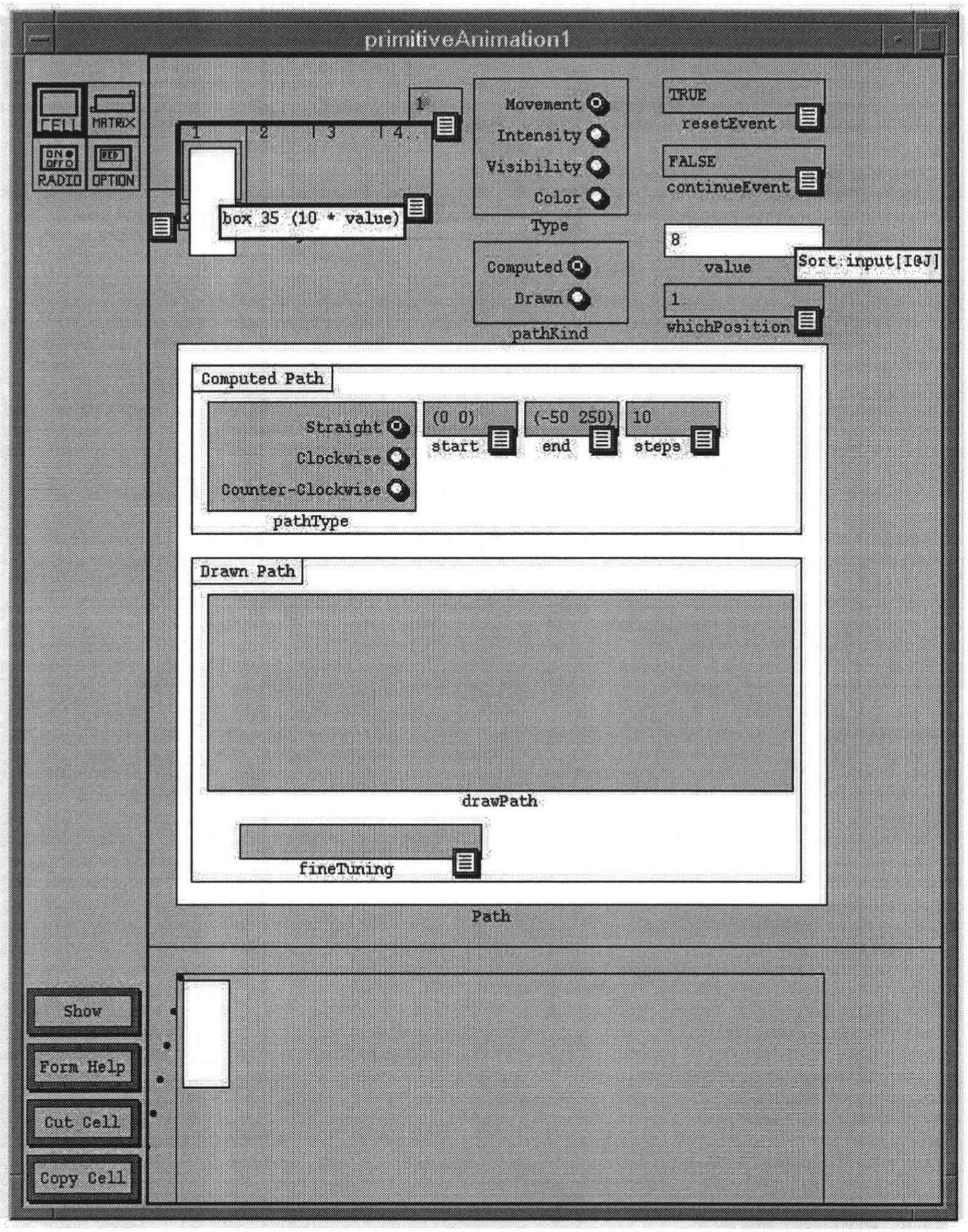
```
0 fby (earlier (counter + 1)
```

```
input fby (if ((counter mod 11) = 0)
   then (earlier (removeOne unsorted smallest[1@1]))
   else (earlier unsorted))
```

```
if ((counter mod 11) = 0)
   then ""
   else (min unsorted)
```

```
empty fby (if ((counter mod 11) = 0)
           then (earlier sorted)
           else (earlier prevSorted)
           )
```

```
if ((counter mod 11) = 0)
   then prevSorted
   else (append prevSorted smallest)
```



box 35 (10 * value)

TRUE
resetEvent

FALSE
continueEvent

8
value

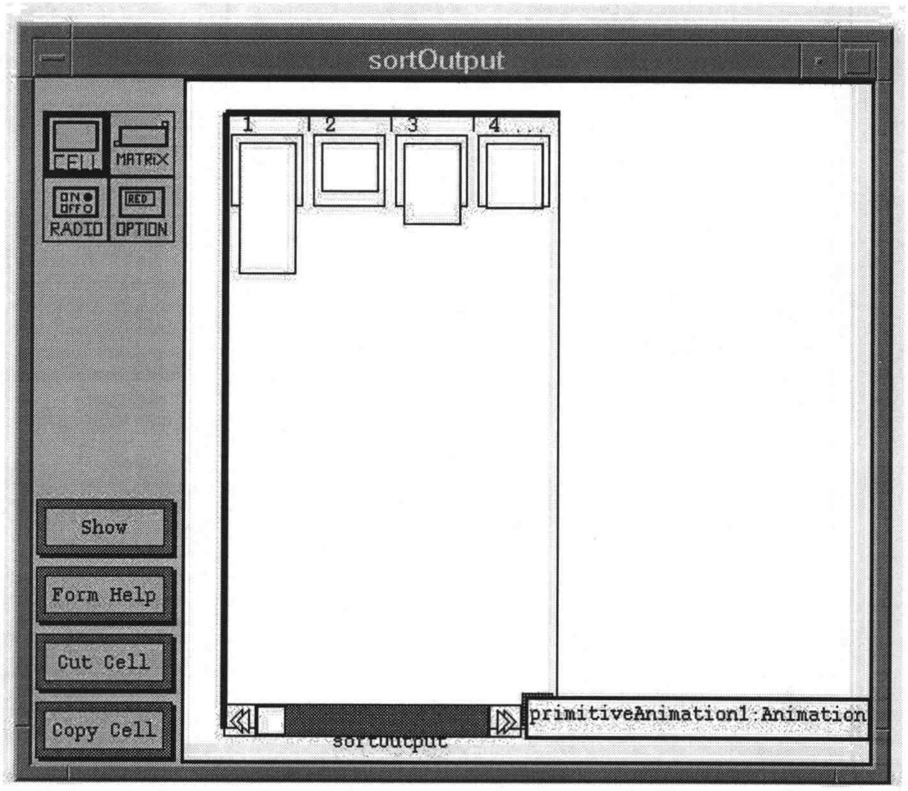
Sort input[I@J]

1
whichPosition

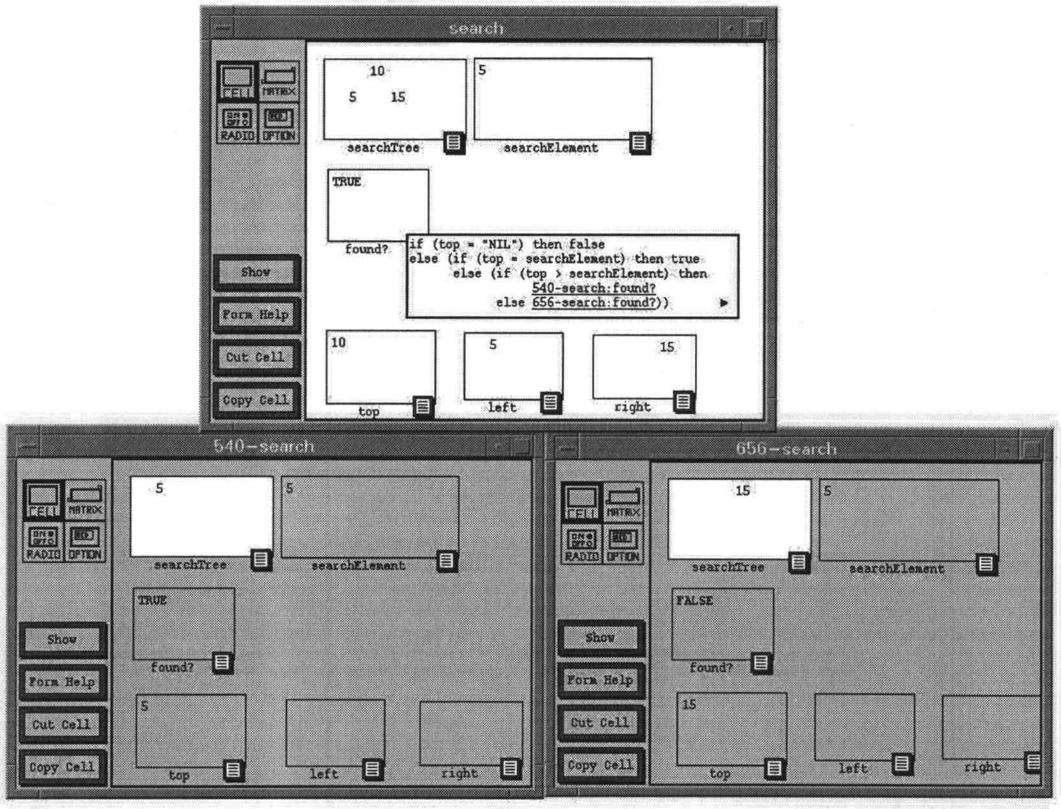
(0 0)
start

(-50 250)
end

10
steps


sortOutput
CELL
MATRIX
ON
OFF
RADIO
RED
OPTION
Show
Form Help
Cut Cell
Copy Cell
1
2
3
4 . . .
sortOutput
primitiveAnimation1:Animation

Binary Tree Search



```
if (top = "NIL") then false
else (if (top = searchElement) then true
      else (if (top > searchElement) then
                  540-search:found?
            else 656-search:found?))
```