Journal of Visual Languages and Computing (1997) **8,** 563–599

Representation Design Benchmarks: A Design-Time Aid for VPL Navigable Static Representations

SHERRY YANG*, MARGARET M. BURNETT*,†,
ELYON DEKOVEN‡ AND MOSHÉ ZLOOF‡

**Department of Computer Science, Oregon State University, Corvallis, OR 97331-3202 U.S.A.*
yang@cs.orst.edu, burnett@cs.orst.edu
‡Hewlett-Packard Labs, P.O. Box 10490, Palo Alto, CA 94303-0969 U.S.A.
dekoven@hpl.hp.com, zloof@hpl.hp.com

Received 18 August 1995; revised 10 March 1997; accepted 8 April 1997

A weakness of many interactive visual programming languages (VPLs) 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.

1. Introduction

Many visual programming languages (VPLs) 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 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

† This work was supported in part by Hewlett-Packard and by the National Science Foundation under grant CCR-9308649 and an NSF Young Investigator Award.

1045-926X/97/050563+37 $25.00/0/vl970047

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 [1–3].

In this paper, we describe *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. The benchmarks are currently being used at Oregon State University and at Hewlett-Packard Laboratories to design new static representations for the languages Forms/3 [4] and ICBE [5, 6].

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 [7, 8]. 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.

2. Related work

Cognitive dimensions (CDs) [7, 8] 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 [9] and LabVIEW [10] (see Appendix A for an excerpt). Modugno used CDs to evaluate Pursuit, a research programming-by-demonstration VPL [11] and Yazdani and Ford used CDs to evaluate PrologSpace, a general-purpose visual programming

system [12]. Hendry also used CDs to evaluate cognitive aspects of a modification to spreadsheet formula languages [13].

CDs are one of the two design-time evaluation approaches that have been applied to VPLs [14]. The other is the programming walkthrough [15, 16]. 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.

In the realm of software metrics, Glinert introduced a framework for formulating software metrics to compare visual computing environments [17]. 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.

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 a representation. GOMS, pattern analysis, heuristic evaluation and layout appropriateness are a few such methods. For example, GOMS [18] is a detailed methodology for giving quantitative time predictions for the user to perform activities defined as a detailed sequence of simple tasks such as 'delete a word'. Maximal repeating pattern analysis [19] 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. Heuristic evaluation (HE) [20, 21] is a general evaluative technique that rates a user interface through a set of nine usability principles, such as 'use simple and natural dialogue', 'speak the user's language', and 'minimize user memory load'. Layout appropriateness [22] 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 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.

3. 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.

3.1. Navigable Static Representations

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 the remainder of this paper, the term 'representation' when used alone should be read as an abbreviation of the more cumbersome phrase 'navigable static representation'.

3.1.1. Applying the Definition: An Example

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 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*. 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*. 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.

3.1.2. Implications of the Definition

As this example demonstrates, there are elements of VPLs that are neither in *S* nor in NI. Examples include animations, 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.

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.

3.2. 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 S_c) or (2) the presence (denoted S_p) of the elements of a static representation *S.* For example, the Closeness of Mapping CD pertains to characteristics of static representation elements (S_c), 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 (S_p) 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 S_c and S_p measures. In addition, for each S_p 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.

Finally, we conducted an empirical study to learn more about the suitability of the benchmarks as a design aid. The benchmarks are summarized in Table 1. Sections 4 and 5 discuss the relationships of each of the benchmarks with their corresponding CDs and how to apply them, and Section 6 describes the empirical study.

3.3. Use of the Representation Design Benchmarks: Why, When, Who and How

The purpose of the representation design benchmarks is to provide a set of design-time measures that VPL designers can use to measure and improve their design ideas for navigable static representations in their languages. We wanted a design-time approach

Table 1. Summary of the representation design benchmarks. S_c denotes measures of the characteristics of elements of S. S_p denotes measures of the presence of potential elements of S. Each S_p measure has a corresponding NI measure

Benchmark name	S_c	S_p	NI	Aspect of the representation	Computation
D1		×		Visibility of dependencies	(Sources of dependencies explicitly depicted)/(Sources of dependencies in system)
D2			×		The worst-case number of steps required to navigate to the display of dependency information
PS1		×		Visibility of program structure	Does the representation explicitly show how the parts of the program logically fit together? Yes/No
PS2			×		The worst-case number of steps required to navigate to the display of the program structure
L1		×		Visibility of program logic	Does the representation explicitly show how an element is computed? Yes/No
L2			×		The worst-case number of steps required to make all the program logic visible
L3	×				The number of sources of misrepresentations of generality
R1		×		Display of results with program logic	Is it possible to see results displayed statically with the program source code? Yes/No
R2			×		The worst-case number of steps required to display the results with the source code.
SN1		×		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			×		The worst-case number of steps to access secondary notations
AG1		×		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		×		The worst-case number of steps to abstract away the details
RI1		×	Accessibility of related information	Is it possible to display all related information side by side? Yes/No
RI2		×		The worst-case number of steps required to navigate to the display of related information.
SRE1	×		Use of screen real estate	The maximum number of program elements that can be displayed on a physical screen.
SRE2	×			The number of non-semantic intersections on the physical screen present when obtaining the SRE1 score
AS1, AS2, AS3	×		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?'

instead of an approach to be used later in the lifecycle, because problems uncovered at design time are easier to correct than those uncovered after a prototype has been built. The quality of the problem discovery process can also be greater if done at design time. For example, Winograd points out that in studying usability, a user is more likely to provide substantive suggestions for a rough design than a polished prototype [23]. This observation was borne out in our experiences, a point upon which we will elaborate later in this paper.

Using the benchmarks is a three-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, i.e. 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. The Benchmarks in Detail

In discussing the benchmarks, we will show how they can be applied to the emerging designs of navigable static representations for Forms/3 [4] and ICBE [6]. Forms/3 is a declarative, form-based VPL that aims to achieve the power of traditional programming languages while maintaining the simplicity of the spreadsheet model. 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.

4.1. The Understandability Benchmarks

This section describes benchmarks for elements that relate to understandability of a program's representation. Forms/3 will be used to provide examples of how designers can use the benchmarks in this section. Programs in Forms/3 are defined via cells and formulas on forms. Each cell has a formula, which defines its value. Figure 1 and its detailed caption demonstrate the basic ideas of Forms/3. (A complete description of the language is given by Burnett and Ambler [4]).

Because representation design benchmarks are intended to help in the process of design, the Forms/3 benchmark examples are presented from the perspective of Forms/3 designers during the design of an improved navigable static representation. We will designate the representation used in the current implementation of Forms/3 as Design 1 and the new design that we are creating with the help of the representation design benchmarks as Design 2. All the Design 1 figures are screen shots from the current implementation and all the figures of Design 2 as it emerges through use of the benchmarks are, of course, hand-constructed sketches.

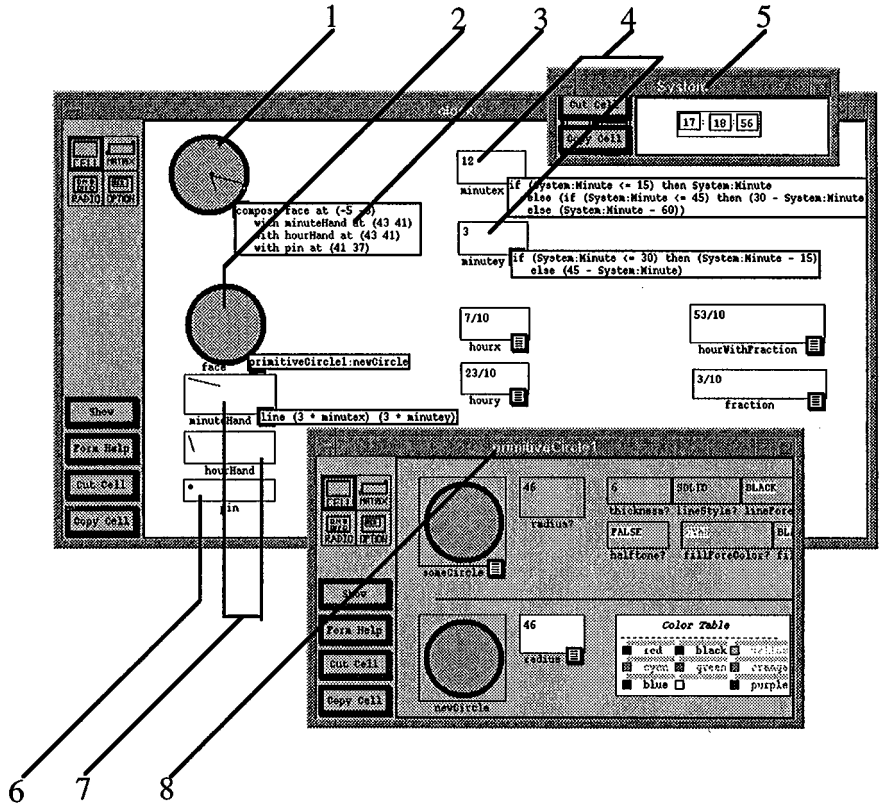

Figure 1. This Forms/3 program maintains a running analog clock (1) such as those commonly found ticking away on graphical workstations. The program requires only 11 new cells (which are on the form entitled 'clock'), some of which access built-in cells representing the internal clock and graphics support. To program the face (2), the programmer fills out a built-in circle specification form (8) and refers to it in the face formula. The pin (6) is specified the same way. The clock can be viewed as a local coordinate system with the pin at the origin, divided into four quadrants. Thus, the minute hand (7) is simply a line drawn from (0, 0) to the *x*- and *y*-positions (4) in the appropriate quadrant for the internal clock's time in minutes (5). The hour hand is calculated similarly. The clock's formula (3) was generated by using direct manipulation to demonstrate the arrangement of the face (2), hands (7) and pin (6) and taking a snapshot; the face, hands and pin cells were later moved apart for clarity (see color plate 1)

4.1.1. 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 [8].

Benchmarks D1 and D2 are based upon this CD. D1 is an $\mathcal{S}_p$ benchmark that measures whether the dependencies are explicitly depicted in the representation and D2

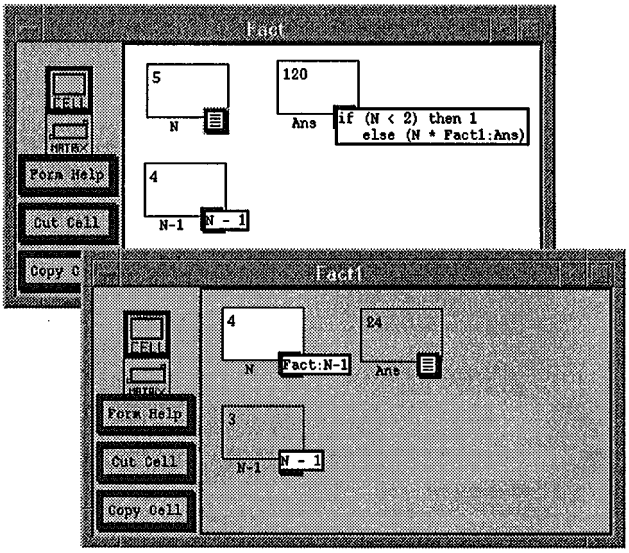

Figure 2. Forms/3 Design 1 screen snapshot: Program to compute the factorial function with selected formulas shown. Instances (gray shaded) inherit their model's cells and formulas unless the programmer explicitly provides a different formula for a cell on an instance, in which case the cell background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

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 bidirectional source of dependency is actually two, unidirectional 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 unidirectional dependency sources that are explicitly represented by the total number of unidirectional 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.

Detailed Example: Using the Dependencies Benchmarks as a Design Aid in Forms/3. Figure 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 form Fact is termed the *model* and Fact1, which was copied from Fact and then modified, is an *instance* of Fact.

The benchmarks were performed on Forms/3's representation Design 1 and Design 2 by members of the Forms/3 design team. There are two bidirectional 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 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 instances—except for formulas that the programmer has explicitly changed on the instance—this is an important dependency in Forms/3.) Multiplying these two bidirectional sources by two gives four unidirectional 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 quest ion '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 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. The Forms/3 design team was 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. The Forms/3 designers used the rating scale in Appendix B. They interpreted both D1 and D2 to be roughly 'fair' according to the scale.

For Design 2, the Forms/3 design team 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 3. The programmer

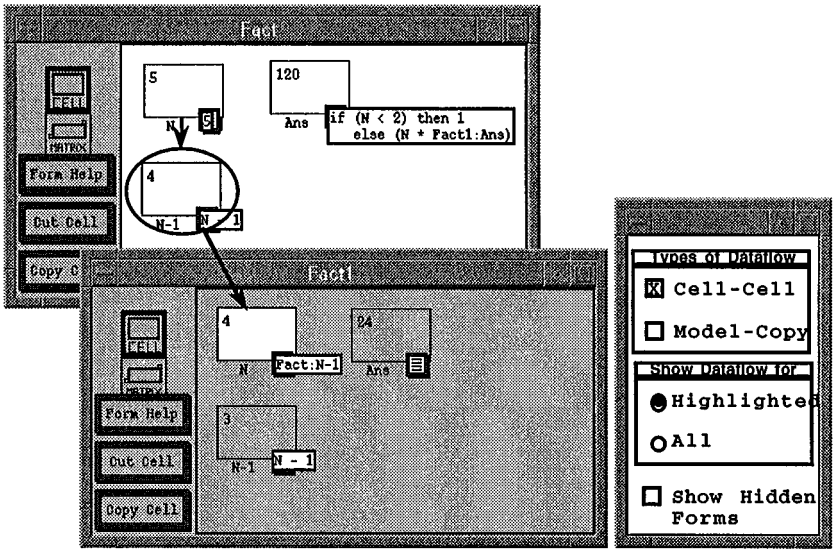

Figure 3. The design changes represented by Forms/3's Design 2 (shown via hand-drawn additions to the current implementation). Dataflow lines are superimposed on the cells. The rightmost window is the control panel. The programmer can select more than one cell at a time, but in this example, only cell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

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, which will be discussed in Section 4.2.

4.1.2. 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.

Returning to the Forms/3 example, 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.

The Forms/3 design team decided to add an optional view of the hierarchical dataflow between forms (Figure 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 [24]. The design team 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).

4.1.3. 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 the 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

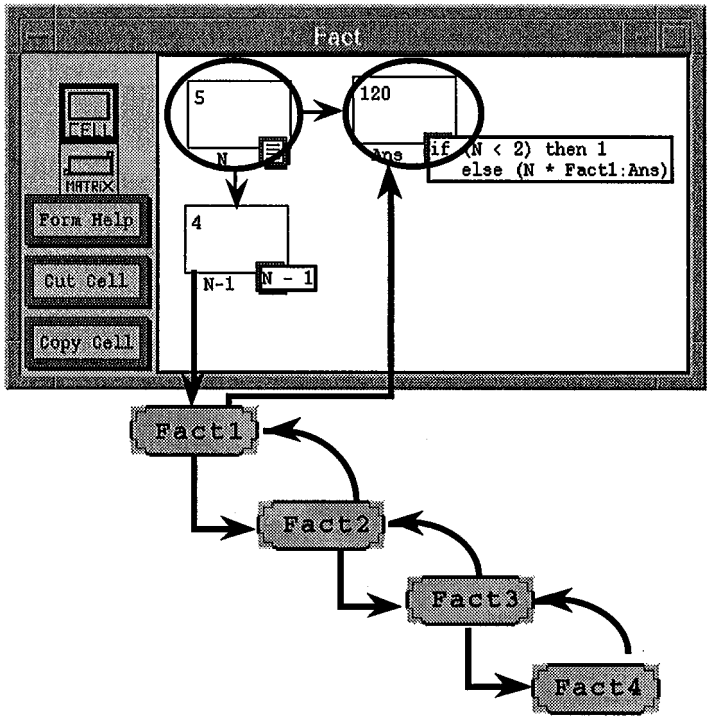

Figure 4. Forms/3 Design 2's program structure view of the factorial function. The source and sink of the dataflow are circled. Those forms that do not contain sources or sinks are shown as collapsed icons

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 [25].

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.

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.

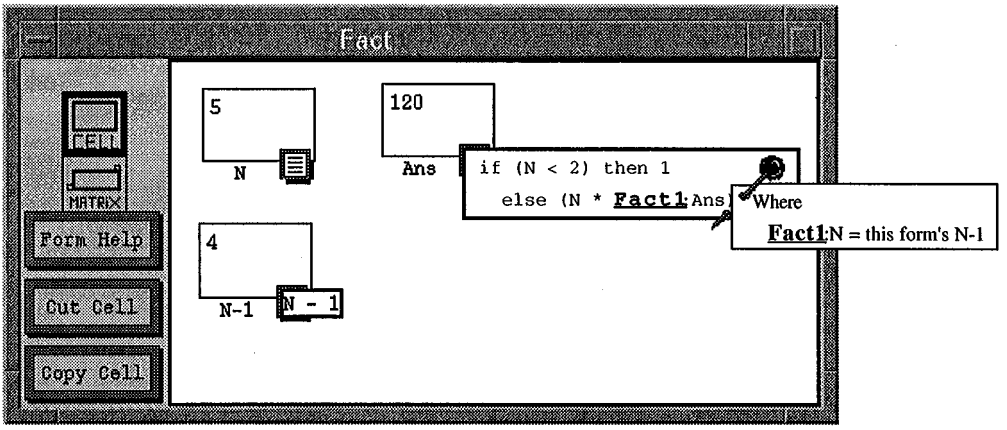

Figure 5. Forms/3 Design 2: The factorial function with legend. The bold and underlined form name *Fact1* indicates that the concrete form name is just an example of a more general relationship. Clicking on this name causes a legend to be attached to the formula display explaining the generalized relationship between this form and the form represented by Fact1

The Forms/3 design team 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. Forms/3's Design 1 contains one such source of misrepresentation, namely the use of concrete examples to identify form instances. 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 instance Fact1 (which computes 4 factorial); however, in actuality the formula refers to a generic instance of Fact whose computations are defined relative to the value of the $N-1$ cell on the referring form. In Design 2, the Forms/3 design team added a legend to provide complete information about the general relationship, as shown in Figure 5. Programmers can include or exclude such legends from the representation as desired.

4.1.4. 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 [8] 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. 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. The Forms/3 design team considered these Design 1 scores to be excellent, and made no changes in Design 2.

4.1.5. 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 instance 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 [26]. 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 instances 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 animations 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[a].

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.

[a] 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.

4.2. Scalability Benchmarks

In Burnett et al. [3], a VPL's navigable static representation is counted as an important aspect in the language's overall scalability. By measuring the 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.

4.2.1. 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 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.)[b]. Thus, to calculate the benchmark AG1, the designer divides the sources of detail that can be abstracted away in S by four.

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).

[b] 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.

4.2.2. 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 [8]. 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.

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. The Forms/3 design team 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 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.

4.2.3. 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,

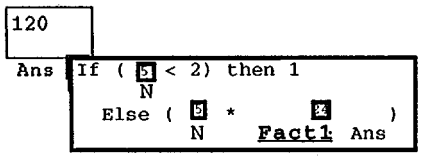

Figure 6. Forms/3 Design 2: The values are displayed with the cells reference in the formula. This eliminates the need for a programmer to search for these cells to find out their current values contributing to the value of Ans

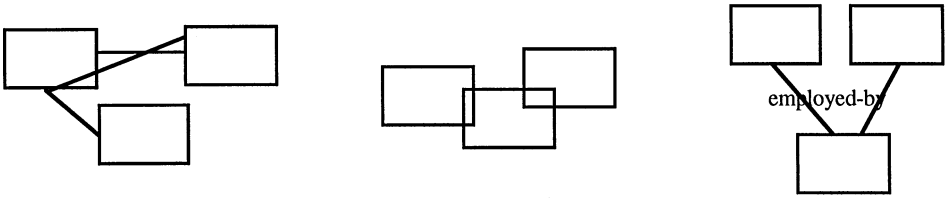

Figure 7. Non-semantic intersection examples that might be found in a VPL. (Left): Line crossings. (Middle): Unrelated boxes overlapping, seeming to imply a logical grouping. (Right): A line's label overlaps an unrelated line

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. Purchase and others pointed out the problem with representation of graphs with line crossings [27]. 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 7.

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.

Returning to the Forms/3 example, the program elements are the cells. In performing SRE1 and SRE2 for Design 1, the Forms/3 design team 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 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. The Forms/3 design team 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.

5. Benchmarks for Audience-Specific VPLs

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 section 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 [28]. 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

Table 2. 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, the Forms/3 design team 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

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.
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

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?

5.1. How to Apply the Audience-Specific Benchmarks

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.

5.2. Detailed Example: Using the Audience-Specific Benchmarks as a Design Aid in ICBE

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 high-level, set-oriented dataflow[c] VPL for users who are comfortable with computers but have no formal training in programming. 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 8 for an example. ICBE is a generalization of the kind of

[c] The term 'set-oriented dataflow' is meant to describe the fact that sets, rather than atomic values, flow along dataflow paths.

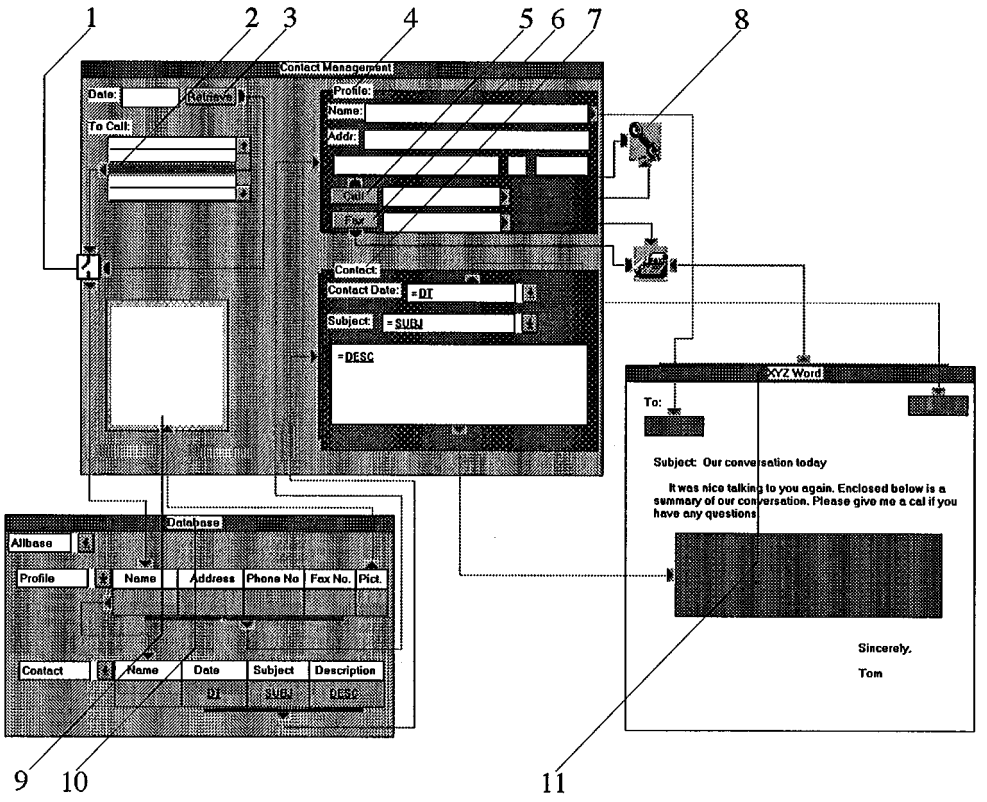

Figure 8. A salesperson is creating a program for a contact management application in ICBE. To make a call, the salesperson will highlight a customer (2) in the 'To Call' list and press the 'Retrieve' button (3). This will close the gate (1) and thereby complete the circuit, allowing the highlighted 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) (see color plate 2)

declarative by-example programming used in QBE and OBE [29, 30]; a more complete description of ICBE can be found in Zloof and Krishnamurthy and Krishnamurthy and Zloof [5, 6].

5.2.1. 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 email. (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

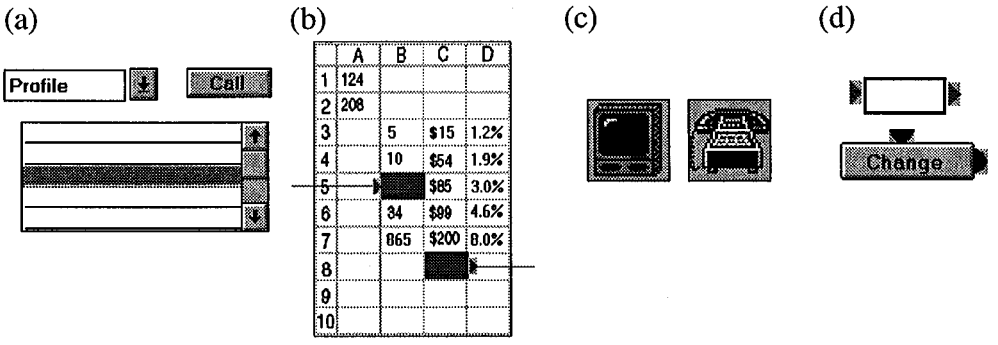

Figure 9. (a) Some ICBE user interface primitives. (b) A grid represents a spreadsheet, which is an example of an interoperation object. (c) Television and telephone plug-in objects. (d) Arrows represent ports: the red (pointed) arrows are dataflow ports and the blue (rounded) arrows are control flow ports

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

5.2.2. 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 9. 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.

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

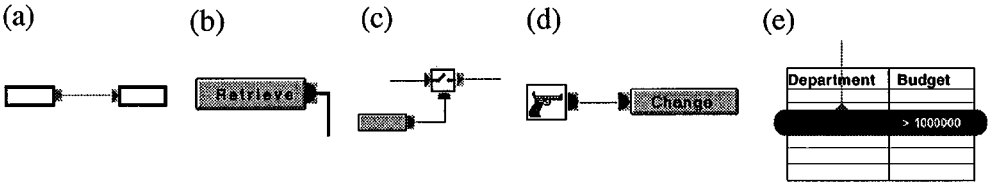

Figure 10. Some ICBE operations. (a) Dataflow. (b) Event-based control flow, initiated by pressing the Retrieve button. Control flow for transfer of control is also shown via these blue lines. (c) An open gate interrupts dataflow. (d) A trigger causes the change button to be 'pushed' automatically. (e) A query slider is a data selection operator

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.

5.2.3. 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 10. 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.

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[d]. 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 $3/6 = 0.50$.

5.2.4. 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 8 shows one example of layout with several examples of line connections and Figure 11 shows a container nested within another container.

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

[d] However, the designers noted that denoting the *difference* between water/data lines and electrical/control lines by using the colors red and blue does not map to any generally accepted convention.

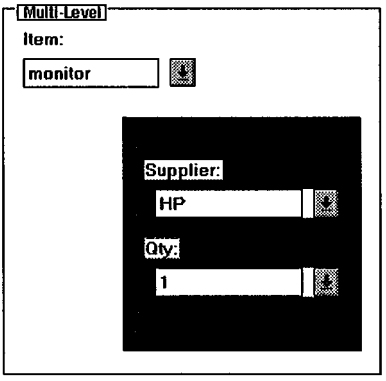

Figure 11. ICBE containers. The blue inner container combines a supplier list, list of possible quantities and textual labels for each. The outer container in turn combines the inner container with an item list and textual label. The nesting implies a constrained relationship; for example, if the value 'monitor' were 'keyboard' instead, the contents of the nested container would reflect the supplier and quantity of the keyboard order

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.

5.2.5. Outcomes of the Benchmarks for ICBE

The ICBE designers found that using the representation design benchmarks—both the audience-specific benchmarks described in this section and the rest of the benchmarks described in Section 4—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.

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.

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 taught by one of the authors (Burnett). Recent studies of usability testing show that five test participants are sufficient to uncover approximately 80% of the usability problems [31]. (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 the Forms/3 and ICBE design teams 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[e].

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.

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 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 animations, 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

[e] See, for example, the study of the VPL LabView (5 subjects) [32], the study of the VPL ChemTrains (6 subjects) [15], and the study of a generic Petri-net language (12 subjects) [33].

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.

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 re-measure. (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.

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 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.

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 3.

7. 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

Table 3. Problems found and corrections made that resulted from using the representation design benchmarks, as reported by the subjects

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.

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, 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.

8. Conclusions

VPL researchers are continually making advances in devising new ways to create programs—from the kinds of programs that professional programmers create to the kinds of component interweaving and macro-building that end-users do. To provide these advances, many VPLs make innovative use of the capabilities of today's graphical workstations, including graphics, color, and animation. But most of the design advances have been related to VPLs' dynamic aspects, and as a result, the static representations of many VPLs have been their Achilles heel. Unfortunately, this deficiency can seriously handicap VPLs' suitability for certain tasks that arise in programming, such as working with and understanding a program written by someone else.

Representation design benchmarks are the first approach devised specifically to help VPL designers address this deficiency. Extending the work on cognitive dimensions for programming systems, the benchmarks allow a designer to see how a navigable static representation design's features impact the accessibility and usefulness of the information available about a program. Representation design benchmarks have been used both by experienced VPL designers in designing navigable 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 make a measurable difference in helping VPL de signers discover problems with their designs.

The key characteristics of representation design benchmarks are that they provide a concrete way for VPL designers to apply HCI principles on cognitive aspects of programming, they are a set of measurement procedures rather than a set of guidelines, and they focus directly and exclusively on VPLs' navigable static representations. Through these characteristics, they provide a practical means to measure a VPL's navigable static representation in isolation from the other, more dynamic aspects of the VPL, helping the designer obtain a clear view of a proposed navigable static representation's strengths and weaknesses. In this way, representation design benchmarks can help VPL designers combine the flexibility and responsiveness that can be supported by a VPL's dynamic aspects, with the easy access to large amounts of program information that ensues from a well-designed navigable static representation.

Acknowledgments

We would like to thank Judith Hays and Eric Wilcox for assistance with this paper, and John Atwood, Baljinder Ghotra, Herkimer Gottfried, Shikha Gottfried, Dianne

Hackborn, Luca Tallini, Rebecca Walpole, and the members of the Forms/3 and the ICBE design teams for their help in the research that led to this paper. We especially thank Thomas Green for his helpful comments.

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 [8], 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'.

Table A1. The Cognitive Dimensions (extracted from Green and Petre [8])

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?

"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 [8])

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 B1, 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. 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

Benchmark	S_c	S_p	NI	Aspect of the representation	Example rating scale
D1		×		Visibility of dependencies	Ratio = Poor Fair Good (0.0 0.5 1.0)
D2			×		# steps = Poor Fair Good (>n n 0)
PS2			×	Visibility of program structure	# steps = Poor Fair Good (>n n 0)
L2			×	Visibility of program logic	# steps = Poor Fair Good (>n n 0)
L3	×				# steps = Poor Good (>n 0)
R2			×	Display of results with program logic	# steps = Poor Fair Good (>n n 0)

Table B1. Continued

Benchmark	S_c	S_p	NI	Aspect of the representation	Example rating scale
SN1		×		Secondary notation: non-semantic devices	Ratio = Poor Fair Good (0.0 0.5 1.0)
SN2			×		# steps = Poor Fair Good (>n n 0)
AG1		×		Abstraction gradient	Ratio = Poor Fair Good (0.0 0.5 1.0)
AG2			×		# steps = Poor Fair Good (>n n 0)
RI2			×	Accessibility of related information	# steps = Poor Fair Good (>n n 0)

References

1. B. Myers (1990) Taxonomies of visual programming and program visualization. *Journal of Visual Languages and Computing* **1,** 97–123.
2. A. Cypher, D. Kosbie & D. Maulsby (1993) Characterizing PBD systems. In: *Watch What I Do: Programming by Demonstration* (A. Cypher, ed.). MIT Press, Cambridge, MA.
3. M. Burnett, M. Baker, C. Bohus, P. Carlson, S. Yang & P. van Zee (1995) Scaling up visual programming languages. *IEEE Computer* **28,** 45–54.
4. M. Burnett & A. Ambler (1994) Interactive visual data abstraction in a declarative visual programming language. *Journal of Visual Languages and Computing* **5,** 29–60.
5. M. Zloof & R. Krishnamurthy (1994) IC by example: empowering the uninitiated to construct database applications. Technical Report, Hewlett-Packard Laboratories.
6. R. Krishnamurthy & M. Zloof (1995) RBE: Rendering by ex ample. In: *Eleventh International Conference on Data Engineering*, Taipei, Taiwan, pp. 288–297.
7. T. Green (1991) Describing information artifacts with cognitive dimensions and structure maps. In: *People and Computers* VI (D. Diaper & N. Hammond, eds). Cambridge University Press, Cambridge.
8. T. R. G. Green and M. Petre (1996) Usability analysis of visual programming environments: a 'cognitive dimensions' framework. *Journal of Visual Languages and Computing* **7,** 131–174.
9. P. T. Cox, F. R. Giles & T. Pietrzykowski (1989) Prograph: a step towards liberating programming from textual conditioning. In: *1989 IEEE Workshop on Visual Languages*, Rome, Italy, pp. 150–156.
10. J. Kodosky, J. MacCrisken & G. Rymar (1991) Visual programming using structured data flow. In: *1991 IEEE Workshop on Visual Languages*, Kobe, Japan, pp. 34-39.
11. F. Modugno, T. Green B. Myers (1994) Visual programming in a visual domain: a case study of cognitive dimensions. In: *People and Computers* IX (G. Cockton, S. Draper & G. Weir, eds). Cambridge University Press, Cambridge, UK.
12. M. Yazdani & L. Ford (1996) Reducing the cognitive requirements of visual programming. In: *1996 IEEE Symposium on Visual Languages*, Boulder, CO, pp. 255–262.
13. D. Hendry (1995) Display-based problems in spreadsheets: a critical incident and a design remedy. In: *1995 IEEE Symposium on Visual Languages*, Darmstadt, Germany, pp 284–290.
14. M. Bell (1994) Evaluation of visual programming languages and environments. Technical Report, CTI Centre for Chemistry, University of Liverpool.

15. B. Bell, J. Rieman & C. Lewis (1991) Usability testing of a graphical programming system: things we missed in a programming walkthrough. In: *ACM SIGCHI 1991*. ACM Press, New Orleans, pp. 7–12.
16. B. Bell, W. Citrin, C. Lewis, J. Rieman, R. Weaver, N. Wilde & B. Zorn (1994) Using the programming walkthrough to aid in programming language design. *Software Practice and Experience* **24,** 1–25.
17. E. Glinert (1989) Towards a software metrics for visual programming. *International Journal of Man–Machine Studies* **30,** 425–445.
18. S. Card, T. Moran & A. Newell (1983) *The Psychology of Human–Computer Interaction*. Erlbaum, Hillsdale, NJ.
19. A. Siochi & D. Hix (1991) A study of computer-supported user interface evaluation using maximal repeating pattern analysis. In: *ACM SIGCHI 1991*, New Orleans, LA, pp. 301– 305.
20. J. Nielsen & R. Molich (1990) Heuristic evaluation of user interfaces. In: *ACM SIGCHI 1990*, Seattle, Washington, pp. 249– 256.
21. J. Nielsen (1992) Finding usability problems through heuristic evaluation. In: *ACM SIGCHI 1992*, pp. 373–380.
22. A. Sears (1993) Layout appropriateness: a metric for evaluating user interface widget layout. *IEEE Transactions on Software Engineering* **19,** pp. 707–719.
23. T. Winograd (1995) From programming environments to environments for designing. *Communications of the ACM* **38,** 65–74.
24. S. Yang & M. Burnett (1994) From concrete forms to generalized abstractions through perspective-oriented analysis of logical relationships. In: *1994 IEEE Symposium on Visual Languages*, St. Louis, MO, pp. 6–14.
25. P. Brown & J. Gould (1987) An experimental study of people creating spreadsheets. *ACM Transactions on Office Information Systems* **5,** 258–272.
26. M. Petre (1995) Why looking isn't always seeing: readership skills and graphical programming. *Communications of the ACM* **38,** 33–44.
27. H. C. Purchase, R. F. & Cohen M. James (1995) Validating Graph Drawing Aesthetics. In: *Lecture Notes in Computer Science* (F. Brandenburg, ed.). Springer, Berlin, 1995 .
28. B. Nardi (1993) *A Small Matter of Programming: Perspectives on End User Computing*. MIT Press, Cambridge, MA.
29. M. Zloof (1977) Query by example: a data base language. *IBM Systems Journal* **16,** 324–343.
30. M. Zloof (1981) QBE/OBE: a language for office and business automation. *Computer* **14,** 13–22.
31. R. Virzi (1992) Refining the test phase of usability evaluation: How many subjects is enough? *Human Factors* **34,** 457–468.
32. T. Green, M. Petre & R. Bellamy (1991) Comprehensibility of visual and textual programs: a test of superlativism against the 'match–mismatch' conjecture. In: *Empirical Studies of Programmers: Fourth Workshop* (J. Koenemann-Belliveau, T. Moher & S. Robertson, eds). Ablex Publishing, Norwood, NJ.
33. T. Moher, D. Mak, B. Blumenthal & L. Leventhal (1993) Comparing the comprehensibility of textual and graphical programs: the case of Petri Nets. In: *Proc. Empirical Studies of Programmers: Fifth Workshop*. Palo Alto, CA, Ablex Publishing, Norwood, NJ.