Graphical Definitions: Expanding Spreadsheet Languages through Direct Manipulation and Gestures

MARGARET M. BURNETT
Oregon State University
and
HERKIMER J. GOTTFRIED
Hewlett-Packard Company

In the past, attempts to extend the spreadsheet paradigm to support graphical objects, such as colored circles or user-defined graphical types, have led to approaches featuring *either* a direct way of creating objects graphically *or* strong compatibility with the spreadsheet paradigm, but not both. This inability to conveniently go beyond numbers and strings without straying outside the spreadsheet paradigm has been a limiting factor in the applicability of spreadsheet languages. In this article we present graphical definitions, an approach that removes this limitation, allowing both simple and complex graphical objects to be programmed directly using direct manipulation and gestures, in a manner that fits seamlessly within the spreadsheet paradigm. We also describe an empirical study, in which subjects programmed such objects faster and with fewer errors using this approach than when using a traditional approach to formula specification. Because the approach is expressive enough to be used with both built-in and user-defined types, it allows the directness of demonstrational and spreadsheet techniques to be used in programming a wider range of applications than has been possible before.

Categories and Subject Descriptors: D.1.1 [**Programming Techniques**]: Applicative (Functional) Programming; D.1.7 [**Programming Techniques**]: Visual Programming; D.3.3 [**Programming Languages**]: Language Constructs and Features—*abstract data types*; *data types and structures*; H.4.1 [**Information Systems Applications**]: Office Automation—*spreadsheets*

General Terms: Design, Human Factors, Languages

Additional Key Words and Phrases: Direct manipulation, Forms/3, Gestures, Programming by demonstration

This work was supported by Hewlett-Packard, by Harlequin, and by the National Science Foundation under grants CCR-9308649, ASC-9523629, and an NSF Young Investigator Award.
Authors' addresses: M. M. Burnett, Department of Computer Science, Oregon State University, Corvallis, OR 97331; email: burnett@cs.orst.edu; H. J. Gottfried, Hewlett-Packard Company, 1000 NE Circle Blvd., Corvallis, OR 97330; herkyg@cv.hp.com.

1. INTRODUCTION

In recent years, many new graphical techniques have been developed to support programming with graphical objects. Of particular note are the contributions of demonstrational programming research, which have brought straightforward, graphical techniques for creating and working with graphical objects to both end-users and programmers. Unfortunately, however, users of spreadsheets have been left out of these advances and still find themselves in a highly textual world with limited abilities to incorporate graphical objects into their computations.

This article presents a solution to this problem. Our goal was to incorporate graphical objects into spreadsheets in a way that would fit seamlessly within the spreadsheet paradigm. Further, we wanted our approach, like most other features found in spreadsheets, to be applicable to *all* users of spreadsheet languages. That is, we wanted to support the simple, built-in graphical objects likely to be used by ordinary end-users, in a way expressive enough to also support the complex, user-defined objects needed by programmers.

The approach described in this article has these attributes. It supports both simple and complex objects in a spreadsheet language via direct manipulation and gestures. We call these direct manipulations and gestures *graphical definitions*, to emphasize that they are a declarative way to *define formulas* for cells in a *graphical* manner. The primary new contributions of the approach are that

—it supports a wide range of applications, from end-user-oriented programmable graphics to programmer-oriented data structures;

—it supports not only built-in graphical types, but also user-defined types; and

—it fully supports working directly and graphically with objects in a way that fits completely within the spreadsheet paradigm of cells and formulas, without employing state-modifying sublanguages, macros, or trapdoors to other programming languages.

1.1 Organization of this Article

We begin with a discussion of the design goals of our approach. In Section 2, we review related work, evaluating how other systems relate to these design goals. In Section 3, we present the approach informally via two examples, showing how our technique might be used both by end-users and by programmers. In Section 4 we formally present the semantics of graphical definitions. After a discussion of interactions between the approach and the spreadsheet paradigm in Section 5, the results of an empirical study are presented in Section 6, followed by the conclusion in Section 7.

1.2 Design Goals

We use the term *spreadsheet languages*[1] to refer to all systems that follow the spreadsheet paradigm, from commercial spreadsheets to more sophisticated systems whose computations are defined by one-way constraints in the cells' formulas. The essence of this paradigm is expressed well by Alan Kay's *value rule*, which states that a cell's value is defined solely by the formula explicitly given it by the user [Kay 1984]. By "fitting seamlessly within the spreadsheet paradigm," we mean that the approach follows the value rule. The characteristic of seamlessness within the spreadsheet paradigm was one of our two primary design goals.

Our other primary design goal was *directness*, a term we will use to mean following the principles advocated by Shneiderman, by Hutchins, Hollan, and Norman, by Green and Petre, and by Nardi. The term *direct manipulation* was coined by Shneiderman, who describes three principles of direct-manipulation systems: continuous representation of the objects of interest, physical actions or presses of labeled buttons instead of complex syntax, and rapid incremental reversible operations whose effect on the object of interest is immediately visible [Shneiderman 1983].

Hutchins, Hollan, and Norman expand upon these notions, suggesting that the degree to which a user interface feels direct is inversely proportional to the cognitive effort needed to use the interface [Hutchins et al. 1986]. They describe directness as having two aspects. The first aspect is the *distance* between one's goals and the actions required by the system to achieve those goals. In traditional spreadsheet programming, distance is fairly small because there is a well-understood, one-to-one mapping from each operator and term in the goal to the formula that must be specified (e.g., from the goal "add A and B" to the formula "A + B"). In contrast, Green and Petre enumerate several examples showing the unfortunate lack of this aspect of directness (termed "closeness of mapping" in their work) in commonly used programming languages [Green and Petre 1996]. The second aspect is a feeling of *direct engagement*, "the feeling that one is directly manipulating the objects of interest." Nardi sees direct engagement as a critical element in spreadsheets, emphasizing freedom from low-level programming minutiae in favor of task-specific operations [Nardi 1993]. Direct engagement has been largely absent from prior approaches to supporting graphics in spreadsheet languages.

2. RELATED WORK AND BACKGROUND

2.1 Related Work

Many commercial spreadsheets provide the capability to display simple graphics and charts. However, these graphical objects are strictly output

[1]We have chosen this terminology to emphasize the fact that even commercial spreadsheets are indeed languages for programming, although they differ in audience, application, and environment from traditional programming languages.

mechanisms: they cannot be values of cells; other cells' values cannot depend on them; and only the charts (not the other kinds of graphics) can be dependent on other cells. Furthermore, these spreadsheets do not allow users to extend the set of graphical objects that are supported. In some spreadsheets, it is possible to gain some graphical support for objects through the use of macro languages and incorporation of state-modifying programming languages, but these approaches violate the spreadsheet value rule. Macros violate it because a macro stored in one group of cells actually changes other cells' formulas during execution—the spreadsheet equivalent of self-modifying programs.

Although some research spreadsheet languages have used graphical techniques, they have not achieved the combination of generality and directness that we sought. One of the pioneering systems in this direction was NoPumpG [Lewis 1990] and its successor NoPumpII [Wilde and Lewis 1990], two spreadsheet languages designed to support interactive graphics. The design goal of these systems was to provide the capability to create low-level graphs while adding as little as possible to the basic spreadsheet paradigm. Thus, NoPumpG and NoPumpII include some built-in graphical types that may be instantiated using cells and formulas, and they support limited (built-in) manipulations for these objects, but do not support complex or user-defined objects.

Penguims [Hudson 1994] is an environment based on the spreadsheet model for specifying user interfaces. Its goal is to allow interactive user interfaces to be created with little or no explicit programming. This work is similar to ours in its support for abstraction—it provides the capability to collect cells together into *objects*—but unlike our work, it employs several techniques that do not conform to the spreadsheet value rule, such as interactor objects that can modify the formulas of other cells, and imperative code similar to macros.

Action Graphics [Hughes and Moshell 1990] is a spreadsheet language for graphics animations. It provides some support for complex objects, such as the ability to group cells into "composite cells," but does not provide the directness we sought. Also, animation in Action Graphics is performed through functions that cause side effects; thus, this approach violates the spreadsheet value rule. Smedley, Cox, and Byrne have incorporated the visual programming language Prograph and user interface objects into a conventional spreadsheet in order to provide spreadsheet users with a graphical interface for input and feedback [Smedley et al. 1996]. However, although the Prograph approach to spreadsheets adds the ability to incorporate graphical objects into spreadsheets, it does not make programming them more direct.

Wilde's WYSIWYC spreadsheet [Wilde 1993] aims to improve traditional spreadsheet programming by making cell formulas visible and by making the visible structure of the spreadsheet match its computational structure. Although this work is similar to ours in its attempt to emphasize the task-specific operations of spreadsheet languages, Wilde focuses on the visual representation of the resulting program rather than on the means of specifying it and does not address graphical types.

C32 [Myers 1991] is a spreadsheet language that uses graphical techniques along with inference to specify constraints in user interfaces. Unlike the other spreadsheet languages described here, C32 is not a full-fledged spreadsheet language; rather, it is a front-end to the underlying textual language Lisp used in the Garnet user interface development environment [Myers et al. 1990]. C32 is a way of viewing constraints, but does not itself feature the graphical creation and manipulation of graphical objects. Instead, this function is performed by the demonstrational system Lapidary [Vander Zanden and Myers 1995], which is another part of the Garnet package. The combination of C32 and Lapidary (and the other portions of the Garnet package) features strong support for direct manipulation of built-in graphical user interface (GUI) objects, but not for any other kinds of objects, which must be written and manipulated in Lisp.

Recent work on the spreadsheet language Formulate introduced the use of voice, handwriting, and gestures as input modalities for entering standard spreadsheet formulas [Leopold and Ambler 1997]. All three of these modalities can be mixed in the entry of a single formula. The Formulate work addresses a different problem than does our approach. The use of gestures (and the other modalities) in Formulate replaces one token or a small group of tokens in a single formula in order to enhance the convenience of formula entry, whereas our approach replaces collections of formulas in order to enhance the directness of the language syntax.

Our work is also related to research on demonstrational programming by direct manipulation of objects, such as in Chimera [Kurlander 1993], KidSim/Cocoa [Smith et al. 1994], Visual AgenTalk [Repenning and Ambach 1996], Mondrian [Lieberman 1993], TRIP3, [Miyashita et al. 1992], and IMAGE [Miyashita et al. 1994]. Of these, the most closely related to our work are those featuring a declarative approach, namely the rule-based and the constraint-based systems. KidSim and Visual AgenTalk are two rule-based systems from the demonstrational family, and they use direct manipulation to specify declarative graphical rewrite rules. Although the approaches used by these systems have some similarity to ours in their support for directness using a declarative mechanism, they do not provide the kind of flexible, declarative specification of objects and attributes that we sought for a full-featured, spreadsheet-based approach.

The multiway constraint-based systems TRIP3 [Miyashita et al. 1992] and IMAGE [Miyashita et al. 1994] also use direct manipulation as a means of specifying relations declaratively; in these systems a visual example defines a relationship between the application data and its visual representation. However, like many demonstrational systems, their approach uses inference to determine this relation rather than having the relation be specified explicitly by the programmer. Although our system shares with inferential languages the property that concrete examples are used in programming, our approach avoids using inference to derive program logic. Also, the purpose of TRIP3 and IMAGE is to provide a *visual* interface to traditional *textual* programming languages, while our approach attempts to extend the power of the spreadsheet without involving any other programming language. Another kind of multiway constraint system that uses direct manipulation as a means of specifying relations among visual objects is EUPHORIA [McCartney et al. 1995]. With EUPHORIA, end-users can specify user interfaces by direct manipulation of both GUI objects and aggregates of these objects, but EUPHORIA does not support more general types of objects or more general types of operations.

Because of the multiway nature of these constraint-oriented systems, they would not fit seamlessly within the spreadsheet paradigm: multiway constraints violate the spreadsheet value rule. To see why, imagine specifying the formula for cell X to be a box whose width is a reference to cell W (whose formula is cell A plus cell B). If the user then selects and stretches the box in X, what does that mean for cells W, A, and B? If any of these are automatically changed, the value rule is violated for the changed cell(s); if they are not changed, the multiway nature of the constraints is not being maintained. Multiway constraints also add a new concept to spreadsheet languages, which was another reason we wanted to explore whether we could develop a declarative approach that would work with the *one*-way constraints used in spreadsheet languages.

2.2 Background: Forms/3

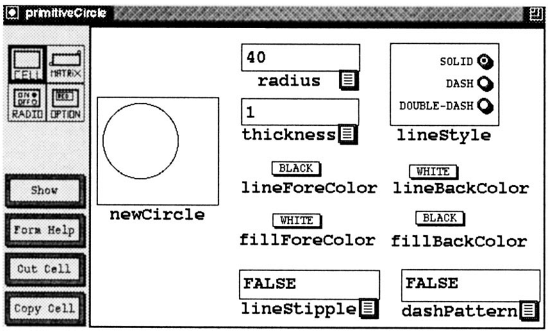
We have prototyped our approach in the spreadsheet language Forms/3 [Atwood et al. 1996; Burnett and Ambler 1994], and the examples in this article are presented in that language. Forms/3 has long supported both built-in graphical types and user-defined graphical types.[2] (Built-in types are provided in the language implementation but are otherwise identical to user-defined types.) Attributes of a type are defined by formulas in groups of cells, and an instance of a type is the value of an ordinary cell that can be referenced just like any other cell. For example, the built-in circle object shown in Figure 1 is defined by cells defining its radius, line thickness, color, and other attributes. Multiple circles can be defined by making copies of the circle form, changing formulas as needed on the copies to specify the desired attributes. In this article, we extend this approach with graphical definitions to support a more direct style of specifying graphical objects.

3. PROGRAMMING GRAPHICAL OBJECTS DIRECTLY

3.1 Example 1: An End-User's Use of Graphical Definitions

We have stated that we wanted to devise an approach appropriate for a wide range of users who use spreadsheet languages, from end-users to professional programmers. This section relates to the end-user side of that

[2]Besides its support for graphical types, Forms/3 has some other noticeable differences from commercial spreadsheets. For example, Forms/3 cells need not be placed in a grid of rows and columns (as shown in Figure 1), although they can be (as will be shown by Figure 2). Its formula language is fairly conventional, however. See Table I.



radius	40
thickness	1
lineStyle	SOLID / DASH / DOUBLE-DASH
lineForeColor	BLACK
lineBackColor	WHITE
fillForeColor	WHITE
fillBackColor	BLACK
lineStipple	FALSE
dashPattern	FALSE

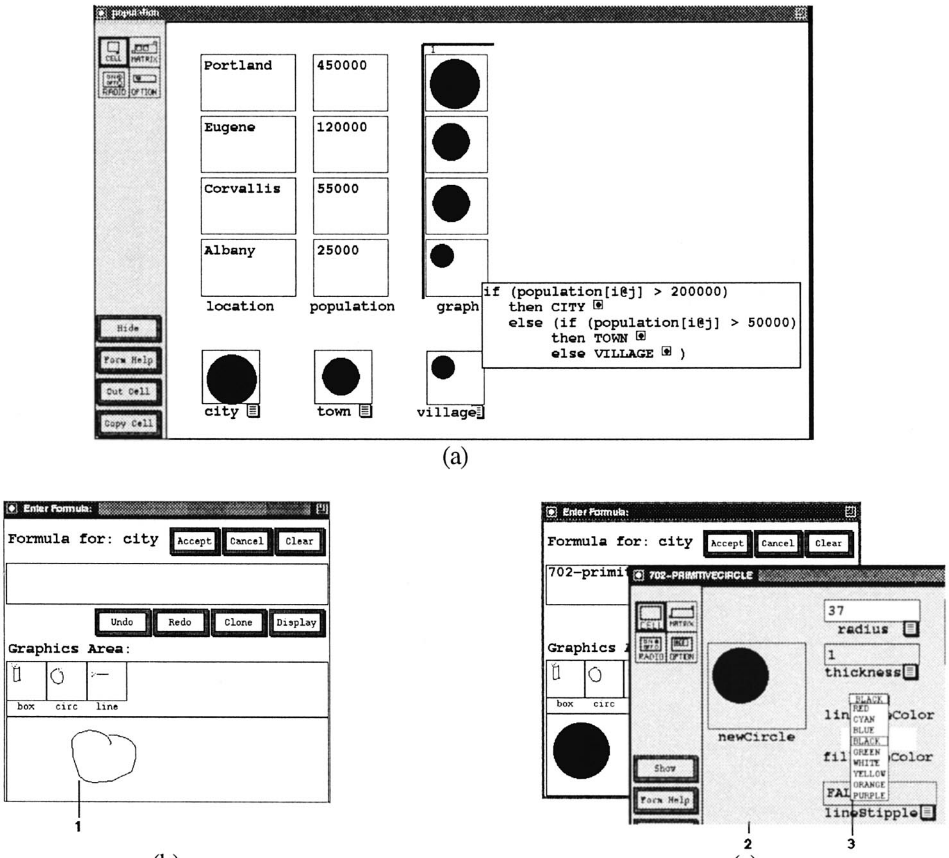
Fig. 1. A portion of a Forms/3 form that defines a circle. The circle in cell *newCircle* is specified by the other cells, which define its attributes. A user can view and specify spreadsheet formulas by clicking on the formula tabs (the rectangular icon next to the cell names). Radio buttons and popup menus are the equivalent of cells with simple formulas.

Table I. The Grammar for the Subset of Forms/3 Formula Language Used in this Article

formula	::=	*Blank* | expr
expr	::=	*Constant* | ref | infixExpr | prefixExpr | ifExpr | composeExpr
infixExpr	::=	subExpr infixOperator subExpr
prefixExpr	::=	unaryPrefixOperator subExpr | binaryPrefixOperator subExpr subExpr
ifExpr	::=	IF subExpr THEN subExpr ELSE subExpr
composeExpr	::=	COMPOSE subExpr WITH subexpr AT (subexpr subexpr) | COMPOSE subExpr AT (subexpr subexpr)
subExpr	::=	*Constant* | ref | (expr)
infixOperator	::=	+ | − | * | / | AND | OR | = | > | < | . . .
unaryPrefixOperator	::=	ROUND | WIDTH | − | . . .
binaryPrefixOperator	::=	APPEND | . . .
ref	::=	cellRef | *Form* : cellRef
cellRef	::=	*Cell* | *Matrix* | *Abs* | *Abs* [*Cell*] | *Matrix* [subscripts] | *Abs* [*Matrix*] | *Abs* [*Matrix*] [subscripts]
subscripts	::=	matrixSubscript@matrixSubscript
matrixSubscript	::=	expr

As the top section shows, Forms/3 has the usual formula operators and some operators supporting computations on graphics. The bottom four lines show cell reference syntax, which includes cell groups (*Abs*traction boxes, which will be discussed in Section 4), and row/column referencing for cells that are in a grid (matrix). There are also 6 "pseudoreferences"—I, J, NUMROWS, NUMCOLS, LASTROW, and LASTCOL—that are used in matrix subscripts. Including these in the grammar is straightforward but tedious, and we have omitted them for brevity.

range, considering a task that a spreadsheet end-user might be interested in performing that is beyond the capabilities of current spreadsheets. One such task might be displaying a visual representation of data, using domain-specific visualization rules. Figure 2(a) shows such a visualization that a population analyst might wish to specify in a spreadsheet language.



location	population
Portland	450000
Eugene	120000
Corvallis	55000
Albany	25000

```
if (population[i@j] > 200000)
   then CITY
   else (if (population[i@j] > 50000)
         then TOWN
         else VILLAGE )
```

Formula for: city

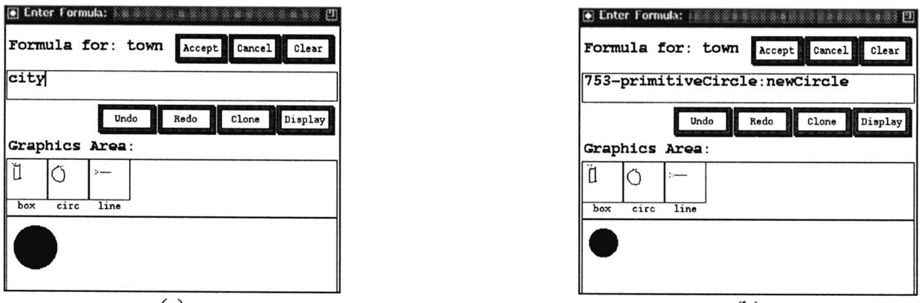
Formula for: city
702–primi

37
radius
1
thickness

Fig. 2. (a) A visualization of population data. The formula shown is shared by the 4 × 1 matrix labeled *graph*. The tiny icons inside the formula are miniaturized drawings of the cells' current values. (b) To define the circle for cell *city*, the population analyst first draws a *circle* gesture in *city*'s formula edit window (1) and then (c) after clicking on the resulting circle to display its definition form (2) (in gray because it is a copy; white indicates formulas different from the original), the population analyst specifies the *fillForeColor* formula via a popup menu (3). Each manipulation is immediately reflected textually and graphically in *city*'s formula edit window (shown behind the gray form in (c)).

The program categorizes population data into cities, towns, and villages, and represents each with a differently sized black circle.

The population analyst starts by drawing a circle-shaped gesture to define the large *city* circle (Figure 2(b)), resizing the resulting circle if necessary by direct manipulation. This defines the cell's formula to be a reference to cell *newCircle* on a copy of the built-in circle definition form whose radius formula is defined to be the radius of the drawn circle gesture. However, the analyst wants the circles in the program to be solid black. Because there are no gestures to specify fill color, the population analyst clicks on the circle to display its definition form and then defines the formula for cell *fillForeColor* (Figure 2(c)).


Formula for: town

city

Formula for: town

753-primitiveCircle:newCircle

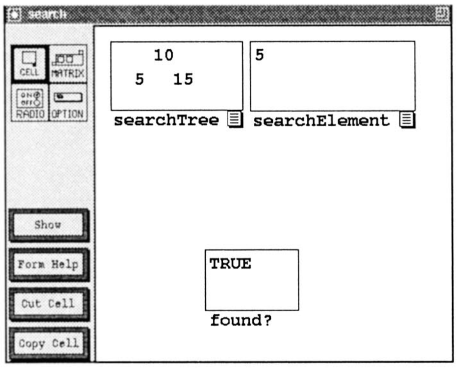
Fig. 3. In defining the circle for cell *town*, (a) the population analyst first opens *town*'s formula edit window and clicks on *city* to define *town*'s formula to be a reference to *city*. This formula is immediately reflected textually and graphically. (b) The population analyst then modifies *town*'s formula by shrinking the circle via direct manipulation. Thus *town*'s formula no longer refers to *city*, but to a similar but smaller circle instead. This formula change is immediately reflected textually and graphically.

An alternative technique for specifying cell *city* would have been to click on the circle icon in Figure 2(b). This icon shows that sketching a circle is one of the gestures that is applicable in this context. (How context works will be discussed in detail in Section 5.2). Clicking on a gesture icon produces a "representative" value (here, a circle with radius 25), which can then be resized via direct manipulation. The icon-clicking technique has the disadvantage of having fewer degrees of freedom than an actual gesture, so it requires more manipulations by the user to communicate equivalent information as a gesture if the gesture's size and/or orientation are important, but has the advantage of ensuring perfect recognition of the intended object. In the empirical study presented later in this article, we found that many subjects preferred to use a mixture of these two graphical techniques, and others opted to use only the icon-clicking technique; only one subject preferred to use gestures without sometimes using icon-clicking.

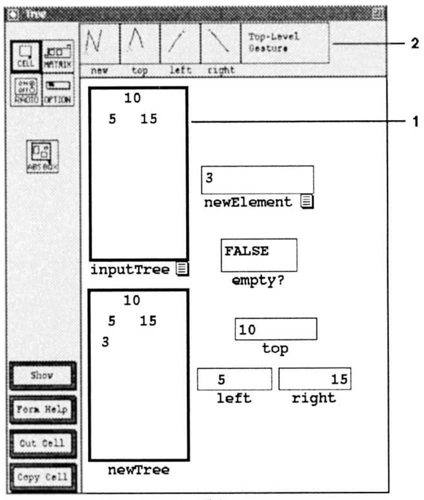
Cells *town* and *village* could be specified in the same ways as above, or alternatively by showing how they are different from the *city* circle. For example, to define the *town* circle, the population analyst clicks on cell *city* instead of drawing a new circle. This displays the circle in the formula edit window so that it can be manipulated (Figure 3). The population analyst then resizes the circle in the formula window to define the *town* circle, which has all of the attributes of the *city* circle except its radius.

3.2 Example 2: A Programmer's Use of Graphical Definitions

The previous example was an end-user-oriented application of graphical definitions that included only built-in types such as circles, with only constant parameters defining their attributes, such as the width of the sketched circle. Many programming environments that are intended for end-users provide ease of use, but do not support programmers without some kind of "trapdoor" mechanism leading to a different programming



(a)



(b)

Fig. 4. (a) The "user view" of the binary search program that Programmer A intends to write. The formula tabs indicate that *searchTree* and *searchElement* serve as inputs. (b) Programmer A's view of the Tree which will be used to implement the search. Most of the cells report information about the incoming tree (1). Tree gestures are enumerated at the top (2).

language than the easy-to-use top layer. (Typically the trapdoor leads either to an imperative macro language or to a traditional interpreted or incrementally compiled programming language, such as Lisp, Smalltalk, or Visual Basic). To support more sophisticated programmers without the use of trapdoors, graphical definitions needed to support complex, user-defined types (such as graphs or trees) whose parameters would not be restricted to constants, but rather could be based upon the relationships and operations that have been defined for these user-defined types (such as the left subtree of some tree). The example in this section uses a very traditional data structure processing task to show how graphical definitions support such types and parameters.

Suppose Programmer A wants to develop a binary search algorithm along the lines of Figure 4(a), using a binary tree (Figure 4(b)) that was previously implemented by Programmer B. This example focuses on how graphical definitions support programmers such as Programmer A in this kind of task. (Programmer B's implementation of the binary tree will be discussed in Section 4.) The user-defined tree type contains operations to insert a *new* element into a tree, report the *top* element of the tree, and report the *left* and *right* subtrees. The tree implementor (Programmer B) has previously defined gestures (Figure 4(b)), to perform these operations and to instantiate a new empty tree (the "top-level" gesture alluded to in the figure).

Like the population analyst in the previous example, Programmer A can use graphical definitions to experiment with and access different elements of the tree. Programmer A starts by using graphical definitions in cell *searchTree* to instantiate a new tree (via the "top level" gesture) and inserts three new elements (via three "new" gestures). For convenience, Programmer A has decided to set up three hidden (private) cells—*top*, *left*, and *right* (Figure 5(a))—each of which is defined using one gesture. For instance, the programmer can define cell *left's* formula by clicking on *searchTree* and drawing the *left* gesture—a line pointing down to the left (Figure 5(b)).

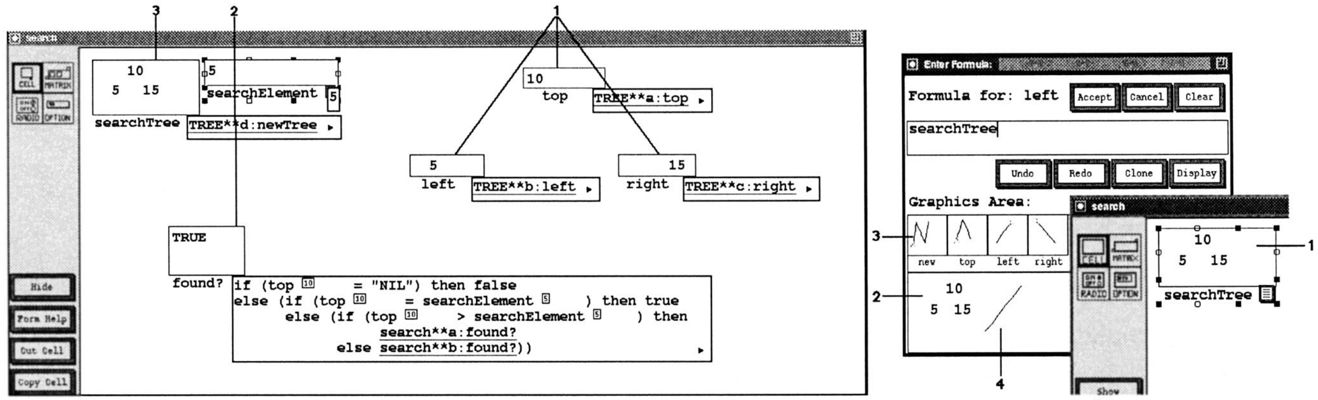
This *left* gesture is semantically equivalent to entering a pair of formulas in a more conventional way: namely, by copying the tree definition form of Figure 4(b), defining the formula for cell *inputTree* on the new copy to be a reference to the search tree, and referencing cell *left*. Making copies of a form and changing a few of the formulas on the copy to reference cells on other forms (similar to the "linked spreadsheets" of commercial spreadsheets) provides the same functionality as parameter passing and is general enough to support both constant and nonconstant parameters. Since Programmer B has previously set up the *left* gesture to be equivalent to this pair of formulas (using techniques that will be shown in Section 4), it includes the same "parameters" as would the textual formulas in this context. However, unlike the actions of copying the form and writing textual formulas, the gestural specification corresponds directly to the italicized words in Programmer A's intent: "I want *that* tree's *left* subtree."

To finish up the program, Programmer A needs to specify the recursive formula for *found?*. The way recursion is done in Forms/3 is by pointing at cells on copies of the form being defined, which are then automatically generalized using a deductive technique [Yang and Burnett 1994]. Thus, Programmer A copies the search form twice and defines the first and second copies' *searchTree* cells to refer to the original's *left* and *right* cells, respectively. The recursive references in *found?* are then specified by pointing at the *found?* cells on the copies.

To automatically generalize the recursive calls to all other copies, the dataflow subgraph containing all relationships among the original form and its first two copies that lead to *found?* are first analyzed to eliminate duplicated information about relationships. Then the resulting acyclic subgraph is topologically sorted. The result of the sort is a sequence of references that describes the computation in terms of relationships among the original and its copies. This allows the system to express formulas in terms of these relationships rather than in terms of references to specific copies. Since there is enough information present in the system to perform the tasks just described, inference[3] is not needed; rather, the generalization algorithm uses solely deductive reasoning.

To allow users to view the results of generalization, the static representation, which was designed using benchmarks quantifying some of Green's and Petre's work from the psychology of programming area [Green and Petre 1996; Yang et al. 1997], includes a complete representation of the

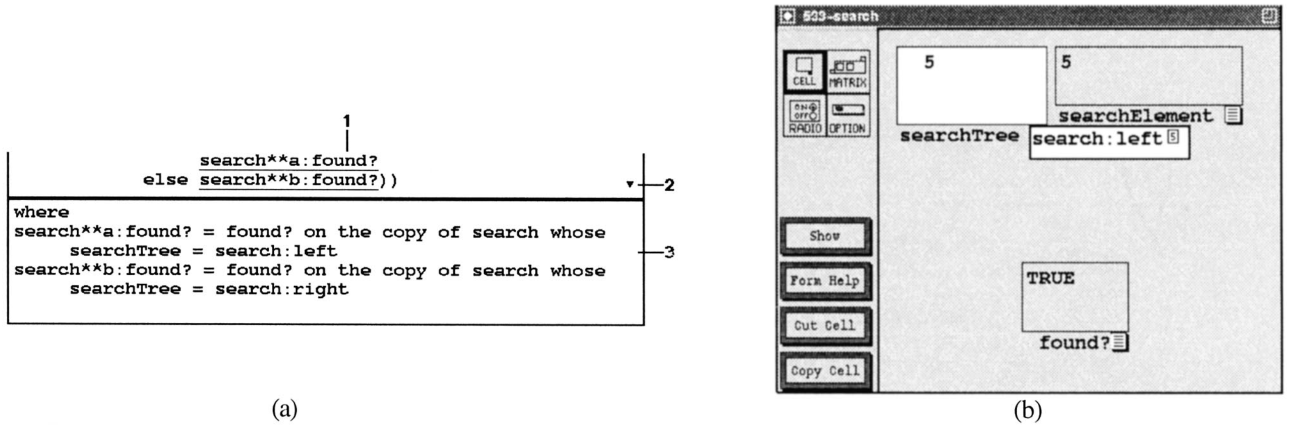
[3]In much of the AI literature, the term *inference* includes both sound reasoning techniques, such as deduction, and techniques employing guesswork. However, in literature about demonstrational programming languages, the term is normally used to mean only reasoning techniques employing guesswork. In this article we follow this latter convention.



```
found? if (top 10    = "NIL") then false
       else (if (top 10    = searchElement 5    ) then true
            else (if (top 10    > searchElement 5    ) then
                       search**a:found?
                  else search**b:found?))
```

```
Formula for: left
searchTree
```

Fig. 5. (a) The search program. There are 3 hidden cells (1), all of whose formulas were entered using gestures. The recursive formula for cell *found?* (2) was entered using a combination of typing and pointing. (3) marks an input cell *searchTree*. Its sample value was entered using gestures. (b) To program cell *left*, Programmer A clicked on the search tree (1) to set the context (2) for the gesture. Iconic representations of the tree gestures were then automatically displayed (3). The programmer then drew a gesture (4) to reference the left subtree.



```
                                1
                                |
             search**a:found?
        else search**b:found?))
                                          ▾—2
where
search**a:found? = found? on the copy of search whose
      searchTree = search:left                   —3
search**b:found? = found? on the copy of search whose
      searchTree = search:right
```

Fig. 6. (a) The underlined references such as *search**a* (1) in *found?*'s formula have been automatically generalized. Programmer A can point at the legend arrow (2) at the bottom of *found?*'s formula to see these generalized definitions (3). Also, the programmer can touch the underlined reference to bring up a concrete instance of the generalized form being referred to, which is shown in (b). Cell *searchTree* is highlighted because it identifies how this form instance differs from the original (in Figure 5).

generalized formulas via the textual portions of Figure 5 and Figure 6. These textual portions alone, however, would be rather indirect. To add directness, the static representation includes miniaturized sketches of the current values displayed next to the formula references, which removes the indirection of having to locate the cell via its name in order to find out its value. The user can also see a "concretized" version of any form being referenced by simply pointing at the reference, as in Figure 6(b) ("show me what *that* refers to").

4. THE SEMANTICS OF GRAPHICAL DEFINITIONS

Thus far, we have informally described through examples how graphical definitions extend our previous work on graphical types in the spreadsheet paradigm. In this section, we present the semantics of graphical definitions more precisely by defining the mapping from graphical definitions to these graphical types.

4.1 A Brief Review of the Model for Graphical Types

The central philosophy of our past work on graphical types has been that in a language in which even intermediate values are automatically displayed (such as in spreadsheet languages' on-screen cells), all types are in a sense graphical [Burnett and Ambler 1994]. Graphical types are useful in a wide variety of applications, and examples of their uses include event-based programs [Burnett and Ambler 1992], inventory tracking [Burnett and Ambler 1994], an analog desktop clock, [van Zee et al. 1996], algorithm animation [Carlson et al. 1996], and an animated Turing Machine [DuPuis and Burnett 1997]. Since the purpose of graphical definitions is to provide a direct way to work with such graphical types, we briefly review here our underlying model of graphical types in the spreadsheet paradigm.

In keeping with the philosophy that all types are graphical, a type is the four-tuple: (components, operations, graphical representations, interactive behaviors). To define a new type τ, a programmer creates a *type definition form* (spreadsheet) F_τ which, following the spreadsheet paradigm, consists of cells with formulas. The form contains two distinguished cells: an *abstraction box*, which is a complex cell that defines the structure of the type as the composition of cells placed inside it (the first element of the four-tuple), and an *image cell*, whose formula defines the type's appearance(s) (the third element of the four-tuple). The other two elements of the four-tuple, operations and interactive behaviors for type τ, are specified by additional abstraction boxes and ordinary cells on F_τ. All cells inside abstraction boxes are hidden (private), and other cells can explicitly be made hidden by the programmer.

F_τ's distinguished abstraction box defines as its value a sample instance of type τ, and each additional instance τ_i of τ is defined by the distinguished abstraction box on a copy of F_τ, denoted F_{τ_i}, upon which formulas different from those on F_τ can be defined to allow individual differences among instances of type τ. Instances of type τ can be referred to by any cell but, except for cells on copies of F_τ, can only be operated upon in more substantive ways via the nonhidden cells (public operations) that have been defined on F_τ.

Note that in this model there is no theoretical distinction between built-in and user-defined types. Both are theoretically defined by the above four-tuples and are practically defined by their accompanying type definition forms. The only distinction is implementation, i.e., whether the type's definition form has already been provided by the language implementor.

F_{circle}, the circle form in Figure 1, is one example of a type definition form. Because circles are a built-in type, F_{circle} is provided in the language implementation. The abstraction box is *newCircle*, and the image cell is hidden because it is not useful to the user: its formula consists of noneditable low-level code that draws a circle with the attributes specified by the other cells and formulas on the form. If the user copies F_{circle} and changes the formulas on the resulting form F_{circle_1} for the attribute cells, a different instance of a circle is defined in F_{circle_1}'s abstraction box *newCircle*. Information about the new circle is available in F_{circle_1}'s cells such as *radius* and *lineForeColor*. The user can continue to make more circles by copying F_{circle} and specifying the formulas on each as desired. We term this way of working with graphical types the "copying technique." This copying technique, allowing a user to create multiple copies of a form and refer to various cells on those copies from a common spreadsheet, is the same idea as the "linked spreadsheets" of many commercial spreadsheet systems (although, unlike our system, commercial spreadsheet systems do not use linked spreadsheets to support defining and referring to graphical types). Prior to the development of graphical definitions, the copying technique was the only way to work with graphical types.

4.2 Adding Graphical Definitions: Semantics for Built-In Types

In order to add a graphical approach for specifying graphical objects to the model just described, we defined semantics that would map a programmer's direct manipulations and gestures into the elements of this model. First we consider the built-in types.

As was shown by population example, we have provided a gesture and a clickable icon allowing users to instantiate the built-in type *circle* using direct manipulation and gestures. We also provided gestures and clickable icons for two other built-in graphical types, *box* and *line*. The user's gesture (or click on the icon) defines a formula that is a reference to an abstraction box on a copy F_{τ_β} of the definition form F_τ for the built-in graphical type τ. The mappings from such actions to formulas are shown in Table II. As this table shows, the formulas for some of the cells on copy F_{τ_β} are defined by the attributes of the gesture itself: for instance, the circle gesture defines a reference to the abstraction box on a copy of F_{circle} in which the formula for cell *radius* is defined to be the radius of the drawn circle gesture. If the user clicks on the icon instead of drawing a gesture, the default formulas enumerated in Table II are used to define the attributes.

Table II. The Mapping from Gestures and Icon Clicks to Formulas for Built-In Types

Graphical Type	Action	Formula
Circle	draw circle of radius ρ	primitiveCircle (radius$\equiv\rho$):newCircle
	click on circle icon	primitiveCircle (radius$\equiv$25):newCircle
Box	draw box of width ω and height η	primitiveBox (width$\equiv\omega$, height$\equiv\eta$):newBox
	click on box icon	primitiveBox (width$\equiv$50, height$\equiv$50):newBox
Line	draw line with dx ξ and dy Ψ	primitiveLine (deltax$\equiv\xi$, deltay$\equiv\Psi$):newLine
	click on line icon	primitiveLine (deltax$\equiv$50, deltay$\equiv$50):newLine

In each case, the formula is a reference to an abstraction box on a definition form copy F_{τ_β}. The notation is F_{τ_β} (*DefList*):χ, where *DefList* is a list of formula definitions for each cell defined differently on form F_{τ_β} than on F_τ, and χ is the abstraction box being referenced on F_{τ_β}. The notation for each element of *DefList* is (X$\equiv\phi$), denoting that a cell X has the formula ϕ.

Table III. The Mapping from Direct Manipulation of an Object α to Formulas for Built-In Types

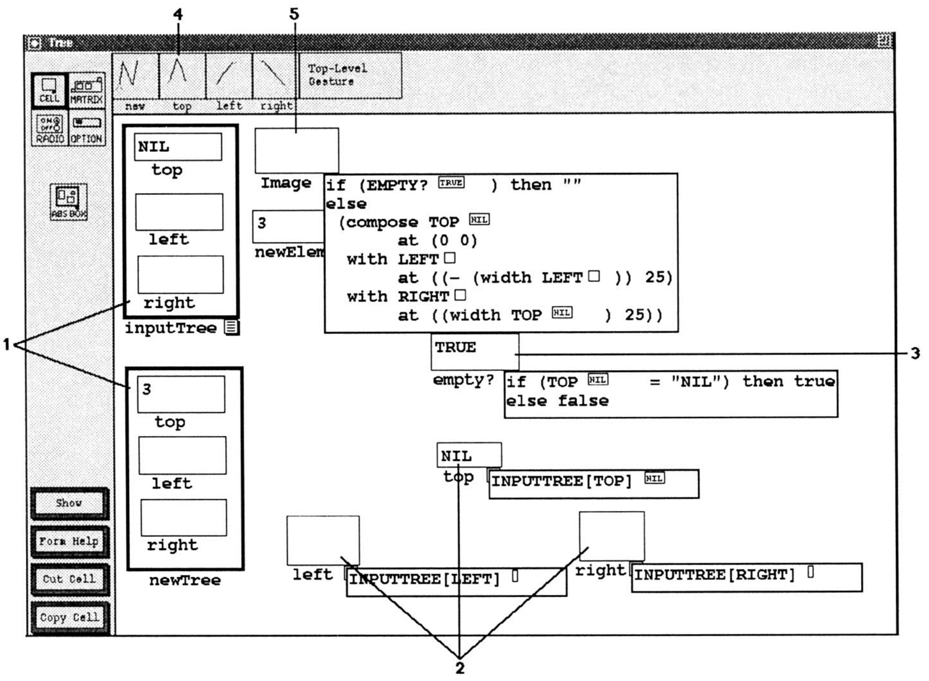
Graphical Type	Action	Formula
Circle	stretch edge of circle α to radius ρ	primitiveCircle(radius$\equiv\rho$, $cell_\forall\equiv cell_\alpha$): newCircle
Box	stretch corner of box α to width ω and height η	primitiveBox(width$\equiv\omega$, height$\equiv\eta$, $cell_\forall\equiv cell_\alpha$):newBox
Line	stretch line α's endpoint to position (ξ, Ψ)	primitiveLine(deltax$\equiv\xi$, deltay$\equiv\Psi$, $cell_\forall\equiv cell_\alpha$):newLine

In addition to the notation from Table II, the notation $cell_\forall\equiv cell_\alpha$ denotes that for all cells X not specified explicitly in the table, the formula for cell X on F_{τ_β} is the same as the formula for cell X on F_{τ_α}.

The other graphical way to create a new object β of type τ is to define a formula that refers to another object of the same type τ, and then to directly manipulate the object, such as shrinking the circle's radius as in Figure 3. These manipulations, like the gestures described above, specify a reference to an abstraction box on a copy F_{τ_β} of the definition form F_τ. The formulas for the cells on F_{τ_β} that do not depend on the attributes of the manipulation itself will be the same as those on F_{τ_α}, the definition form for the object α being manipulated. Table III defines this mapping from direct manipulations to formulas.

4.3 Preface to Example 3: Implementing the Tree Type

For a user-defined type to support direct manipulation and gestures, the new type must first be implemented. To do so, using the mechanism just described for graphical types, the programmer creates the type definition form, placing abstraction boxes and ordinary cells on it as needed and defining their formulas. Programmers will often use more than one abstraction box, placing an input abstraction box, other cells for input specifications and output information, and one or more output abstraction boxes on the definition form. Each abstraction box for a particular type definition



```
if (EMPTY? TRUE ) then ""
else
  (compose TOP NIL
        at (0 0)
   with LEFT □
        at ((- (width LEFT □ )) 25)
   with RIGHT □
        at ((width TOP NIL ) 25))
```

```
if (TOP NIL = "NIL") then true
else false
```

```
INPUTTREE[TOP] NIL
```

```
INPUTTREE[LEFT] □
```

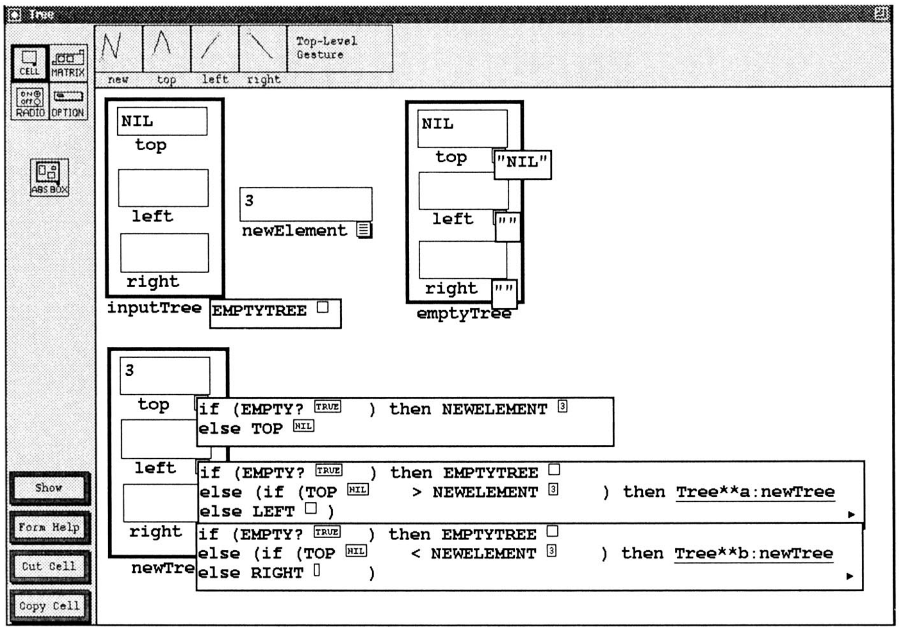
```
INPUTTREE[RIGHT] □
```

Fig. 7. The tree definition form's accessible cells (1–3) and gestures (4) as in Figure 4(b), plus the hidden cells used in the implementation of these accessors. Cells inside abstraction boxes (1) are by definition hidden (private). The image cell (5), which is also hidden, defines how instances of this type appear, and was specified by arranging the cells and rubberbanding the arrangement, and then modifying the x-coordinates to refer to widths of the components.

form must contain the same set of cells, although they may have different formulas. For example, recall Programmer A's view of the tree definition form (Figure 4(b)). The way the tree's implementor, Programmer B, implemented this type is shown in Figures 7 and 8. As these figures show, the form contains an input abstraction box *inputTree* intended to contain an incoming tree, input cell *newElement* for an element to be inserted into the tree, and output abstraction box *newTree* to define a tree into which the new element has been inserted. Other cells providing operations for the tree (such as the predicate reporting whether the incoming tree is empty, and a cell reporting the root element) are also present. Just as with the circle type, multiple instances of type *tree* can be instantiated using multiple copies of the tree form.

4.4 Semantics of Graphical Definitions for User-Defined Types

The semantics for gestures on user-defined types are a generalization of the semantics for built-in types, as can be seen by comparing Table IV with Tables II and III. (Direct manipulations can be viewed as gestures in the context of an existing instance of a type, and hence their semantics do not really need to be separated, although they were in Tables II and III, for clarity.)



```
if (EMPTY? TRUE ) then NEWELEMENT 3
else TOP NIL
```

```
if (EMPTY? TRUE ) then EMPTYTREE □
else (if (TOP NIL > NEWELEMENT 3 ) then Tree**a:newTree
else LEFT □ )
```

```
if (EMPTY? TRUE ) then EMPTYTREE □
else (if (TOP NIL < NEWELEMENT 3 ) then Tree**b:newTree
else RIGHT □ )
```

Fig. 8. The formulas defining how trees are constructed. The accessor cells have been moved aside to make room for these formulas to be displayed. The underlined references refer to generalized instances of the Tree form that recursively construct the left and right subtrees.

4.5 Example 3: Specifying Gesture Semantics

Here is how Programmer B provides the *formulaSpecs* in Table IV and Table V that map the desired gestures to the tree definition form's cells and formulas. The first step is to specify the set of gestures that are applicable to the type. In our implementation, gestures are defined and trained using the Agate gesture recognizer [Landay and Myers 1993], which is part of the Garnet environment [Myers et al. 1990]. Programmer B presses a button on the type definition form to start Agate, and then types the name of a gesture and draws a few examples of the gesture. Our implementation creates and displays miniature gesture icons at the top of the type definition form when Agate is exited. After demonstrating the gesture, the programmer specifies the gesture's semantics: the mapping from the gesture to a collection of formulas. For instance, the *new* gesture at the top of Figure 8 needs to be defined to mean a reference to *newTree* on a copy of the tree definition form, in which the formula for cell *newElement* is the element to be inserted into the tree, and the formula for the abstraction box *inputTree* is a reference to the tree being manipulated. This is accomplished as follows.

As indicated by Table IV, the semantics of a gesture on object α are specified by two things: a cell χ to be referenced on form F_{τ_β} (e.g., *newTree*), and formula specifications *formulaSpec*$_\forall$ for form F_{τ_β}'s other cells *cell*$_\forall$ (e.g., *newElement*). The input cells, namely those that are not hidden and have not had their formula tabs removed, are the ones that serve a parameter-like function; the formula specifications for these cells need to be provided

Table IV. The Mapping from a Graphical Definition Applied to Object α of Type τ to Formulas

Action	Formula
draw gesture, or click on gesture icon	$F_{\tau_\beta}(A \equiv \alpha, cell_\forall \equiv formulaSpec_\forall):\chi$

A represents the distinguished abstraction box, and χ represents the cell to be referenced on F_{τ_β}, which is a copy of F_{τ_α}. The programmer explicitly specifies which cell is χ in defining the new gesture's semantics. The notation $cell_\forall \equiv formulaSpec_\forall$ denotes that for every cell X other than A on F_{τ_β}, its formula is defined by the formula specifications in Table V.

Table V. Explicit Formula Specifications

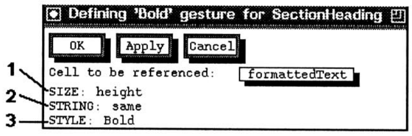
Type of Formula Specification	Permissible Formula Specification Values	Formula Defined for a Cell X on Form Copy F_{τ_β}
Gesture Attribute	height	*height of user's gesture*
	width	*width of user's gesture*
	radius	*radius of user's gesture*
	dx	*dx of user's gesture*
	dy	*dy of user's gesture*
Same	same	X_α
Constant	*anything*	*same as formula specification value*
askUser	ask "*string*"	*the user's response*

The programmer defines the *formulaSpec* of Table IV as a one-to-many mapping from a gesture G on some graphical object α to formulas for cells X on form F_{τ_β} using the specification types shown in this table. X_α is the cell on form F_{τ_α} corresponding to cell X on form F_{τ_β}. For the *askUser* formula specification, the keyword *ask* followed by the prompt "string" causes a dialog box to be displayed when the user makes the gesture; the user's response becomes the formula for cell X.

explicitly by Programmer B. In Figure 8, there is an input cell, *newElement*, that requires explicit formula specifications. There is also an input abstraction box in the figure: the distinguished abstraction box *inputTree*. The distinguished abstraction box is always an input cell, and its formula is always defined automatically to be a reference to α, the object being manipulated. Programmer B can use the four types of explicit formula specifications given in Table V's four rows, each of which is illustrated in Figure 9, to specify the semantics of input cells. All remaining cells on F_{τ_β} are defined implicitly to have the same formulas as on F_{τ_α}.

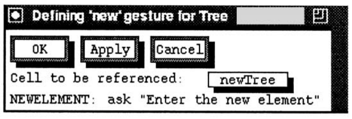
In addition to specifying gestures that derive one object from another, the programmer can specify a gesture to create an object not derived from any other object. To specify such a gesture, the programmer presses the "top-level gesture" button on the type's definition form and specifies a new gesture (whose name is the name of the type). This gesture is automatically added to the set of gestures understood by the top-level gesture recognizer.

Top-level gestures are important to the consistency of the approach for two reasons. First, they allow user-defined types to be instantiated using



```
Cell to be referenced: formattedText
SIZE: height
STRING: same
STYLE: Bold
```

(a)



```
Cell to be referenced: newTree
NEWELEMENT: ask "Enter the new element"
```

(b)

Fig. 9. Defining gesture semantics. (a) The *bold* gesture defines a reference to cell *formatted-Text* on a copy of the sectionHeading definition form in which (1) the formula for cell *size* is defined to be the height of the drawn gesture, (2) *string* is defined to be the same as the *string* formula for the sectionHeading object being manipulated, and (3) *style* is the constant "Bold." (b) The *new* tree gesture defines a reference to cell *newTree* on a copy of the tree definition form whose *newElement* formula is to be entered by the user at the time the gesture is drawn (see Figure 10).

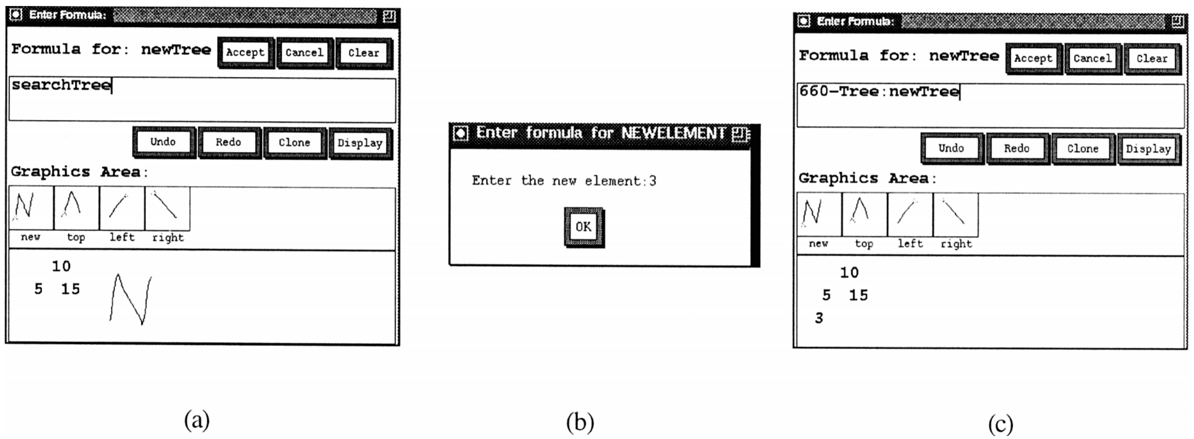
the same interface mechanism that is provided for built-in types. Second, they provide the same interface for instantiating a new graphical object "from scratch" as for deriving one object from another object.

5. HOW THE APPROACH AFFECTS THE SPREADSHEET PARADIGM

5.1 Graphical Definitions and the Value Rule

That graphical definitions are consistent with the value rule follows from the previous section, which showed how they map to the textual, spread-sheet formula mechanism. A key feature of graphical definitions that is needed for this consistency with the value rule and that is different from most demonstrational approaches is that what the user directly manipu-lates is the cell's formula, not its value. If graphical definitions operated on cell values as opposed to cell formulas, the spreadsheet value rule would be violated. For example, suppose there are two cells x and y, both of whose formulas are references to some cell *newCircle*, and the user was allowed to manipulate x's value directly by, say, shrinking its radius. In that case, manipulating x's value would be in contradiction of x's formula, which is supposed to have the same value as cell *newCircle*. Or, if interpreted to mean that both x and *newCircle* should change, then even y would be affected, even though y was not defined to be dependent upon x. Neither of these possibilities would be consistent with the value rule.

Another alternative would be for the user interface to be set up such that the user's manipulation of x's value did not change exactly that object (x's value), but rather changed x's formula. Such a solution would be consistent with the value rule, but may be misleading to the user, because the manipulations that the user was directly applying to the value would be actually happening to a subtly different object, namely the formula. To avoid this potential miscommunication, our user interface makes explicit the fact that the user is manipulating formulas by supporting graphical definitions in the formula edit windows instead of in the main part of the spreadsheet.



(a)

Enter Formula:

Formula for: newTree

searchTree

Graphics Area:

```
   10
 5  15
```

(b)

Enter formula for NEWELEMENT

Enter the new element:3

(c)

Enter Formula:

Formula for: newTree

660-Tree:newTree

Graphics Area:

```
   10
 5  15
3
```

Fig. 10. Using the graphical definition defined in Figure 9(b) to insert a new element into a tree. (a) The programmer draws the *new* gesture. (b) After drawing the gesture, the programmer is prompted for the element to be inserted. (c) The resulting formula is a reference to a new copy of the tree definition form in which cell *newElement* has the formula 3 and in which cell *inputTree* is a reference to the original tree.

5.2 Gesture Spaces and Context

A problem for gesture-based systems is that many gestures are too similar for the gesture recognizer to be able to differentiate them, and some researchers have advocated the desirability of supporting context-dependent gestures, e.g., "The system needs a way to map the same gesture into multiple meanings based on the context" [Landay and Myers 1993]. To date, however, only a few gesture-oriented approaches have incorporated the notion of context.

Our approach addresses this problem by defining context from a formula perspective: it defines the set of gestures recognized by the gesture classifier completely via the context established by the formula being edited. By partitioning the gestures into different *gesture spaces* (a concept similar to name spaces in traditional programming language literature), gestures' shapes need only be distinct within a specific context. This allows a gesture shape to have different meanings in different contexts without introducing ambiguity. For example, a carat-shaped gesture (^) might be an insertion mark in the context of a text editor, but a "roof of" accessor when working with a building object.

The "scope rules" that determine which gestures are applicable in the current context are simple. If the formula being edited is a reference to an instance of type τ, then the set of gestures for type τ—and only those gestures—will be recognized. Otherwise, the recognized gestures are the set of top-level gestures. For example, recall from Figure 5 that the programmer clicked on an instance of type *tree* to establish the context for cell *left*'s gestures.

In the work of Gross and Do, as in ours, gestures are only applicable in certain contexts, but in their system the context is inferred and may not yet be defined at the time a particular gesture is drawn, and thus the meaning of the gesture may be ambiguous [Gross and Do 1996]. Such ambiguities may be left unresolved until further information is added by the user. Since their system is intended to support conceptual and creative design, ambiguity may be an advantage because it supports the designer's creativity by allowing specific design choices to be deferred until some later time. In contrast, our approach is intended for programming, which is not particularly compatible with ambiguity.

Even given a well-defined way of keeping the number of operations (gestures) possible small, a problem that programmers face in any programming language is that of remembering *which* operations are permissible in that context, and this problem is exacerbated if the operations are dynamic gestures that are rarely actually seen. Our approach addresses this problem by displaying miniature icons of the allowable gestures (and their names) for the current context. These icons document the set of allowable operations, and, as was demonstrated in the population example, can be used as an alternative means of specifying gestures: rather than drawing a gesture, a programmer can click on a gesture icon. The partitioning of the gestures into different gesture spaces paired with the automatic display of the allowable gestures keeps the set of operations permissible at any one time small, recognizable, and visible.

Note that although this approach is effective for keeping the set of allowable gestures small at all levels except the top, it does not solve the problem of the size of the gesture space at the top level. For example, if there are many user-defined types, and if the programmer provides a top-level gesture for every type, then the top-level gesture space will be large. This problem remains unsolved and affects not only the iconic display of the gestures, but also the probability that one gesture will be misinterpreted as another by the system. A second, more manageable problem is that in our current implementation it is not possible to switch among multiple contexts within a single formula, such as to use the carat in one subexpression to mean "roof of" and in another subexpression of the same formula to mean "insert." In future work, we intend to switch context at each different cell reference instead of at each entire formula in order to solve this problem.

5.3 Scalability of Spreadsheet Languages

A practical side effect of graphical definitions is that they allow the screen real estate and memory usage of a spreadsheet program to be significantly less than that required under the copying technique, thus helping make spreadsheet languages more suitable for building large applications. Perhaps even more important to the programmer is that graphical definitions reduce the amount of work required to create programs containing graphical objects (Table VI). To consider a small example, building the population visualization program shown in Figure 2 without graphical definitions would have required the programmer to copy the circle definition form three times, to define the *radius* formula on each copy, and to reference each circle from the population form, whereas graphical definitions required only a single copy of the definition form to define the first circle's fill color. Although each graphical object specified with a graphical definition is defined by a definition form behind the scenes, only the graphical object itself is explicitly displayed on-screen; its definition form is shown only if the programmer elects to display it by clicking on the object. Because so many fewer visual components need to be constructed, displayed, and redrawn, supporting the programmer's manipulations requires less screen real estate, less memory, and less computation time.

6. EMPIRICAL STUDY

We devised graphical definitions because we thought their greater degree of directness would allow users to program more correctly or more quickly than with the previous approach, which uses a traditional, primarily textual way of working with spreadsheet formulas. In order to find out whether this hoped-for benefit had been achieved, we conducted an empirical study [Gottfried and Burnett 1997]. In this section, we describe the primary results of the study.

Table VI. Actions Needed to Create Graphical Objects *without* (top four) and *with* (bottom four) Graphical Definitions

To Create These Graphical Objects	Number of Formulas to Define	Number of Gestures	Number of Cells to Reference	Number of Off-Form Cells to Reference	Number of Type Definition Forms to Copy
three circles (population program)	9	N/A	3	3	3
n circles (population program)	$3n$	N/A	n	n	n
three-element search tree	6	N/A	3	3	3
n-element search tree	$2n$	N/A	n	n	n
three circles (population program)	4	3	2	0	1
n circles (population program)	$n + 1$	n	$n - 1$	0	1
three-element search tree	1	4	0	0	0
n-element search tree	1	$n + 1$	0	0	0

Programmers perform fewer actions using graphical definitions; in some cases the reduction is as much as a factor of n. Of particular importance is the reduction in the more complex programming actions, i.e., those that require multiple forms (linked spreadsheets), shown in the two rightmost columns.

The study was conducted one subject at a time at a workstation. The subjects were 20 computer science graduate students at Oregon State University. Each subject was given a scripted introduction to programming in Forms/3, followed by instruction on how to create boxes using either graphical definitions or a primarily textual technique (the copying technique). The subject was then asked to use the newly learned technique to create the colored circles in the population program. This problem session was followed by instruction in the second technique on the binary tree data type, and then a second programming task was given, in which the subject used the tree type to create a tree of three elements and accessed the left subtree of that tree.

As this description shows, the study was counterbalanced with regard to the programming method involved; that is, half the subjects completed the population program using graphical definitions, and the tree program using the copying technique, and the other half of the subjects did the same programs in the same order, but using the opposite techniques. The placement of subjects into the two groups was random, except that care was taken to ensure equal representation of subjects with Forms/3 experience in the two groups.[4]

[4]There were four such subjects, and their prior experience with Forms/3 was quite limited. Subsequent analysis showed that there was no significant correlation between performance in this study and prior experience with Forms/3 (Fisher's exact test, $p = 0.406$).

Table VII. Program Correctness

		Population		Tree		Total	
		n	%	*n*	%	*n*	%
Graphical Definitions	*Correct*	10	100%	9	90%	19	95%
	Incorrect	0	0%	1	10%	1	5%
Copying Technique	*Correct*	10	100%	4	40%	14	70%
	Incorrect	0	0%	6	60%	6	30%
Total	*Correct*	20	100%	13	65%	33	82.5%
	Incorrect	0	0%	7	35%	7	17.5%

The population problem was completed correctly by all subjects. The tree program was completed correctly by 90% of the subjects using graphical definitions, as compared to only 40% of the subjects using the copying technique.

Note that since the same program was always performed first, the second program had a learning advantage. However, because the study's results did not require any assumption that the problems be of equal difficulty, this did not affect their validity.

6.1 Correctness Results: Do Graphical Definitions Help Programmers Construct Correct Programs?

The two primary research questions were how graphical definitions affected the correctness and speed with which the subjects could complete their programming tasks. A summary of the results pertaining to the first of these, correctness, is shown in Table VII. For the population program, all subjects—both those using graphical definitions and those using the copying technique—were able to complete the program correctly. However, for the tree program, significantly more subjects were able to complete the program correctly using graphical definitions than those using the copying technique (Fisher's exact test, $p = 0.03$): whereas 90% of the subjects using graphical definitions completed the program correctly, only 40% of the subjects using the copying technique did so. These results produced a significant difference in the cumulative results, which show that, overall, significantly more programs were completed correctly with graphical definitions than with the copying technique (Fisher's exact test, $p = 0.05$).

6.2 Speed Results: Do Graphical Definitions Help Programmers Construct Programs More Quickly?

We measured the amount of time it took each subject to complete each program. Both programs were completed significantly faster using graphical definitions (population: Mann-Whitney test, $p < 0.02$; tree: Mann-Whitney, $p < 0.002$). In fact, in the tree program, each of the subjects who used graphical definitions completed the tree program faster than *any* subject who used the copying technique for that program. Further, the standard deviation for graphical definitions was much smaller on both problems than the standard deviation for the copying technique, demon-

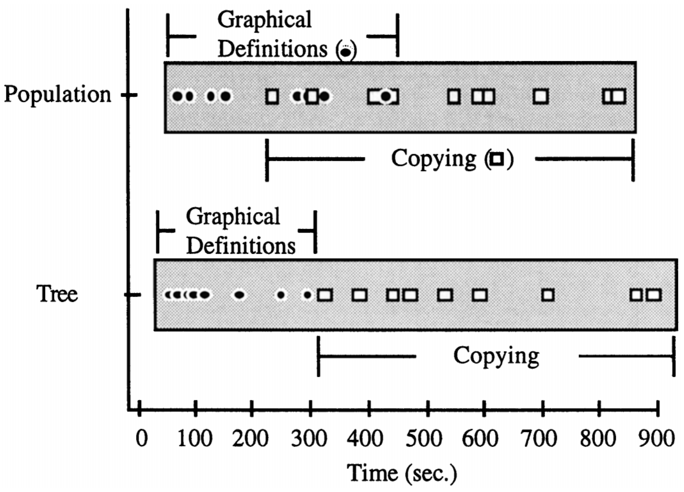


Fig. 11. Subjects' programming times on each problem.

strating the consistency of the faster times for graphical definitions. Figure 11 shows the data.

6.3 Programming Errors

One possible explanation of the fact that the subjects programmed more quickly using graphical definitions could have been the directness of graphical definitions, i.e., that directness was helping with the problem-solving aspect of the programming task. However, another possible explanation could have been that the speedup was simply a matter of eliminating typing, i.e., that the approach sped up entry time alone without improving any of the problem-solving aspects of programming. To investigate this possibility, we looked specifically at graphical definitions' effects on problem-solving difficulties by examining the numbers and types of errors subjects made along the way to completing the programs.

Only one subject using graphical definitions (10%) had any problems in defining the formulas on the tree program for cell *binTree*, and only one (10%) had difficulty with cell *left*. The subjects using the copying technique, on the other hand, encountered several difficulties. When instantiating the tree, several subjects made errors in defining the relationships between cells on the various copies of the tree definition form. Many of these errors were later corrected, in part because the subjects could see that the formulas they had defined were not producing the results they had expected. However, in defining the formula for cell *left*, this continuous immediate feedback may actually have contributed to some errors: a few subjects defined incorrect formulas (such as creating a new tree with the elements 3 and 8) that *looked* correct, but did not define the correct relationship. See Table VIII. As this table shows, significantly more errors were made when the subjects used the copying technique than when they used graphical definitions (Fisher's exact test, $p = 0.001$).

Table VIII. Difficulties Encountered on the Tree Program

		binTree	left	entire program
		(number of subjects)		
Graphical Definitions	*none*	9	9	8
	minor	1	0	1
	major	0	1	1
Copying	*none*	3	3	2
	minor	3	2	3
	major	4	5	5

Minor difficulties consisted of incorrect formulas that were quickly discovered and corrected. Major difficulties included numerous minor errors or errors that were never corrected.

6.4 Do Programmers Prefer to Draw Gestures or to Click on Gesture Icons When Using Graphical Definitions?

In using the graphical definitions technique, the subjects could either draw a gesture or click on a gesture icon. For some formulas, such as those used in the tree program, these two actions specified exactly the same formula. For the circles used in the population program, however, drawing a gesture specified more information (the size of the circle) than did clicking on the gesture icon. Thus, a subject who clicked on the gesture icon would need to perform an extra action of resizing the circle using direct manipulation.

We were interested in determining which of these techniques the subjects preferred and whether this depended on the problem they had solved using graphical definitions. We asked the question, "When you used gestures, did you prefer to draw the gesture or click on the gesture icon?" The possible answers to this question on the questionnaire were drawing the gesture, clicking on the gesture icon, or using both techniques. A little over half of the subjects (55%) said they preferred to click on the gesture icon, while most of the others said they preferred to use both techniques ($\chi^2 = 7.90$, df = 2, $p < 0.02$). Only one subject preferred solely to draw the gesture (Table IX).

6.5 Limitations of the Study and Questions for Followup

Because controlled experiments such as this are not done under "real-world" conditions, it is important to use caution in generalizing from them. Our subjects were graduate students, and further experimentation is needed with different populations, especially with spreadsheet end-users and with professional programmers comfortable with spreadsheets. Also, the tasks were very small, designed to fit in a time reasonable for the controlled experiment, and how well the results generalize to larger, more realistic programming tasks has not been shown.

The subjects in the study were given about 15 minutes of training on the first technique before they worked on the first program, which was followed by a similar period of training on the second technique, and then the second program. One possible concern that arises in any experiment with this design is the extent to which the training itself influenced the results of the

Table IX. The Subjects' Preferred Method of Using Graphical Definitions

		Problem Done Using Graphical Definitions		
		Population	*Tree*	*Total*
Preferred Method	*draw*	0	1	1
	click	7	4	11
	both	3	5	8

study. For instance, did the results reflect the techniques themselves, or only the effectiveness of the training approach on the two techniques? Although we took steps to minimize this possibility, it was not feasible to eliminate the possibility completely. One strategy we used was to reduce dependence on factors related solely to training quality, such as memorization and reliance on visually oriented training devices, by providing written information sheets as part of the training, to which subjects were able to refer throughout the problem-solving sessions. We also trained on exactly the same functionality in both techniques, to ensure the equivalence of the material being covered during the training.

A related question requiring followup study is what impact the first technique learned had on the second program the subjects worked. For instance, the subjects who used graphical definitions on the tree problem had previously been trained on the copying technique for the population problem; did knowing the copying technique improve their performance in using graphical definitions on the tree? Further experimentation on this question is needed, particularly to determine whether the effectiveness of graphical definitions is impacted by understanding of the copying technique.

Several followup questions arise regarding the subjects' willingness to draw gestures. Since in our approach the gesture's context must be established before the gesture begins (e.g., by referencing a tree), rather than by where a gesture's "hotpoint" falls, each gestural operation takes two actions (a click and a gesture) instead of the one it would have taken with hotpoints. This decreased the efficiency advantage that could be gained using the gestures as opposed to the clickable icons, and may have influenced subjects' interest in using the actual gestures. Another factor may have been that our gesture device was a mouse. Especially in the population program and the preceding training with boxes, we noticed that several subjects had difficulties drawing circle and box gestures. Since a mouse is not particularly well suited to the task of gesturing, this probably made use of gestures less attractive to the subjects than it would otherwise have been. In fact, some subjects had difficulty in using the mouse to draw gestures at all, and others initially drew circular gestures that were incorrectly recognized by the gesture recognizer as boxes or lines. (Gesture recognition errors did not occur with the tree gestures, which, unlike the circle gesture, all consisted of straight line segments.) Even with a different gesture device such as a pen, however, any gesture recognition software necessarily includes a possibility of error, and this also seems likely to affect subjects' willingness to use gestures. In the graphical definitions approach, the drawback of gesture recognition difficulties is tempered because of the clickable icons, which offer an alternative graphical means for entering shapes, and subjects used this alternative often. Indeed, the subjects' preferences indicated that the clickable icon alternative was important to them, and it is possible that this ability to click when gesture recognition was imperfect was critical to our study's strongly positive results for graphical definitions.

6.6 Relationship to Other Studies

As was discussed in Related Work (Section 2.1), other than spreadsheets that employ macros written in other languages, there are only a few spreadsheet languages that support either complex or graphical objects. Thus, it is not surprising that our empirical study is the first to study the use of dynamic graphics to program such objects in spreadsheet languages. In fact, using dynamic graphics in programming is not yet well-studied even outside of spreadsheet languages, and hence it is useful to consider how the results of our study relate to the other studies to date about the use of graphics in programming.

In the area of visual programming languages (VPLs), there are surprisingly few studies on the usefulness of *dynamic* graphical devices for programming, such as direct manipulation and gestures. Most of the studies that have been done on VPLs have been intended to answer some kind of "static graphics versus static text" question regarding comprehension (as opposed to program construction), and most of them have concentrated on diagrammatic programming via flowcharts, dataflow diagrams, and Petri nets (e.g., see Green et al. [1991], Moher et al. [1993], and Petre [1995]). See Whitley [1997] for an excellent survey of that work. There is, however, a little work relating to direct manipulation and/or dynamic graphics and how it affects people's ability to comprehend and work with previously existing programs through algorithm animation [Lawrence et al. 1994, Stasko et al. 1993] and in debugging [Cook et al. 1997; Wilcox et al. 1997]. These studies did not find statistically significant results for all the aspects studied, but for the aspects in which statistical significance was found, the dynamic approaches were found to be superior to the static approaches.

Unlike the studies in the previous paragraph, our study relates to constructing new programs or portions of programs, not to debugging or understanding existing programs. To date, there are only a few other reports on the effects of graphical techniques for constructing new programs, and most of these reports investigate static, not dynamic, graphics. One study on program construction with static graphics compared ability to construct matrix programs using a static version of Forms/3 with ability to construct the same programs using static versions of two textual languages, and it found that subjects constructed more correct programs using Forms/3 [Pandey and Burnett 1993]. Baroth and Hartsough report one of the few industrial studies of VPLs, a three-month study in which program construction of a sizable project in the dataflow language LabView produced a higher-quality solution in a shorter time period than was achieved when a different team programmed the same problem in C [Baroth and Hartsough 1995]. Since this was an industrial experiment involving a real project, there were many uncontrolled variables, but it may be the only report evaluating the construction of real, industrial-strength programs using a graphical technique.

Regarding the use of dynamic graphics in constructing new programs, such as programming-by-demonstration techniques, most of the studies that we have been able to locate are informal evaluations and user studies intended to verify usability or to discover potential improvements by observing where users have trouble with the system. See Cypher [1993] for several examples of these studies. However, there is one recent empirical study on a by-demonstration visual shell language about the importance of the representation used to reflect dynamic graphics used in programming [Modugno et al. 1996]. This study found that the use of dynamic graphics paired with a static graphical representation was more effective than the use of dynamic graphics paired with a static textual representation.

One interpretation of the results of our study is that it adds to the body of evidence favoring graphics over static text in constructing new programs. However, a slightly different interpretation, and the one to which we subscribe, is that the results are more an indicator that the use of directness, which is a central feature of the graphical definitions approach, can lead to reduced errors and increased programming speed. Our study seems to be the first to quantitatively analyze a direct approach compared with a computationally equivalent indirect approach, and we are hopeful that more such studies will be done by others, because empirically establishing the effects of directness in the domain of programming could have significant implications for the design of future visual languages and environments.

7. CONCLUSION

Although direct manipulation is not new to programming, most other approaches have been imperative, and even those few direct-manipulation approaches that have been declarative have not fit with the spreadsheet paradigm. As a result, spreadsheet languages have either been limited to supporting only the simplest of textual types—numbers and strings—within the computations themselves, or have incorporated indirect approaches to support other types.

Graphical definitions remove this limitation, allowing graphical objects to be seamlessly integrated into the spreadsheet paradigm without the use of macros, state-modifying mechanisms, or indirection. The approach is expressive enough to support a wide range of uses, from programmable graphics such as circles and boxes likely to be of interest to end-users to graphical data structures for programmers. The approach supports these capabilities while still satisfying the principles of directness advocated by

Shneiderman; by Hutchins, Hollan, and Norman; by Green and Petre; and by Nardi. Two strategies central to this result that are unique to graphical definitions are (1) complete adherence to the spreadsheet value rule of *all* parts of the program, including the definition of new user-defined types, and (2) the approach's use of gesture spaces and visible, clickable gesture icons to keep the number of gestures applicable in any one context small and visible.

The empirical study showed that graphical definitions were quite usable by the subjects. In particular, compared with the traditional, primarily textual way of entering spreadsheet formulas, graphical definitions led to a significantly smaller number of programming errors, greater degree of correctness in the completed programs, and faster programming speed. Although these results greatly favored the graphical definitions technique, there were indications that the effectiveness of the gestures aspect was highly dependent on the quality of the gesture recognition, which may mean that having an alternative program entry mechanism (such as clickable icons) may be critical in maintaining these strongly favorable results.

ACKNOWLEDGMENTS

We would like to thank the members of our research group, especially John Atwood, Rebecca Walpole, and Sherry Yang, for their work on the Forms/3 design and implementation and for their feedback on graphical definitions. Special thanks also go to Shikha Ghosh Gottfried and Judith Hays for their help in testing the empirical study's design, and to the subjects of the study for their participation.

REFERENCES

ATWOOD, J., BURNETT, M., WALPOLE, R., WILCOX, E., AND YANG, S. 1996. Steering programs via time travel. In *Proceedings of the 1996 IEEE Symposium on Visual Languages* (Boulder, Colo., Sept. 3–6). IEEE Computer Society Press, Los Alamitos, Calif., 4–11.

BAROTH, E. AND HARTSOUGH, C. 1995. Visual programming in the real world. In *Visual Object-Oriented Programming: Concepts and Environments*, M. Burnett, A. Goldberg, and T. Lewis, Eds. Prentice-Hall, Englewood Cliffs, N.J.

BURNETT, M. AND AMBLER, A. 1992. A declarative approach to event-handling in visual programming languages. In *Proceedings of the 1992 IEEE Workshop on Visual Languages* (Seattle, Wash., Sept.). IEEE Computer Society Press, Los Alamitos, Calif., 34–40.

BURNETT, M. AND AMBLER, A. 1994. Interactive visual data abstraction in a declarative visual programming language. *J. Vis. Lang. Comput. 5,* 1 (Mar.). 29–60.

CARLSON, P., BURNETT, M., AND CADIZ, J. J. 1996. A seamless integration of algorithm animation into a visual programming language. In *Proceedings of Advanced Visual Interfaces '96* (Gubbio, Italy, May 27–29). ACM, New York, 194–202.

COOK, C., BURNETT, M., AND BOOM, D. 1997. A bug's eye view of immediate visual feedback in direct-manipulation programming systems. In *Empirical Studies of Programmers: Proceedings of the 7th Workshop* (Alexandria, Va., Oct. 24–26). ACM, New York, 20–41.

CYPHER, A., ED. 1993. *Watch What I Do: Programming by Demonstration*. MIT Press, Cambridge, Mass.

DUPUIS, C. AND BURNETT, M. 1997. An animated Turing machine simulator in Forms/3. Tech. Rep. TR 97-60-08, Dept. of Computer Science, Oregon State Univ., Corvallis, Ore.

GOTTFRIED, H. AND BURNETT, M. 1997. Programming complex objects in spreadsheets: An empirical study comparing textual formula entry with direct manipulation and gestures. In *Empirical Studies of Programmers: Proceedings of the 7th Workshop* (Alexandria, Va., Oct. 24–26). ACM, New York, 42–68.

GREEN, T. AND PETRE, M. 1996. Usability analysis of visual programming environments: A "cognitive dimensions" framework. *J. Vis. Lang. Comput. 7,* 2 (June). 131–174.

GREEN, T., PETRE, M., AND BELLAMY, R. 1991. Comprehensibility of visual and textual programs: A test of superlativism against the "match-mismatch" conjecture. In *Empirical Studies of Programmers: Proceedings of the 4th Workshop* (New Brunswick, N.J., Dec. 7–9). Ablex, Norwood, N.J., 121–146.

GROSS, M. AND DO, E. 1996. Ambiguous intentions: A paper-like interface for creative design. In *Proceedings of the ACM Symposium on User Interface Software and Technology* (Seattle, Wash., Nov. 6–9). ACM, New York, 183–192.

HUDSON, S. 1994. User interface specification using an enhanced spreadsheet model. *ACM Trans. Graph. 13,* 3 (July). 209–239.

HUGHES, C. AND MOSHELL, J. 1990. Action Graphics: A spreadsheet-based language for animated simulation. In *Visual Languages and Applications*, T. Ichikawa, E. Jungert, and R. Korfhage, Eds. Plenum Publishing, New York, 203–235.

HUTCHINS, E., HOLLAN, J., AND NORMAN, D. 1986. Direct manipulation interfaces. In *User Centered System Design: New Perspectives on Human-Computer Interaction*, D. Norman and S. Draper, Eds. Lawrence Erlbaum, Hillsdale, N.J., 87–124.

KAY, A. 1984. Computer software. *Sci. Am. 250*, 5 (Sept.). 53–59.

KURLANDER, D. 1993. Chimera: Example-based graphical editing. In *Watch What I Do: Programming by Demonstration*, A. Cypher, Ed. MIT Press, Cambridge, Mass.

LANDAY, J. AND MYERS, B. 1993. Extending an existing user interface toolkit to support gesture recognition. In *Adjunct Proceedings INTERCHI '93* (Amsterdam, The Netherlands, Apr. 24–29). ACM, New York, 91–92.

LAWRENCE, A., BADRE, A., AND STASKO, J. 1994. Empirically evaluating the use of animations to teach algorithms. In *Proceedings of the 1994 IEEE Symposium on Visual Languages* (St. Louis, Mo., Oct. 4–7). IEEE Computer Society Press, Los Alamitos, Calif., 48–54.

LEOPOLD, J. AND AMBLER, A. 1997. Keyboardless visual programming using voice, handwriting, and gesture. In *Proceedings of the 1997 IEEE Symposium on Visual Languages* (Capri, Italy, Sept. 23–26). IEEE Computer Society Press, Los Alamitos, Calif., 28–35.

LEWIS, C. 1990. NoPumpG: Creating interactive graphics with spreadsheet machinery. In *Visual Programming Environments: Paradigms and Systems*, E. Glinert, Ed. IEEE Computer Society Press, Los Alamitos, Calif., 526–546.

LIEBERMAN, H. 1993. Mondrian: A teachable graphical editor. In *Watch What I Do: Programming by Demonstration*, A. Cypher, Ed. MIT Press, Cambridge, Mass.

MCCARTNEY, T., GOLDMAN, K., AND STAFF, D. 1995. EUPHORIA: End-user construction of direct manipulation user interfaces for distributed applications. *Softw. Concepts Tools 16,* 4, 147–159.

MIYASHITA, K., MATSUOKA, S., TAKAHASHI, S., YONEZAWA, A., AND KAMADA, T. 1992. Declarative programming of graphical interfaces by visual examples. In *Proceedings of the ACM Symposium on User Interface Software and Technology* (Monterey, Calif., Nov. 15–18). ACM, New York, 107–116.

MIYASHITA, K., MATSUOKA, S., TAKAHASHI, S., AND YONEZAWA, A. 1994. Iterative generation of graphical user interfaces by multiple visual examples. In *Proceedings of the ACM Symposium on User Interface Software and Technology* (Marina del Rey, Calif., Nov. 2–4). ACM, New York, 85–94.

MODUGNO, F., CORBETT, A., AND MYERS, B. 1996. Evaluating program representation in a demonstrational visual shell. In *Empirical Studies of Programmers: Proceedings of the 6th Workshop* (Alexandria, Va., Jan.). Ablex, Norwood, N.J., 131–146.

MOHER, T., MAK, D., BLUMENTHAL, B., AND LEVENTHAL, L. 1993. Comparing the comprehensibility of textual and graphical programs: The case of Petri nets. In *Empirical Studies of Programmers: Proceedings of the 5th Workshop* (Palo Alto, Calif.). Ablex, Norwood, N.J., 137–161.

Myers, B. 1991. Graphical techniques in a spreadsheet for specifying user interfaces. In *Proceedings of CHI '91: Conference on Human Factors in Computing Systems* (New Orleans, La., Apr. 28–May 2). ACM, New York, 243–249.

Myers, B., Guise, D., Dannenberg, R., Vander Zanden, B., Kosbie, D., Pervin, E., Mickish, A., and Marchal, P. 1990. Garnet: Comprehensive support for graphical, highly interactive user interfaces. *Computer 23,* 11 (Nov.). 71–85.

Nardi, B. 1993. *A Small Matter of Programming: Perspectives on End User Computing*. MIT Press, Cambridge, Mass.

Pandey, R. and Burnett, M. 1993. Is it easier to write matrix manipulation programs visually or textually? An empirical study. In *Proceedings of the 1993 IEEE Symposium on Visual Languages* (Bergen, Norway, Aug. 24–27). IEEE Computer Society Press, Los Alamitos, Calif., 344–351.

Petre, M. 1995. Why looking isn't always seeing: Readership skills and graphical programming. *Commun. ACM 38,* 6 (June), 33–44.

Repenning, A. and Ambach, J. 1996. Tactile programming: A unified manipulation paradigm supporting program comprehension, composition and sharing. In *Proceedings of the 1996 IEEE Symposium on Visual Languages* (Boulder, Colo., Sept. 3–6). IEEE Computer Society Press, Los Alamitos, Calif., 102–109.

Shneiderman, B. 1983. Direct manipulation: A step beyond programming languages. *Computer 16,* 8 (Aug.), 57–69.

Smedley, T., Cox, P., and Byrne, S. 1996. Expanding the utility of spreadsheets through the integration of visual programming and user interface objects. In *Proceedings of Advanced Visual Interfaces '96* (Gubbio, Italy, May 27–29). ACM, New York, 148–155.

Smith, D., Cypher, A., and Spohrer, J. 1994. KidSim: Programming agents without a programming language. *Commun. ACM 37,* 7 (July), 54–67.

Stasko, J., Badre, A., and Lewis, C. 1993. Do algorithm animations assist learning? An empirical study and analysis. In *Proceedings of INTERCHI '93* (Amsterdam, The Netherlands, Apr. 24–29). ACM, New York, 61–66.

Vander Zanden, B. and Myers, B. 1995. Demonstrational and constraint-based technologies for pictorially specifying application objects and behaviors. *ACM Trans. Comput. Hum. Interact. 2,* 4 (Dec.), 308–356.

van Zee, P., Burnett, M., and Chesire, M. 1996. Retire Superman: Handling exceptions seamlessly in declarative visual programming languages. In *Proceedings of the 1996 IEEE Symposium on Visual Languages* (Boulder, Colo., Sept. 3–6). IEEE Computer Society Press, Los Alamitos, Calif., 222–230.

Whitley, K. 1997. Visual programming languages and the empirical evidence for and against. *J. Vis. Lang. Comput. 8,* 1 (Feb.), 109–142.

Wilcox, E., Atwood, J., Burnett, M., Cadiz, J., and Cook, C. 1997. Does continuous visual feedback aid debugging in direct-manipulation programming systems? In *Proceedings of CHI '97: Conference on Human Factors in Computing Systems* (Atlanta, Ga., Mar. 22–27). ACM, New York, 258–265.

Wilde, N. and Lewis, C. 1990. Spreadsheet-based interactive graphics: From prototype to tool. In *Proceedings of the CHI '90 Conference on Human Factors in Computing Systems* (Seattle, Wash., Apr. 1–5). ACM, New York, 153–159.

Wilde, N. 1993. A WYSIWYC (what you see is what you compute) spreadsheet. In *Proceedings of the 1993 IEEE Symposium on Visual Languages* (Bergen, Norway, Aug. 24–27). IEEE Computer Society Press, Los Alamitos, Calif., 72–76.

Yang, S. and Burnett, M. 1994. From concrete forms to generalized abstractions through perspective-oriented analysis of logical relationships. In *Proceedings of the 1994 IEEE Symposium on Visual Languages* (St. Louis, Mo., Oct. 4–7). IEEE Computer Society Press, Los Alamitos, Calif., 6–14.

Yang, S., Burnett, M., DeKoven, E., and Zloof, M. 1997. Representation design benchmarks: A design-time aid for VPL navigable static representations. *J. Vis. Lang. Comput. 8,* 5/6 (Oct./Dec.), 563–599.

Received August 1997; revised December 1997; accepted January 1998