

## Interactive Visual Data Abstraction in a Declarative Visual Programming Language

MARGARET M. BURNETT\*‡ AND ALLEN L. AMBLER†

*\*Department of Computer Science, Oregon State University, Corvallis, Oregon 97331, U.S.A.  
and † Department of Electrical Engineering and Computer Science, University of Kansas,  
Lawrence, Kansas 66045, U.S.A.*

*Received 17 April 1993 and accepted 2 November 1993*

Visual data abstraction is the concept of data abstraction for visual languages. In this paper, first we discuss how the requirements of data abstraction for visual languages differ from the requirements for traditional textual languages. We then present a declarative approach to visual data abstraction in the language Forms/3. Within the context of this system, issues of particular importance to declarative visual languages are examined. These issues include enforcing information hiding through visual techniques, supporting abstraction while preserving concreteness, conceptual simplicity and specification of a type's appearance and interactive behavior as part of its definition. Interactive behavior is seen to be part of the larger problem of event-handling in a declarative language. A significant feature is that all programming and execution are done in a fully-integrated visual manner, without requiring other languages or tools for any part of the programming process.

### 1. Introduction

THE IDEA OF VISUAL PROGRAMMING is intuitively very appealing because it allows the programmer to express relationships among, or transformations to, data simply by sketching them, pointing at them or demonstrating them—not by translating them into sequences of commands, pointers and abstract symbols. Declarative visual programming languages (VPLs) contribute the additional strength that the programmer has only to specify *what* the solution is, not *how* the computer must go about modifying and manipulating the contents of memory to arrive at a solution. These simplifications and removal of many of the concepts traditionally required to program, combined with continuous visual feedback, seem to have great promise in making programming easier and more reliable. Yet, the potential of declarative VPLs has remained largely untapped. One reason for this has been the lack of an approach to data abstraction that is suitable for declarative visual programming languages. Without data abstraction, expressive power and high-level programming capabilities are very limited because the programmer must always express all manipulations on data in terms of the low-level details.

An *abstract data type* is a type traditionally specified as a collection of other types and a set of operations. It is expected that programmers will operate upon instances of abstract data types only through these defined operations. *Data abstraction* is the concept of abstract data types plus the information hiding that enforces use of abstract data types only through the defined operations.

---

‡ To whom correspondence should be addressed.

There has been little work done that supports data abstraction in declarative VPLs. Prior approaches have fallen into three categories: (1) approaches in which a VPL is used for only part of the programming task, requiring switching to another (textual) programming language for the remainder; (2) approaches designed for special-purpose problem domains which can be handled via combinations of a few “built-in” types; and (3) approaches that are not fully declarative. These approaches cannot be said to offer general solutions to the problem of data abstraction for declarative visual programming languages because the first approach solves the problem only with the help of traditional textual languages, the second lacks generality and the third is not declarative. Our work differs from these in that it does not require the use of external languages or tools, it is general and it is fully declarative.

Our goal was to find a way to achieve the needed power by exploiting the visualness and conceptual simplicity of VPLs. In our efforts to achieve this goal, we started at the foundations. Rather than finding a visual way to incorporate data abstraction as defined for textual programming languages, we decided to instead start afresh by exploring the concept of data abstraction for *visual* languages. We term this concept *visual data abstraction*. Visual data abstraction is different from traditional data abstraction in two ways: (1) it adds graphical representations and interactive behaviors to a type’s definition; and (2) it is accomplished entirely through visual programming mechanisms. In developing this concept, we wanted to achieve the following:

- The *definition* of a new data type will be visual and concrete, and will be viewable at any time. This implies that definitions can only be accomplished as part of the VPL. This requirement cannot be satisfied via ‘visual code generators’ because such approaches produce textual, not visual, definitions.
- The *process* of creating, changing and working with instances of the data type will be interactive and visual, featuring concrete, immediate visual feedback. Approaches in which data types are created textually but can later be used visually do not provide this characteristic.
- The *data resulting* from such a definition will be interactive and visual. This implies that its appearance is visual, and that the appearance is part of the definition of the type. It also implies that the definition includes specifications about its behavior under user interaction.

### 1.1. Example Scenarios for Visual Data Abstraction in VPLs

The following programming scenarios provide an intuitive sense of the goals that underlie the research presented in this paper. The first scenario presents a visual, interactive kind of data whose creation would be artificial and cumbersome without the use of a VPL. The second scenario demonstrates the same kind of visual, interactive programming style for a traditional abstract data type. Later in this paper we will return to these examples to explain how they can be programmed using the declarative VPL Forms/3 in which we prototyped our approach to visual data abstraction.

#### 1.1.1. A Movable Object in a Window

An object within a window can be moved by grabbing it and flicking it off in some direction. As long as the mouse button is down, the object is actually being dragged.

But if the mouse button is released while the mouse is still moving, the object continues to move in the direction and speed used while it was being dragged. The movement of the object continues until it arrives at a border, or until it is again grabbed. This problem is indicative of the kind encountered in any sort of gestural interface, which generally requires the ability to detect and abstract arbitrary sequences of events. Such sequences may be quite complex, going far beyond the capabilities of systems able only to recognize discrete button clicks, mouse movements and keystrokes.

There are two types of data that need to be programmed in this scenario—windows and movable objects. A window is simply the composition of the objects on it, and its only operation is the computation of its appearance. A movable object consists of a shape, size, position and event history. The primary task in defining it consists of specifying its appearance and position based upon its event history.

Programming these two interactive visual data types without being able to work with them visually would be frustrating, rather like trying to create a painting blindfolded. But in Forms/3 the process of programming them is highly visual. For example, throughout the programming process the appearance of the window dynamically provides continuous visual feedback as to the speed, direction and location of the object. Part of this feedback is due to the fact that everything on the screen has a value as soon as it is defined. This allows the programmer to experiment while programming through the use of concrete sample values. For example, by providing a definition for the movable object's appearance, an instance of it immediately appears and can be observed. As the programmer experiments with the application, his or her interest changes from low-level events (did the mouse just *move*?), to higher level events (is the object being *dragged*?). The programmer can navigate through these levels of abstraction, sometimes viewing the details of the components of the data types, other times watching its behavior as a whole, switching between alternative views as needed. As the programmer interactively modifies the program, the effects of these changes can be immediately observed because the results are continuously displayed.

The completed movable object type will be usable in a variety of applications. Instances of it can be used directly on an instance of the window type as in the program development process described above, or they can be incorporated into other types (customers, stacks, etc.) to give them the same movability properties.

### 1.1.2. *A Production Planning Simulation*

In this scenario, the goal is to minimize handling of the goods being produced by either avoiding inventory altogether (producing exactly enough to meet demands), or by using LIFO inventory for easiest storage and retrieval of goods from inventory when demand does not exactly match supply. The programmer begins by defining a stack needed to handle LIFO inventory. While defining the components of the stack and its operations using the visual, interactive mechanisms of the language, the programmer begins to test the simulation by interactively placing two items in an inventory stack. The stack is displayed on the screen, and the simulation pops the first one (which causes the stack's appearance on the screen to change), and then the second one. While watching the simulation, the programmer interacts with it to test

other circumstances, instantiating more stacks with different inventory items, increasing the production rate causing a build-up of inventory, etc. If there are troublesome aspects of the simulation (why is this stack so large? how old is the oldest item in it?), the programmer can inspect the detailed components of the data. As needed, he or she changes the simulation program, changes the definition of the stack operations and adds alternative views of the stacks, all using the same programming constructs and visual notations. All changes are immediately reflected in the data that appears on the screen.

Long after the programming of the production planning problem is complete, the programmer uses the same stack type in a maze-playing game, an infix expression parser for a calculator, and a variety of other applications. This reuse of the stack does not require re-programming multiple versions of the same code or dropping down into a translated textual version, but rather can be done by simple manipulations to the existing code using the same visual programming constructs.

## 1.2. Organization of this Paper

In this paper we present a declarative approach to visual data abstraction. After a discussion of related work in Section 2, we present a brief introduction to the language Forms/3 in Section 3, and then describe the approach to visual data abstraction in Sections 4 through 6.

## 2. Related work

Most earlier visual languages have not attempted to provide visual constructs for data abstraction and none have attempted fully-declarative solutions, but several have contributed ideas that are related. ThinkerToy [1], a concrete modeling environment for decision support problems, is a visual language which supports many aspects of visual data abstraction for a specific problem domain. It uses a concrete, object-oriented approach to allow the user to define the composition and inheritance of new data types in an entirely visual manner, and to compose these new data types into larger constructions. However, it does not visually support more abstract kinds of data. Other successful visual languages designed for specific problem domains (for example, ThingLab [2] and Action Graphics [3]) have incorporated varying degrees of support for the creation of problem-specific data types, but are not intended to support greater generality.

Several object-oriented systems, such as Hi-Visual [4], use coarse-grained visual approaches. In these systems, definition of most new abstract data types is of a finer grain than is supported by the visual part of the system, and is done at least in part via a traditional textual language. The GRClass system [5] focuses directly on data abstraction and supports completely visual definition of new abstract data types. GRClass is a visual tool for Andrew Toolkit data structure code generation. Thus, it supports visualness only during data type definition, generating textual code which is then incorporated into a textual program that is maintained and debugged in the traditional way. In ObjectWorld [6], concrete, continuous feedback is provided throughout a visual process of defining general-purpose abstract data types (objects), but the methods for the objects are programmed using a textual language.

Prograph [7, 8] is a language in the declarative family with strong support for general-purpose abstract data types. Prograph is an object-oriented dataflow visual language, although it does not claim to be strictly declarative—some side-effecting constructs have been introduced. A university research project for several years, it is now a commercial product for the Macintosh. The support for the definition of new abstract data types is completely visual, and is tightly coupled with the visual execution environment, but the information-hiding aspect of data abstraction is not present.

Much of the work supporting interactivity in types has been done in special-purpose VPLs specifically pertaining to user interfaces, and has not generally been tied to data abstraction. InterCONS [9] is a visual dataflow language which has certain primitives associated with user interaction (e.g. buttons, sliders). These primitives accept interactive input events, and generate integer outputs which can be routed into other dataflow nodes for further calculation. The system is intended only for the specific problem domain of user-interface generation. A similar approach can be found in ConMan [10]. Prograph also provides such an approach, but supports more generality by providing dataflow connectivity to the Macintosh Toolbox methods. This allows direct access and manipulation of the various Macintosh user-interface and operating system primitives.

Fabrik [11, 12] does not support abstract data types, but it does allow certain kinds of interactive aggregate types. In Fabrik, primitive graphical objects called graphemes (e.g. rectangles, bitmaps) can be sensitized to certain kinds of interactive events. The sensors that are incorporated into these graphemes provide continuous tracking information, such as the mouse's position. This is similar in concept to the Forms/3 approach, but is different in that it ties event detection to specific, viewable objects. A similarity between Fabrik and Forms/3 is that both also support non-interactive events such as system clock ticks.

While NoPump II [13] does not include a general approach to data abstraction or to interactive events, it does support certain specific events, namely system clock ticks and the relocation of objects on the screen. These events are supported through the ability to reference event-supporting cells on a spreadsheet, one being the clock cell and the others being pairs of coordinate cells for graphical objects. The authors also propose adding an additional distinguished cell to report the state of the mouse's button. However, generality has not been a goal of the NoPump system, and the generality of the event-handling proposed is insufficient for complex interactions for several reasons. First, there is no access to mouse movement events unless an object is being dragged. Second, due to the lack of any facility for abstraction, interactions of any complexity would require a large and detailed group of coordinate and sequencing comparisons, making the program's intended logic obscure.

Schoberth's proposal [14] for event-handling in Forms/2 [15], the predecessor of Forms/3, had a strong influence on Forms/3's approach to interactive events. In Forms/2, a form is a collection of cells. Each cell has a formula defining its value and a set of attributes which can be formulated in cells themselves. In an extension of this fact, events are represented by special-attribute cells whose values are maintained and updated automatically whenever events occur in the cells they describe. Attribute cells' values can then be referenced by formulas in other cells, thus contributing to calculations as needed. Values dependent on newly-arrived events are marked

out-of-date by creating a new entry in time for the affected forms with the out-of-date cells' values shown as 'unknown' (eventually to be re-computed if needed). The special attributes supported are `mouseIn`, `mouseDown`, `keyboardValue` and `focusIn`. In addition, a special form has three event cells (`mouseLocX`, `mouseLocY` and `clock`) which are always kept up-to-date by the system in the same manner as above. Forms/2 does not support construction of new types. Our current approach to events is different from the Forms/2 approach, but borrows from its declarative, time-based aspects.

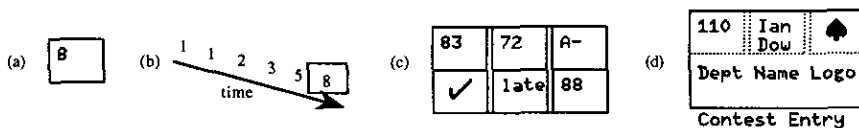
### 3. Forms/3, a Declarative Visual Programming Language

Our work was done using the declarative VPL Forms/3 as a prototype. In order to explain the approach to visual data abstraction, we first present Forms/3. This section includes an informal presentation of basic concepts, a more rigorous treatment of the same basic concepts and definitions of some of the primitives. The complete specification is given in Burnett [16].

Forms/3 is a form-based VPL that incorporates facilities for procedural abstraction, visual data abstraction, polymorphic types with type inference, file handling, error handling, and both interactive and non-interactive events. Research prototypes without user interfaces<sup>a</sup> have been implemented for Sun 3/160s and SPARCstation/2 color workstations using Lucid Common Lisp and the CLX interface to X Windows. A new implementation of the system including a user interface is currently being developed.

Forms/3 is a declarative VPL. It is declarative because all programming is done by defining formulas for *cells*, which can be referenced by formulas in other cells. These formulas are expressions such as  $'2 + 3'$  which define the value of the cell. Circular formulas are not allowed in Forms/3. There is no mechanism for side-effecting state modification, and the only way a cell can get a value is through its own formula. Because of the lack of state modification, the property of referential transparency<sup>b</sup> is preserved.

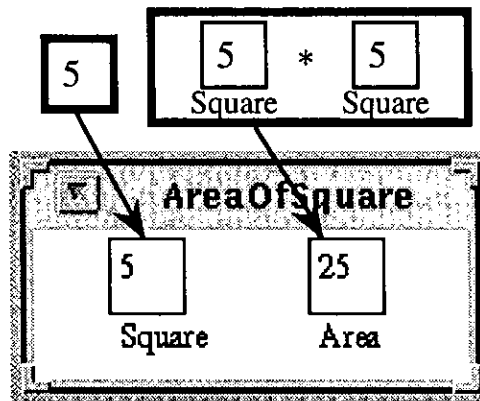
The cells are arranged on *forms* as desired by the user. Cells can also be grouped into a *matrix* or an *abstraction box* which can be referenced like cells from other formulas. The term *referenceable object* denotes a cell, a matrix or an abstraction box,



**Figure 1.** (a) A cell. (b) A sketch of the temporal vector of values defined for the cell. (c) A matrix of six cells representing a student's grades. (d) An abstraction box whose cells constitute the parts of a visual abstract data type. Any of these can be named by the user if desired as in (d)

<sup>a</sup> For this reason, hand-drawn formulas simulating those which would be displayed by the user interface are superimposed on the screen shots in the figures.

<sup>b</sup> The property of referential transparency says that given the same arguments a module will always produce the same outputs, no matter how many times it is executed. This property is not preserved for languages in which it is possible to have mutable (modifiable) values affecting a computation that are not part of the parameter list.



**Figure 2.** Computation of the area of a square. The cell *Square* represents a square by the length of one of its sides. The formulas are shown superimposed on the screen dump. *Square* has the formula 5. The cell *Area* multiplies a reference to *Square* times itself to compute the area

i.e. anything that can be referenced in a formula (see Figure 1). Figure 2 contains a very small programming example in Forms/3. Table 1 presents more precise definitions of this terminology.

Although there is often quite a bit of text in Forms/3 formulas, Forms/3 is a visual programming language. One reason for this is that direct manipulation is part of the language syntax; although it is possible to name cells and refer to them by these textual names, names are optional and the only mechanism for referencing an unnamed cell is by pointing at it with the mouse. Another reason that Forms/3 is a visual programming language is that the scope rules, which are explained in Section 4, are both defined and enforced only through visual mechanisms. Finally, the textual operations often seen in the formulas are actually shortcuts (reminiscent of keyboard shortcuts for menu items) for cell references made by direct manipulation on system-provided forms, as defined in Table 2 and Table 3.

In Forms/3, a cell's formula may define a sequence of values over time, termed a *temporal vector*. There are two differences between temporal vectors and other vectors traditionally found in programming. The first is that subscripting is defined along a logical time dimension rather than a space dimension, and the second is that temporal vectors need not have an element at every point on the temporal dimension. Temporal vectors differ from the sequential 'streams' often found in functional or dataflow languages because elements of temporal vectors can be directly referenced via subscripts. Another difference from streams is that there is no notion of data moving or being consumed in temporal vectors; every element is defined at a fixed location, and remains there forever (at least in theory). The positions along the logical time dimension are not defined by the passage of real time, but rather by the occurrence of computational events. A sketch of the temporal dimension is given in Figure 1, and several examples of its use are given later in this paper.

#### 4. Visual Data Abstraction in a Declarative Language

In this section we will first discuss visual abstract data types (VADTs) and their specification in Forms/3, and then turn our attention to the information-hiding aspect

Table 1. Definitions of the constructs of Forms/3.

A <i>form</i> = $(N, ID, AL, Rset)$ , where $N$ is a user-assigned name, $ID$ is a unique identifier, $AL$ is an attribute list, $Rset$ is a set of referenceable object tuples
A <i>referenceable-object-tuple</i> = $(RO, AL, P)$ , where $RO$ is a referenceable object, $AL$ is an attribute list and $P$ is the x-y position of $RO$
A <i>referenceable object</i> = a cell, a matrix or an abstraction box <sup>a</sup>
A <i>cell</i> = $(ID, F)$ , where $ID$ is a unique identifier and $F$ is a formula
A <i>formula</i> = $rc \rightarrow TV$ , where $rc \in ROset \cup Cset$ , $ROset$ is the set of all referenceable objects, $Cset$ is the set of all constants and $TV$ is a temporal vector
A <i>temporal vector</i> is a vector of elements along the temporal dimension
The element $V_i$ in temporal vector $V$ <i>accessible</i> at time $i$ is the element in $V$ at the maximum time $j$ at which a value is defined, where $j \leq i$
An <i>attribute list</i> = $\{(x, y) \mid (x, y) \in HA \cup FA\}$ , where $HA$ is the set of attribute pairs provided by the host windowing system, and $FA^b$ is the set of host-independent attribute pairs defined for Forms/3
A <i>matrix</i> <sup>c</sup> = $(ID, MFset, RowSeq, Dims)$ where $ID$ is a unique identifier, $MFset$ is a set of $MF$ s where each $MF = rc \rightarrow \{F\}$ , $RowSeq$ is a sequence of $Rows$ , a <i>Row</i> is a sequence of referenceable-object-tuples $(RO, AL, P)$ such that each $RO$ is a cell, $Dims = \{(NR, AL, P), (NC, AL, P)\}$ , $NR$ and $NC$ are distinguished cells whose formulas define the size of all $RowSeq$ s and $Rows^d$ , respectively

<sup>a</sup> The definition of abstraction boxes is deferred until Section 4.<sup>b</sup> For brevity,  $FA$  is not enumerated here. Examples:  $(Name, "Square") \in FA$ ,  $(lineColor, blue) \in HA$ .<sup>c</sup> Details of how the user can specify matrices are beyond the scope of this paper, but can be found in Vichstaedt and Ambler [22].<sup>d</sup> The term *columns* is often used informally to designate the size of rows.



**Table 2.** This table defines the semantics of many of the built-in primitive forms used in this paper. The complete set of definitions is given in Burnett [16]. In the formal specification of Forms/3, there are no formula operators—only references to other referenceable objects or to constants, as shown here. An optional textual syntax with operators is layered on top of these formulas, and is defined in Table 3. Notation:  $X::\langle T^* \rangle$  denotes that  $X$ 's type is a temporal vector of 0 or more values of type  $T$ .  $X::\langle T^+ \rangle$  denotes that  $X$ 's type is a temporal vector of 1 or more values of type  $T$ . If  $T$  is a Greek letter, the type is polymorphic.  $X_t$  is the element of  $X$ 's temporal vector accessible at time  $t$ , and  $X_{-t}$  is the element of  $X$ 's temporal vector that is not accessible at time  $t$  with the maximum subscript  $< t$ .

$N$	Ref. objects	Preconditions	Postconditions
+	$A, B$ and $C$	$A::\langle \text{number}^* \rangle$ $B::\langle \text{number}^* \rangle$	$A$ and $B$ are unchanged $C::\langle \text{number}^* \rangle, C_t = A_t + B_t$
=	$A, B$ and $C$	$A::\langle \text{number}^* \rangle$ $B::\langle \text{number}^* \rangle$	$A$ and $B$ are unchanged $C::\langle \text{boolean}^* \rangle, C_t = A_t = B_t$
and	$A, B$ and $C$	$A::\langle \text{boolean}^* \rangle$ $B::\langle \text{boolean}^* \rangle$	$A$ and $B$ are unchanged $C::\langle \text{boolean}^* \rangle, C_t = A_t$ and $B_t$
if	$A, B, C$ and $D$	$A::\langle \text{boolean}^* \rangle$ $B::\langle \beta^* \rangle$ $C::\langle \chi^* \rangle$	$A, B$ and $C$ are unchanged $D::\langle \delta^* \rangle$ $D_t = B_t$ if $A_t = \text{true}$ $C_t$ otherwise
ifThen	$A, B$ and $C$	$A::\langle \text{boolean}^* \rangle$ $B::\langle \beta^* \rangle$	$A$ and $B$ are unchanged $C::\langle \chi^* \rangle$ $C_t = B_t$ if $A_t = \text{true}$ $C_{t-1}$ if $C_{t-1}$ exists and not $A_t = \text{true}$ NOVALUE otherwise

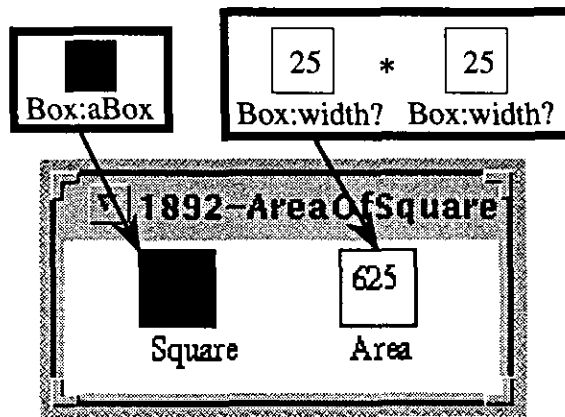
Table 2—(continued)

N	Ref. objects	Preconditions	Postconditions
prev	A, B, UNTIL and D	$A::\langle\alpha^+\rangle$ $B::\langle\beta^+\rangle$ UNTIL:: $\langle\text{boolean}^+\rangle$	A, B and UNTIL are unchanged $D::\langle\delta^+\rangle$ $D_t \equiv B_{-t}$ if $B_{-t}$ exists, $\forall t < = u$ $A_t$ if $B_{-t}$ does not exist, $\forall t < = u$ where $u$ is the minimum $t$ at which UNTIL <sub><math>t</math></sub> = true
earlier	A, B, UNTIL and D	$A::\langle\alpha^+\rangle$ $B::\langle\beta^+\rangle$ UNTIL:: $\langle\text{boolean}^+\rangle$	A, B and UNTIL are unchanged $D::\langle\delta^+\rangle$ $D_t \equiv B_{t-1}$ if $B_{t-1}$ exists, $\forall t < = u$ $A_t$ if $B_{t-1}$ does not exist, $\forall t < = u$ where $u$ is the minimum $t$ at which UNTIL <sub><math>t</math></sub> = true
image Compose	image1, image2, Result, x1, y1, x2, y2	image1:: $\langle\alpha^+\rangle$ image2:: $\langle\beta^+\rangle$ x1, y1, x2, y2:: $\langle\text{number}^+\rangle$	image1, image2, x1, y1, x2, y2 are unchanged Result:: $\langle\chi^+\rangle$ Result <sub><math>t</math></sub> = image1, positioned at (x1, y1) relative to the upper left corner of Result's bounding box, and image2, positioned at (x2, y2) relative to the upper left corner of Result's bounding box, where the z-position <sup>a</sup> of image2 <sub><math>t</math></sub> is not less than the z-position of image1,

<sup>a</sup> The z-axis is a spatial dimension increasing toward the user. Thus, if image1, and image2, share any pixels in common, image2 <sub>$t$</sub>  spatially overlaps (appears 'on top of') image1 <sub>$t$</sub> .

**Table 3.** This table is a partial list of valid textual expressions, defined in terms of references to referenceable objects on built-in forms. Parsing of such expressions is strictly left to right unless the user groups subexpressions with parentheses. The notation is  $FC(DefList):RO$ , where  $FC$  denotes a copy of original form  $F$ ,  $DefList$  is a list of formula definitions of each referenceable object that is defined differently on  $FC$  than on  $F$ , and  $RO$  is a referenceable object of  $FC$ . The notation for each element of  $DefList$  is  $(X \rightarrow \alpha)$ , denoting that referenceable object  $X$ 's formula has been defined to be  $\alpha$ , where  $\alpha$  is a constant, a referenceable object or a valid textual expression.

$N$	Ref. objects	The notation	Is equivalent to a reference to
$+$	$A, B$ and $C$	$\alpha + \beta$	$+(A \rightarrow \alpha, B \rightarrow \beta):C$
$=$	$A, B$ and $C$	$\alpha = \beta$	$=(A \rightarrow \alpha, B \rightarrow \beta):C$
and	$A, B$ and $C$	$\alpha$ and $\beta$	and( $A \rightarrow \alpha, B \rightarrow \beta$ ): $C$
if	$A, B, C$ and $D$	if $\alpha$ then $\beta$ else $\chi$	if( $A \rightarrow \alpha, B \rightarrow \beta, C \rightarrow \chi$ ): $D$
ifThen	$A, B$ and $C$	if $\alpha$ then $\beta$	ifThen( $A \rightarrow \alpha, B \rightarrow \beta$ ): $C$
prev	$A, B, UNTIL$ and $D$	prev $\beta$	prev( $A \rightarrow NOVALUE, B \rightarrow \beta, UNTIL \rightarrow NOVALUE$ ): $D$
earlier	$A, B, UNTIL$ and $D$	earlier $\beta$	earlier( $A \rightarrow NOVALUE, B \rightarrow \beta, UNTIL \rightarrow NOVALUE$ ): $D$
image Compose	image1, image2, Result, $x1$ , $y1, x2, y2$	compose $\mu$ at $(\alpha, \beta)$ with $v$ at $(\chi, \delta)$	imageCompose(image1 $\rightarrow \mu$ , image2 $\rightarrow v, x1 \rightarrow \alpha, y1 \rightarrow \beta, x2 \rightarrow \chi, y2 \rightarrow \delta$ ):Result



**Figure 3.** Another version of the computation of the area of a square. This version uses graphical data types. Here cell *Square* is a reference to a cell named *aBox* on a form named *Box*, and *Area* multiplies a reference to cell *width?* on form *Box* times itself to compute the area. Form *Box* is shown in Figure 4

of visual data abstraction. The notion of a VADT is that it is a 4-tuple: (components, operations, graphical representations, interactive behaviors). The third and fourth elements of the tuple are important differences between the concept of a visual abstract data type and the traditional textual concept of an abstract data type.

In Figure 2 of the previous section we presented a very small programming example in Forms/3. However, in a visual language it seems artificial to represent a square by a number measuring the length of one side. Instead, it seems more appropriate to represent an inherently graphical data type such as a square by a ■, as in Figure 3. Notice the reference to cells on form *Box*. Form *Box* is an example of a VADT definition that in this case is provided with the language,<sup>c</sup> and is shown in Figure 4. The user can incorporate a box into his/her programming simply by referencing cell *aBox* in a formula, as in the example. The next section will describe how the user can define new VADTs that can be used in the same way as the built-in types by simply referencing cells on forms.

**Figure 4.** This copy of the built-in form *Box* describes an instance of the primitive type *Box*

<sup>c</sup> The built-in VADTs are: Boolean, box, error, eventReceptor, file, glyph, line, number and text. Every VADT, whether built-in or user-defined, is defined by a form, as will be discussed in the next subsection. The formal definitions of the built-in VADTs are given axiomatically via pre- and postconditions for referenceable objects on those forms, in the same manner as the primitives defined in Table 2.

#### 4.1. Declaratively Programming a New Visual Abstract Data Type

In Forms/3, a visual abstract data type is defined by placing cells and groups of cells on a form. A form defining a VADT is termed a *visual abstract data type definition form*. There are two distinctions between the definition of a VADT definition form and that of other forms: (1) a VADT definition form contains at least one *abstraction box* which defines in detail the physical parts of the data; and (2) a VADT definition form contains a distinguished cell whose formula defines the appearance of the data (called the *image cell*). In addition, a distinction between the use of VADT definition form versus other forms is that the (user-assigned) name of a VADT definition form defines the type name of all values produced by abstraction boxes on any instance of that form (see Table 4). We will demonstrate the use of these concepts first with a simple example in which the user will visually create a traditional stack. Later examples will show the creation of other, less traditional, kinds of data types.

The user's process of creating a new stack VADT is as follows. First, he/she creates a new VADT definition form via menu selection, and names it 'stack'. An abstraction box and the image cell are automatically created by the system and placed on the form. The user defines formulas for these objects and places additional objects on the form, defining their formulas as shown in Figure 5. The abstraction box *aStack* contains a one-dimensional matrix named *elements*. The matrix shown in the figure has zero columns which is reflected in the unbordered cell attached to the matrix. More about the possible formulas for the elements of this matrix and for the image cell will be discussed in later sections. In this example, the user has also chosen to make a stack's size, topmost element and empty status accessible by defining appropriate formulas, as shown in the figure.

The appearance of the form and the cells on it are entirely controlled by the user. The user may place cells wherever desired, specifying formatting information via menus in a manner reminiscent of spreadsheets. For example, the user also has placed the phrase 'Information about the stack:' on the form for documentation purposes. This phrase is actually just another cell with a formula, in this example a constant formula, and the user has set up the format of that cell to use  $8 \times 13$  bold font with no border around the cell. There are no restrictions on the kinds or placement of these additional objects; they are the same as any other referenceable object on the form and may have as simple or complex formulas as desired. Another documentation technique is naming. Although the user has chosen to name all the cells in this

**Table 4.** Definitions of the VADT constructs of Forms/3.

A *VADT definition form* is a form with the restriction that *Rset* contains at least one referenceable-object-tuple  $(RO, AL, P)$  in which *RO* is an abstraction box, and contains exactly one referenceable-object-tuple  $(RO, AL, P)$  in which *RO* is a cell designated as the image cell

An *abstraction box* on VADT definition form  $(N, ID, AL, Rset)$  is  $(AID, F, ARset)$  where *AID* is a unique identifier,

*ARset* is a set of referenceable-object-tuples  $(RO, AL, P)$  such that each *RO* is a cell or a matrix,

$F = \{RO, \forall ROs \mid RO \text{ is in a referenceable-object-tuple } \in ARset\} \rightarrow N$ , where the mapping operation is composition

example, naming is strictly optional since references to other cells in a formula can be made simply by pointing at them with the mouse.

The user continues with the definition of the stack by adding two more abstraction boxes, naming them *Pushed* and *Popped*, as in Figure 6. All abstraction boxes on the same form must have the same structure. Each abstraction box on form 'stack', by virtue of its presence on that form, results in an instance of type 'stack'. The purpose of *Pushed* is to define a stack based upon the original stack (*aStack*), but with one more element on the top. This approach of defining different stacks resulting from different kinds of operations is entirely declarative and does not involve state modification—the original value of *aStack* is unchanged since the only way its value is ever computed is through its own formula or those of its enclosed cells and matrices. This declarative approach does not carry efficiency penalties because it is implemented largely through pointers, thus avoiding the time and space associated with copying.

So far, we have described the process by which a user can define a new visual abstract data type. Significant features are that the approach is entirely declarative; that it is entirely interactive and visual, involving no external languages or tools in any part of the programming process; and that it is conceptually simple because all programming is done by simply providing formulas for cells. In the next sections we will give an example of the usage of the new stack type, while also looking more deeply into the programming language issues of the approach.

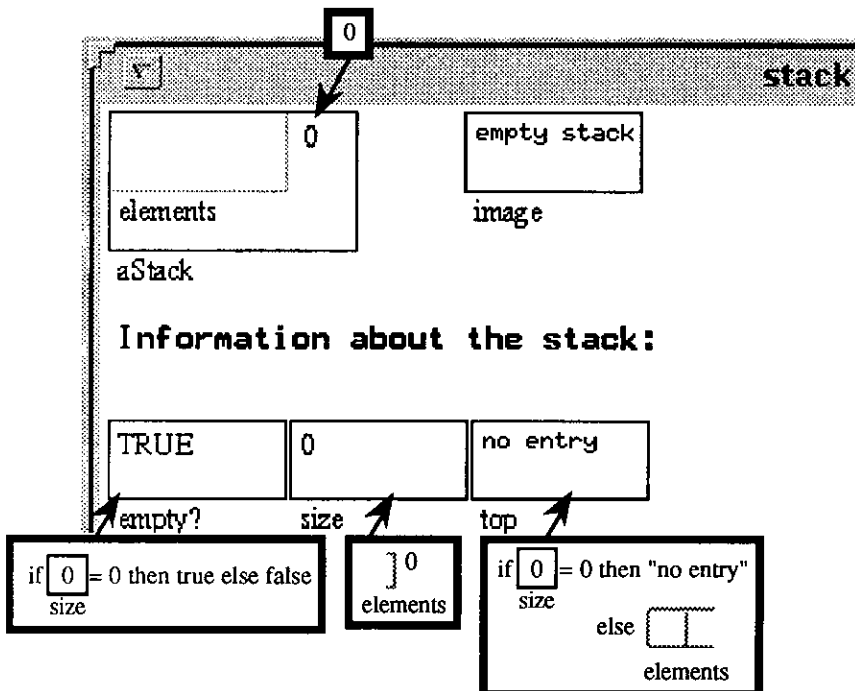


Figure 5. Top part of the stack form. The formula for the column dimension of the matrix (shown just to the right of the matrix) is the constant 0. Cell *size* references this dimension cell as its formula. The top element will always be in the first element of the matrix, as is reflected in the formula for cell *top*. The image cell's formula (not shown) will be discussed in Section 5

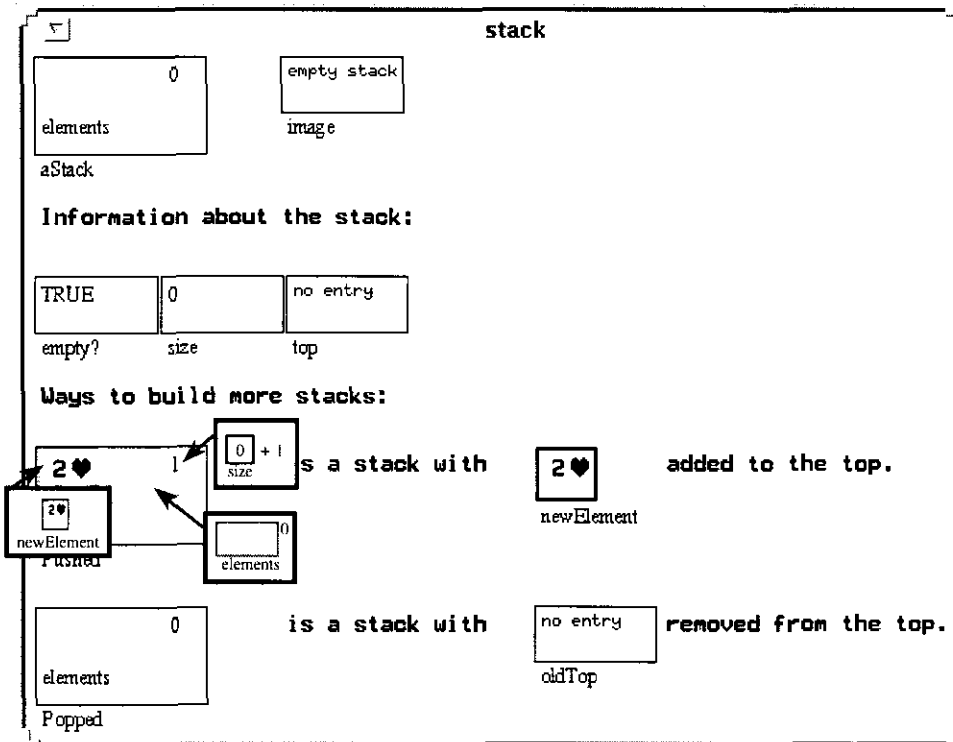


Figure 6. The first element of the matrix in the *Pushed* abstraction box is a reference to cell *newElement*. The rest of the matrix is a reference to the *elements* matrix within the *aStack* abstraction box

#### 4.2. Sample Values for Liveness, and Form Instances for Generalized Calculations

In this section we turn our attention to issues relating to the generality of user-defined VADTs, as well as issues relating to the interactivity of the programming process. In Forms/3 these issues are intertwined.

In Forms/3, sample values are used to help achieve *liveness* level 4. Tanimoto identified four levels of liveness [17]. Level 1 liveness is informative to the user but not to the machine. Level 2 is informative both to the user and to the machine, such as an executable flowchart. Level 3 liveness adds responsiveness—as soon as a visual representation is entered or changed, the machine acts upon it without requiring a separate execution phase. The examples given thus far show liveness level 3. Level 4 is continuously responsive, automatically responding whenever conditions so warrant even if no changes to the visual representation of the program occur (e.g. displaying the result of system occurrences such as clock ticks). The production planning example described later in this section shows liveness level 4.

Recall that on the form given earlier whose purpose was to compute the area of a square, the user defined *Square*'s formula to be the constant 5. This sample value allows immediate visual feedback based on a 'sample' of the area computation because the cell *Area* (based on *Square*) can calculate its value as soon as its formula is entered.

This allows the user developing the form to see the results incrementally, using them to eliminate errors. In the stack example above, the user's decision to define the number of columns in the matrix as 0 was also an example of a sample value. By including such sample values, the entry of each new formula allows the system to calculate immediately the answer based upon references to these sample values. For example, as soon as the user entered the formula for the cell *size*, the system displayed a 0, and as soon as the user entered the formula for the cell *empty?*, the system displayed TRUE. The use of sample values, besides providing immediate feedback, also means that the abstraction boxes on a VADT definition form immediately result in concrete instances of the VADT which can be viewed, referenced and used.

These concrete forms can be generalized beyond the sample values as follows. After the formulas for the objects on a form have been defined, other instances (copies) of the form can be created and modified. The underpinnings for generalization lie in the scheme for constructing these copies' unique IDs. Each copy's ID is of the form FC(DefList):RO as defined in Table 3, and is constructed so that it defines exactly how the copy differs from the original form. For example, a copy of form *AreaOfSquare* in which cell *Square*'s formula has been changed to 10 would have ID 'AreaOfSquare(Square  $\mapsto$  {10})', denoting 'AreaOfSquare in which *Square*'s formula is a reference to the temporal vector consisting of the constant 10'. Similarly, a copy of *Stack* in which *aStack*'s formula has been changed to be a reference to some other form *F*'s cell *X* (whose value is a stack) would have ID 'stack(aStack  $\mapsto$  F:X)'. Through this scheme, the system has exactly the information needed to access any instance of the original form if it already exists, and to create the instance if it does not already exist.<sup>d</sup> Of course, the user does not know about these IDs, but instead creates new instances by copying the form and re-defining formulas on it via direct manipulation.

The production planning example described in the introduction shows this ability to generalize using two copies of the stack form defined in the previous section (see Figure 7). The inventory is represented by a one-dimensional matrix, each element of which is a stack of some particular kind of inventory item. Each inventory stack is calculated based upon the previous value in its temporal vector. The other matrices give the corresponding kind of inventory item, and production and consumption rates for each kind of item in number of clock ticks before the next item is produced/consumed. These formulas are simply integer constants here, although there is no reason why they could not be arbitrarily complex. The formula shown in the figure defines a sequence which begins with the (empty) sample stack in *aStack* on form *stack*, and is followed by the result of pushing or popping inventory items as appropriate to the production and consumption rates. The operator *fbf*<sup>e</sup> means

<sup>d</sup> The system also uses aliases to redundantly describe the values being computed. This allows it to take advantage of computations which are described slightly differently, but which perform the same computation, e.g. *AreaOfSquare*(*Square*  $\mapsto$  {5}) and *AreaOfSquare*(*Square*  $\mapsto$  X), where *X* = {5}. The details and efficiency implications of this mechanism are given in Burnett and Ambler [18].

<sup>e</sup> *Fbf* is a higher-level form implemented using the primitive built-in form *Earlier* as follows. *Fbf* contains four cells: *Initially*, *Rest*, *Until* and *Result*. The first three cells are formulated as needed by the and the formula for *Result* is 'if System:initial? then *Initially* else if not earlier *Until* then *Rest*'. Following the notation of Table 3, the textual equivalent of a reference to *fbf*(*Initially*  $\mapsto$   $\alpha$ , *Rest*  $\mapsto$   $\beta$ , *Until*  $\mapsto$  NOVALUE):*Result* is ' $\alpha$  fbf  $\beta$ ', and the equivalent of a reference to *fbf*(*Initially*  $\mapsto$   $\alpha$ , *Rest*  $\mapsto$   $\beta$ , *Until*  $\mapsto$   $\chi$ ):*Result* is ' $\alpha$  fbf  $\beta$  until  $\chi$ '.



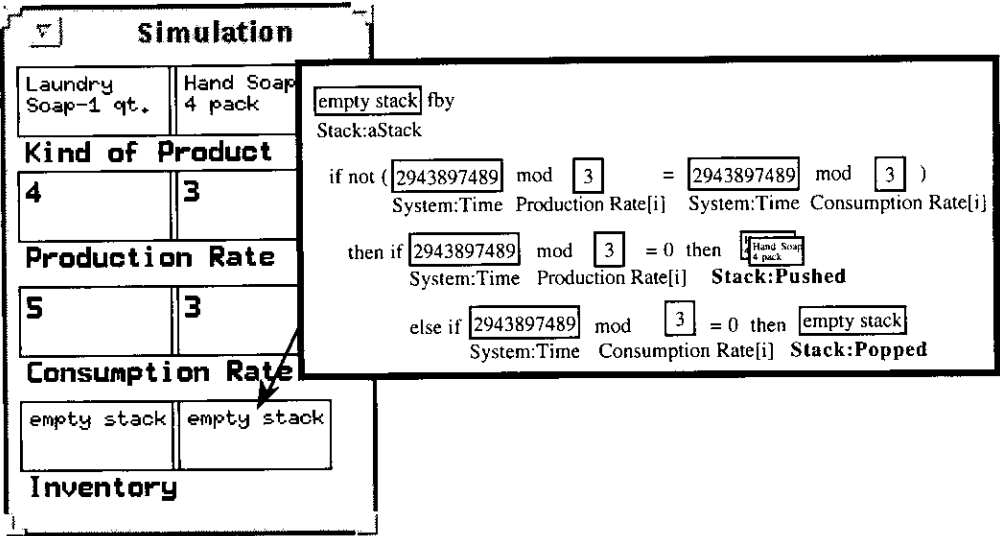


Figure 7 A production planning program. System:time is a cell on form System which contains a count of seconds. The references to cells on the original version of form stack (plain text) and a new instance (bold) are shown here with different text styles

‘followed by’ and defines a sequence of values. (This construct was inspired by Lucid [19]). The second instance of form stack referenced is simply a copy in which *aStack* references the previous element in the *Inventory* cell’s temporal vector; thus references to *Pushed* or *Popped* provide the next element needed for the *Inventory* cell’s temporal vector.

4.3. Information Hiding and Scope Rules via Visibility

It is important in any abstraction mechanism to preserve information hiding. In textual languages, information hiding is enforced via behind-the-scenes scope rules or formal declarations. Forms/3 uses a different philosophy—the accessibility/scope of an object is determined and communicated by its visibility. The rule for accessibility is

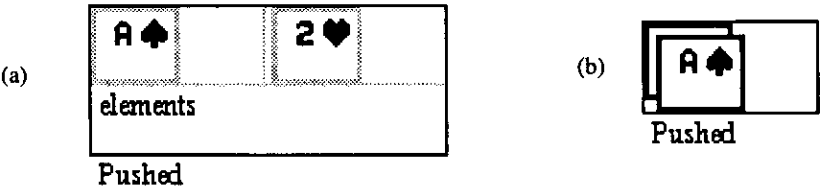


Figure 8. Hidden objects, when they are visible, can freely be referenced in formulas, and are displayed using devices such as dotted border lines or paler colors (depending on the capabilities of the workstation). (a) The abstraction box *Pushed* with its hidden contents shown. (b) The encapsulated view of the same abstraction box. The user can only reference the abstraction box as an encapsulated whole when the hidden objects inside are not visible

that if an object can be seen, it can be referenced in a formula. A referenceable object can be seen at any time unless it is *hidden*. Some objects are hidden by definition, and it is also possible for objects to be explicitly made hidden by the user. Hidden objects can be referenced only when they are visible.

There are two kinds of objects that are hidden by definition: (1) the objects inside an abstraction box; and (2) all objects on attached subforms, which are just ordinary forms which have been employed for reasons of modularity and attached to another form, similar in function to subroutines. Hidden objects are normally not visible, and thus not accessible. For example, the objects in an abstraction box are normally not visible, which means that the implementation details of how instances of VADTs are constructed are normally not accessible. However, there are three ways in which hidden objects can become visible and thus accessible:

1. Hidden objects are automatically made visible by the system when the user is in the process of entering a formula for a referenceable object on the same form. For example, when the user is defining a formula on a VADT definition form, all the hidden objects within the abstraction boxes on that form are visible.
2. Hidden objects are automatically made visible by the system when the user is defining the formula for a directly attached subform. This communicates and enables an expansion of the hidden objects' scope to include the subform. Unlike traditional block-structured languages, more deeply nested subforms do not inherit this access to hidden objects. Thus, each level of nesting corresponds to another layer of information hiding.
3. Hidden objects may be made visible by the user via specific *ad hoc* interactions, aiding debugging. To do this, the user requests, via menu selection, that the hidden objects on a specific form instance on the screen be temporarily displayed. When a hidden object becomes visible, it can be referenced by cells on other forms, but these references are valid only as long as the hidden object is visible, and only for that specific object. An example of the use of this capability will be seen in Section 5. Thus, an *ad hoc* calculation can be done flexibly using even a hidden object, but it is not generalizable to other copies of the VADT form and hence does not provide a general mechanism for circumventing information hiding. An important feature of the approach is that these *ad hoc* calculations allow extensive debugging during the normal course of interacting with the system without the need for 'debugging modes' or external tools, but do so without compromising information hiding.

#### 4.4. Polymorphism

As the stack example illustrates, the approach supports polymorphism. Thus, the same description of the abstract data type 'stack' pertains to stacks containing any type of elements, and separate abstract data type definitions, such as `stackOfCustomers`, `stackOfNumbers`, etc., are not required. But, despite this flexibility, it is possible to guarantee type safety because the approach is not based on dynamic typing. Instead, static type-checking is used to allow only type-correct formulas to be entered, preventing errors such as arithmetic operations on non-numeric values. The use of static typing does not interfere with the simplicity of the language because it does not use explicit type declarations, but rather gathers its type

information via type inference. Details of the approach to type-checking are given in Burnett [20].

## 5. Appearance and Multiple Views

### 5.1. The VADT's Appearance is Flexible

How data can be viewed occupies a central role in visual languages. In keeping with this fact, the appearance of data is an integral part of the visual abstract data type concept. In the approach described here, an object's appearance is entirely flexible and can be based on its properties. This feature, which was inspired by the dynamic icons introduced by PT [21], is realized in our system through the use of the image cell.

The image cell must be present on every VADT definition form. In the stack example presented earlier, the image cell was very much like any other cell in that it had a formula and its value was displayed on the screen. But also, when the hidden items within the abstraction box were no longer displayed, the image cell dictated the way the stack in cell *aStack* appeared on the screen. This is in fact true in the general case—whenever a stack anywhere in the system needs to be displayed, the image cell is used to determine the stack's appearance. This is done as follows.

Suppose the formula for *cellX* on some form *F* results in a stack. This would occur if *cellX*'s formula references a stack that is in some other cell, such as a reference to cell *aStack*, *Pushed* or *Popped* on an instance of form *stack*. Further suppose that *cellX* is on the screen. This means that the system must consult the image formula to decide how to display *cellX*. To accomplish this, the system internally re-uses or creates an instance of form *stack* in which cell *aStack* is a reference to *cellX*, and calculates the image cell on that form instance. The result of that calculation is used to display the contents of *cellX*. The new form instance is not itself automatically displayed, but it is possible for the user to bring it up and view it like any other form.

The evaluation mechanism affects the practicality of this approach to displaying values. In Forms/3 evaluation is done lazily. Thus, a value is only computed if it is needed. It is needed if: (1) it has not already been computed before; and (2) it is needed for display on the screen or it is needed to finish computing something else that is needed. Because of this lazy approach, only the image cell on the new copy is calculated, along with the other cells needed by the image cell's formula; the other objects on the new copy, such as *Pushed*, etc., are not calculated.

The image cell is entirely flexible, and there are no restrictions on which kinds of formulas can be used. In the figures in this paper, empty stacks have been displayed as the string 'empty stack', and non-empty stacks have been displayed by layering two images of the topmost element, via the following formula in the image cell (shown here using the textual equivalent):

```
if empty? then 'empty stack'
else compose top with top at (0 0 5 5)
```

Thus, the data definition itself can determine how an instance of it should be displayed based upon its data. Although this example is very simple, the image formula can be arbitrarily complex, including conditional compositions of images predicated on many different combinations of values.

Note what happens to the calculated image if a cell's value 'changes' over time. Of course, the declarative aspect of our model means that a cell does not change state *per se*, but its temporal vector represents a sequence of values over time. Since the formula of the image cell is normally based on the current element in that temporal vector, then a rudimentary form of animation is inherent in the approach. This is because as a displayed object's temporal vector gains new elements, the image will be recomputed based upon those new elements, causing the screen to be constantly updated.

## 5.2. Layers of Visual Abstraction for Multiple Views

Although the image of a visual abstract data type is formulated by the user in the image cell, sometimes for debugging purposes the user would like to see more details. To do this, the user can simply use instances of the VADT definition form. For example, to see a more detailed view of a stack in some cell *cellX*, the user makes a copy of the VADT definition form but defines the formula for *aStack* on the copy to be a reference to *cellX*. This automatically shows the top entry, size, etc. If the user would also like to see a display of every entry in that stack, he or she can, via menu selection, request that the hidden cells on that copy be displayed, causing the matrix elements implementing *aStack* to appear as own in Figure 9. If even more details

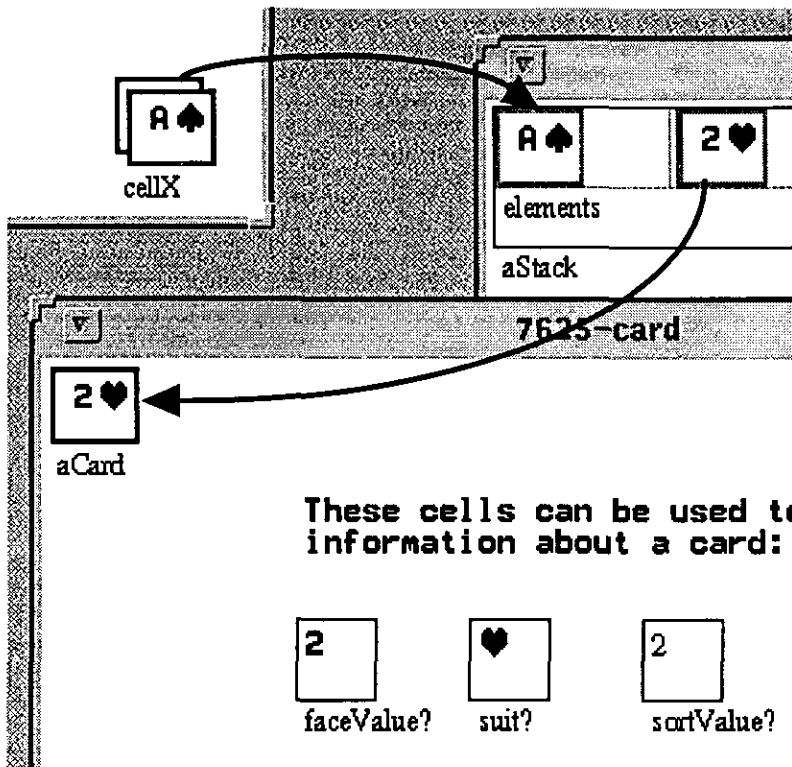
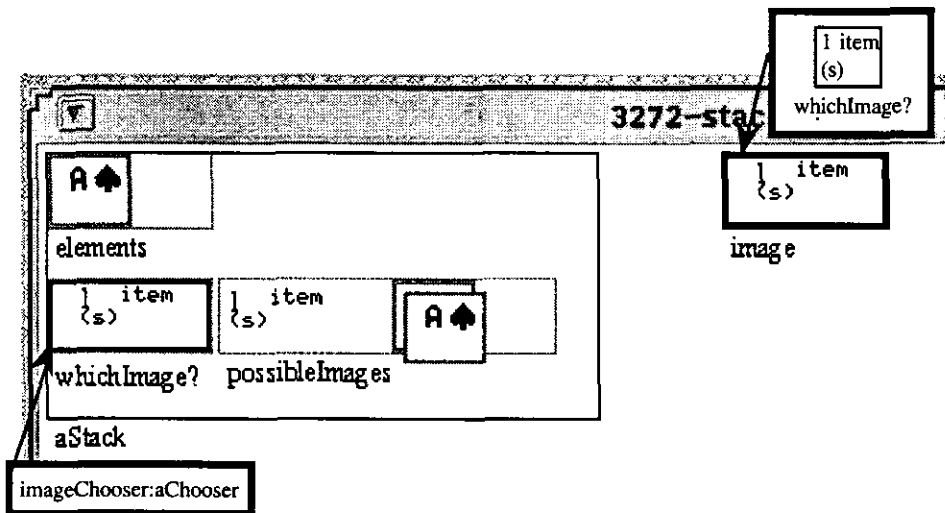


Figure 9. Viewing the details behind the abstractions. The user's interactions are shown via arrows superimposed on the screen dump

about one of the cards shown in the stack is needed, the user can repeat this process by making a copy of the VADT definition form for cards, and setting the formula for the abstraction box *aCard* to be a reference to the entry of interest in the stack. This ability to see ever more detailed views of the data is due to the characteristic that logical visibility is tightly coupled with physical visibility.

### 5.3. Multiple Images Selected Interactively

Many visual languages support interactivity in the programming environment but not in the language. In Forms/3, interactivity is supported in the language, the details of which will be given in the next section. Using this feature, the user can incorporate the notion of interactivity into a VADT's image definition, thus allowing interactive runtime selection from a variety of views of the data. For example, suppose the user wants to interactively choose between the stack image as in Figure 9 and a representation including the stack's size. To accomplish this, a stack can be defined to include more than one possible image and an instance of a VADT called an ImageChooser. An ImageChooser accepts mouse clicks from the user to toggle among an assortment of possible images. How interactive types such as the ImageChooser can be programmed by the user will be discussed in the next section; for now, we will assume that it exists. Since an ImageChooser is just a VADT like any other VADT (number, box, Boolean, customer, etc.), it can be composed into the stack's definition as shown in Figure 10.



**Figure 10.** Incorporated into the abstraction box in this version of the stack is a reference to a cell on an ImageChooser form. ImageChoosers select a new image from a matrix of choices whenever they are clicked upon, and are an example of an interactive data type. Whenever the user clicks upon the border of any instance of the stack, the displayed image toggles back and forth between the two image definitions in matrix *possibleImages*

## 6. Adding Interactivity: a Declarative Approach to Events

We have defined the notion of a VADT as the 4-tuple (components, operations, graphical representations, interactive behaviors). In previous sections we have shown how the first three elements of the tuple are supported in Forms/3, and in this section we will show how interactivity is supported as an integral part of a VADT.

The temporal dimension provides the foundations needed for a declarative approach. Events such as mouse clicks, system clock ticks and printer-out-of-paper interrupts fit naturally into the temporal model, because an event at time  $t$  can be defined as the presence of a value at time  $t$  in a temporal vector. This implies that any cell  $X$  whose formula refers to such a temporal vector will also have an element defined in its temporal vector at time  $t$ . This declarative approach is functionally equivalent to the more traditional imperative notion, expressed as:

‘whenever an event of interest to  $X$  occurs,  $X$  is activated and performs the actions needed’.

Events not tied to user interactions on the screen are recorded in a primitive form called System (Figure 11). System contains cells reflecting the state of the system and the environment. In theory, form System would contain a cell for each type of interrupt. (Our prototype supports only one such type, namely clock ticks). Cell *time* defines a temporal vector of numbers, each of which reflects the (wall) time to the nearest second, in seconds past 1 January 1900. (Recall a reference to this cell in the production planning example.) Form System also contains some additional cells for convenience. The cells on form System can be referenced in the usual way by other cells.

### 6.1. Interactive Events

Although form System could also be used for events stemming from user interaction, we wanted a less centralized approach in which the detection and response to user interaction would be controlled by the data with which that interaction took place. The approach described here generalizes upon the event-detection capabilities found in traditional special-purpose interactive data types (e.g. buttons, windows, etc.) but distinguishes between event-detection and event-response. Interactivity is supported through the use of a built-in VADT called an *event receptor* which is capable of

<b>System</b>		
2961212514	23 : 01 : 54	TRUE
time	digitalClock	initial?

Figure 11. Form System

receiving an interactive event. It can be parameterized and composed in the same manner as in the previous examples to build arbitrary visual objects capable of any kind of response. This generality and support for high-level programming is suitable not only for creating traditional point-and-click user interactions, but also for creating more complex, gestural interactions such as might be found in pen-based interfaces.

#### 6.1.1. *The Visual Abstract Data Type Event Receptor*

Conceptually, an event receptor consists of an invisible image and the sequence of events that have transpired on that event receptor. One way our system is different from others is that there is only one kind of event receptor, although it may be parameterized in ways that cause it to be sensitive to particular events. Its only purpose is to detect interactive events and to reflect the occurrence of such events in its value, not to respond to events in any way. An event receptor can then be used as ordinary data in definitions of other data types or calculations. Similarly, events themselves can be treated as ordinary data, and ordinary data can be treated as events. Because there is no distinction between data associated with an event receptor and other forms of data, event receptors can be composed with other objects to form high-level event detection/response mechanisms.

An event receptor provides event-related information and calculations, but does not have any ability actually to respond to events. This characteristic is crucial to the generality of the approach. An event receptor can be used to perform calculations without being tied to any visible object, or it can be composed with some combination of other data types, or even with just a portion of another object. This allows the user a great deal of flexibility in defining simple interactive objects such as buttons and menus, as well as more complex interactive objects with varying degrees of autonomy such as screen-savers or animated images.

Like all VADTs, an event receptor is instantiated via a VADT definition form. Using this form, the user can specify the events of interest, the shape of the event-sensitive area and the desired amount of transparency. The event receptor VADT definition form also can be used to examine information about an instance of an event receptor, such as the current event or status information derived from the sequence of recent events. The event receptor VADT definition form is shown in Figure 12.

As with any VADT, to instantiate an event receptor the user copies the event receptor definition form, provides new formulas to parameterize it and includes references to the resulting event receptor (cell *EventReceptor*) as needed in formulas elsewhere in the program. The cells *shape*, *transparent*, *Name* and the matrix *eventsOfInterest* are the parameters that can be formulated by the user, although the system automatically provides default formulas.

In the figure, cell *shape*'s formula, which results in a bitmap of a person eating ice cream, is a reference to a cell on a parameterized copy of the built-in VADT definition form *Glyph*. Glyphs are drawn using a normal bitmap editor provided by the window server. Although this parameter defines the shape of the event receptor, the event receptor's image is not visible on the screen. This parameter allows an event receptor's image to be formulated in the image cell as a portion of some other image (such as only a certain box within a larger object), or to have an event-sensitive area on the screen not tied to any visible value at all (as might be needed in a screen-saver).




1334-eventReceptor				
<b>Parameters used to create a new EventReceptor:</b>				
	FALSE	14	BUTTON-PRESS	BUTTON-RELEASE
shape	transparent	Name	eventsOfInterest	
				
EventReceptor				
<b>Event information: what just happened to EventReceptor?</b>				
0	0	NO-EVENT		
x?	y?	whatEvent?	whichButton?	which?
<b>Status information: what's the situation now with EventReceptor?</b>				
FALSE		Was there just a click?		
click?				
		The mouse is Up, Down, or Moving?		
mouse				
0	0	Position of mouse relative to object?		
x-position	y-position			

Figure 12. The user has made a copy of the built-in VADT definition form for type eventReceptor and has placed new formulas in some of the parameter cells

The *Name* parameter gives a name to the event receptor being defined, and the *eventsOfInterest* matrix allows the user to specify which types of events the event receptor should detect (button presses, button releases and mouse motion). This matrix may contain any number of cells, each of whose formula is the name of any event supported by the host (in Forms/3, any X event).

Cell *transparent* is used to specify whether the image of the event receptor is transparent or translucent. A transparent image does not obscure any other image. Thus, if two transparent event receptors overlap, both will receive an event occurring in the overlapped area. An example of the use of a transparent image is a screen saver object because both the screen saver's event receptor and the event receptors



underneath it need to receive interactive events. The screen saver needs to know whenever any interactive event occurs to any other event receptor on the screen so that it can keep track of the time of the last interaction to determine whether the screen saver should be invoked. A translucent image obscures other translucent and transparent images. Thus, if a translucent image covers another event receptor image, only the event receptor associated with the top image will receive the event. Most event receptors are translucent. Since only one translucent event receptor can logically receive an event at once, the user can control event sensitivity by formulating the composition of event receptors so that one obscures the others.

An event queue of interactive events is maintained internally by the system for each event receptor. Since cell *EventReceptor*'s internal formula includes a reference to this system-maintained event queue, and since any calculations that use an event receptor include references to it in their formulas, any event added to the queue will affect the appropriate cells. Thus, it is through the image of the event receptor that interactive events received by the system are associated with the corresponding event receptors, and it is through the system that the event receptor has access to the event queue.

#### 6.1.2. Composition with Other Visual Abstract Data Types

Since an event receptor is a VADT, it can be used to create new VADTs of arbitrary semantics. The scenario in the introduction in which objects in a window can be moved via dragging and flicking gestures demonstrates this.

An event receptor is used to program a VADT which we will call an ImageMover. Instances of this new type can then be placed directly on a window and moved, or they can be incorporated into other types (customers, stacks, etc.) to give them the same movability properties. In the same manner as previous examples, the user defines the construction of an ImageMover using cells and formulas (see Figure 13). Definition of the ImageMover involves only simple calculations, and hence most of the formulas are just references to other cells and constants.

The user starts by defining *width*, *height* and *id* cells on the ImageMover definition form. Cells *width* and *height* allow an instance of type ImageMover to know the area in which it can move. These serve a parameterization purpose, since the user can set the formulas as needed on any instance of the form. Cell *desiredImage* parameterizes how the ImageMover should appear on the screen. Its formula is a reference to a cell on a copy of the built-in VADT definition form Glyph. The *desiredImage* cell is also passed on to the event receptor to indicate its shape, as is the fact that it is to be sensitive to mouse and button movements. The cell *id* serves a parameter-like function, and requires a unique number passed in by the caller. This unique number is passed on to 'name' the corresponding event receptor which is a part of each ImageMover. We will return to this issue later.

The user then parameterizes an instance of the eventReceptor form, resulting in the particular instance of the eventReceptor form that was shown in Figure 12. Cell *shape* is specified to be a reference to *desiredImage*. Cell *Name* is a reference to ImageMover form's *id* cell. The remaining parameter-like cells (*transparent* and *eventsOfInterest*) are left with the system-provided default formulas. Using the resulting eventReceptor form, the programming of the ImageMover can be completed. Cell *er*'s formula is a reference to cell *EventReceptor* in Figure 12. Cells *changeX* and

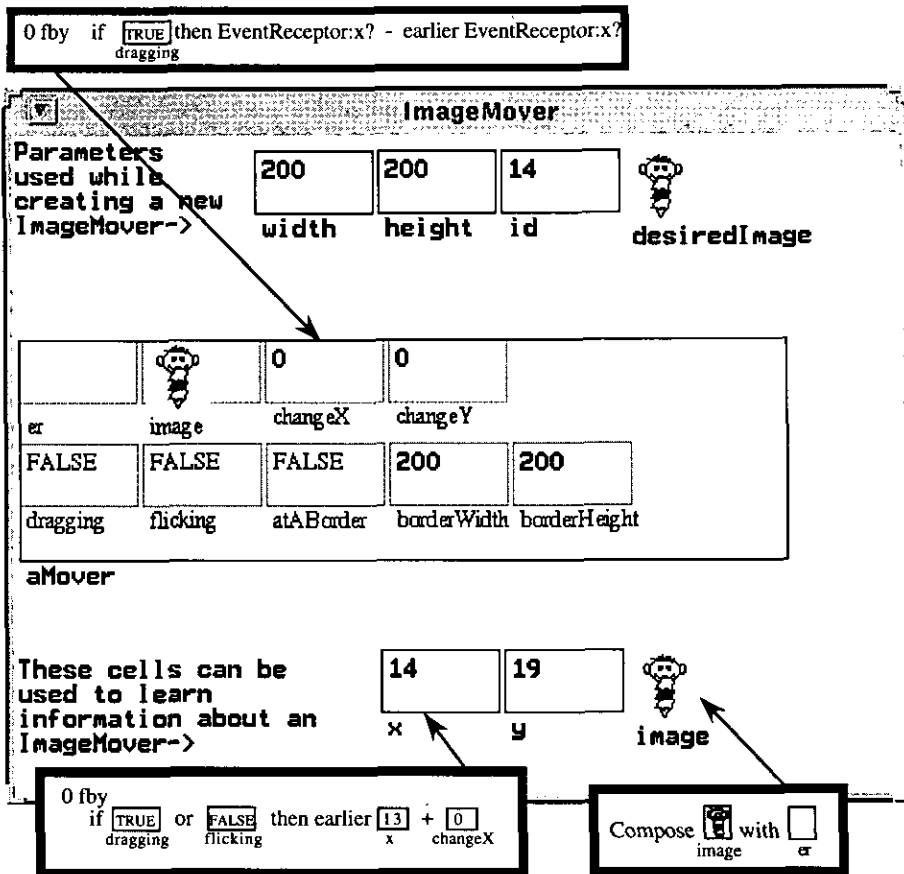


Figure 13. The definition of an ImageMover

*changeY* keep track of the directionality and rate of change when the ImageMover is being dragged so that they can be maintained during flicking. Cell *atABorder* detects when the ImageMover collides with a border. Cells *dragging* and *flicking* will be discussed in the next section.

Once the VADT ImageMover has been defined, new instances can be created as discussed previously. The behavior of these ImageMovers is affected by the formulas defined for the parameter-like cells at the top of the figure. The implementation details of an instance of an ImageMover are encapsulated in the abstraction box *aMover*. The cells at the bottom of the form provide information about an ImageMover (middle of Figure 13). Cell *image* defines the appearance of an instance of an ImageMover; thus, any cell on any form which contains an ImageMover uses the user-defined formula in cell *image* on an instance of form ImageMover to determine how it is to be displayed.

## 6.2. Higher-level Events

On the EventReceptor form, two separate events (ButtonPress and ButtonRelease) can also be thought of in combination as one event (click). This is an example of

combining two interactive events to form a new, higher-level event. Although this particular combination is provided as part of the primitive event receptor type, the user also has the power to combine any combination of events and ordinary data to define higher-level events.

Cells *dragging* and *flicking* demonstrate this ability to define higher-level events. *Dragging* becomes true if there is a Motion-Notify event on the object's event receptor while the mouse is down, and becomes false if there is a Button-Release event, as can be seen in the textual equivalent of *dragging*'s formula:

```
false fby
  if whatEvent? = Motion-Notify and mouse = Down then true
  else if whatEvent? = Button-Release then false
```

Note that there is no 'else' clause on the final 'if'. This is the textual form of the *ifThen* primitive defined in Tables 2 and 3, and defines new values *only* if the desired combination of events and computations mean that dragging is initiated or terminated. This is a higher-level event which has been defined by the user as an abstraction of a sequence of low-level interactive events and internal calculations. The representation of this event is just a Boolean true or false. These simple values serve as events because they affect all 'interested' cells just as we expect events to do. This is due to the fact that any formula that references cell *dragging* will automatically define elements in its own temporal vector whenever an element occurs in *dragging*'s temporal vector. As with all values under the lazy evaluation model, each element in these temporal vectors is defined at the appropriate position, but not actually evaluated unless it is demanded.

Similarly, *flicking* becomes true if the button is released while there is motion and *dragging*'s previous value is true. It becomes false if there is a Button-Press event on the object, or its x-y position collides with a border.

```
false fby
  if whatEvent? = Button-Release and
    ((x?<> earlier x?) or (y?<> earlier y?)) and prev dragging then true
  else if whatEvent? = Button-Press or earlier atABorder then false
```

Even 'normal' data can be combined and filtered to define higher-level events. To define the event of an even number being calculated, the user simply defines the formula of some cell *X* to be:

```
if answer/2 = truncated (answer/2)
  then 'Even'
```

Since all cells whose calculations directly depend on *X* reference it, whenever another even number is calculated for cell *answer*, cell *X* will by its definition generate a new value. This will cause all cells dependent on *X* to generate new values, and in turn all cells dependent on those cells, and so on. Since there is no 'else' in the formula, no new value will be generated in *X* and no related calculations will be generated unless *answer* is even. This uniform treatment of data as events and events as data provides the user the flexibility to write programs using an event-driven philosophy even when no events (in the traditional sense) are involved.

6.3. Moving the Objects Around on a Window

The window itself is also defined as a new VADT, as in Figure 14. Its formulas are very simple. Its image cell (bottom of the figure) is simply the composition of a box with all the objects at their computed *x*- and *y*-positions. The objects are references to instances of type ImageMover (or of other types in which ImageMovers have been composed), and the *x*- and *y*-positions are references to cells *x* and *y* on the corresponding instances of form ImageMover. Because all instances of type window are displayed via the formula in the image cell, the objects can be dragged and flicked around as desired.

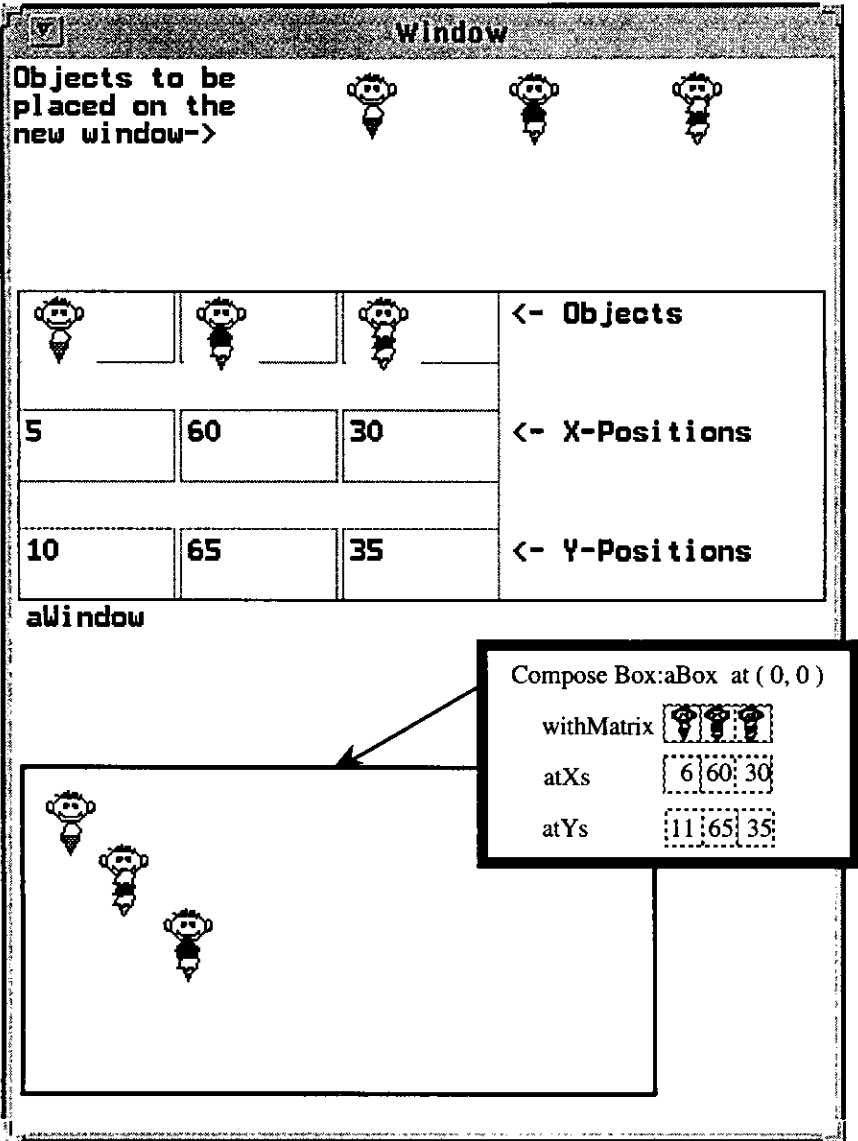


Figure 14. Three ImageMovers in a window

#### 6.4. Evaluation, Laziness and Referential Transparency

The behavior of event receptors is lazy. To see why this is true, recall that the only calculations that are demanded are those required to compute the images on the screen. Since it is impossible for the user to click on or otherwise interact with something that isn't on the screen, this evaluation rule serves interactive event processing well. An event receptor whose image is not on the screen will rest lazily in the background. When it is on the screen, it will continuously respond to user interactions within the confines of its image. All objects on the screen, following the demand-driven evaluation model, generate new demands<sup>f</sup> for their image cells whenever a change occurs because they are needed for output.

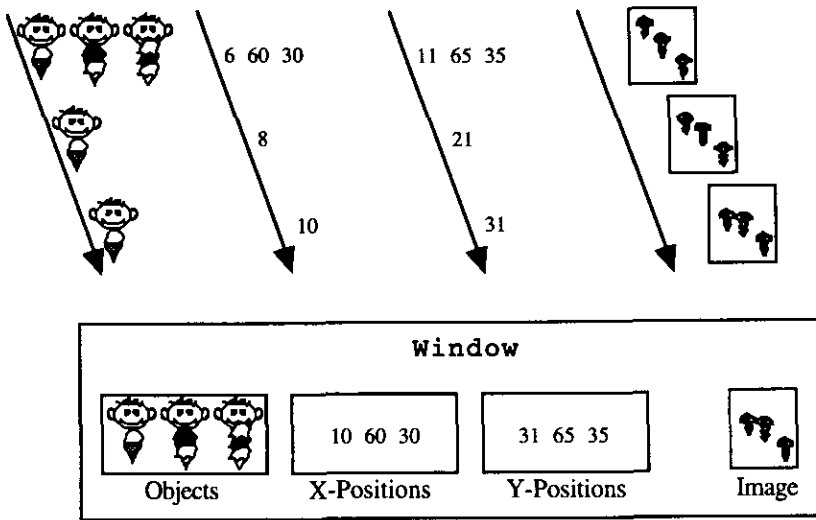
The *Name* parameter on the event receptor's definition form, in addition to its informal documentation purpose, also plays a necessary role in maintaining referential transparency. Recall that under referential transparency, if all incoming parameter-like values to a form are the same (which are reflected in the form ID), then they describe exactly the same calculation. Now, suppose the user wants to construct two different event receptors (one to be composed with a button and one to be composed with a menu), both shaped identically and both responding to only Button-Press and Button-Release events. If there are no differing formulas, the two seemingly identical requests for such an event receptor (i.e. two requests with identical IDs) would have to have identical event sequences by the principle of referential transparency. This apparent difficulty arises because there is another 'parameter' which is needed to describe the calculation completely—namely its input stream of events. From a theoretical standpoint, then, it is necessary to include the event sequence as part of the internal description of the calculation. From a practical standpoint this is not possible since the values of the events are not known in advance. The *Name* cell allows the user to identify the forthcoming sequence of screen interactions in a concrete fashion, just as filenames in traditional languages allow naming of forthcoming external values and updates. In the example above, the user might choose to formulate one event receptor's *Name* cell as 'ButtonEvents' and the other as 'MenuEvents', forcing a difference in the description of the two calculations.

#### 6.5. Evaluation of an ImageMover and Window

Suppose that an ImageMover exists and that its image is on the screen in an instance of type Window. This means that the image of an event receptor is on the screen too (since it is part of the ImageMover's image). If the user generates mouse events on the ImageMover, the event receptor will receive that information via its image.

Suppose the user's interaction so far has been a button press, motion and button release while moving the mouse. Each of these events adds an element to the temporal vector that is defined by the cell *whatEvent?*, which in turn affects the event receptor. Since the event receptor is part of the ImageMover, a cascade of new values are defined for *aMover*, its image cell, *changeX*, *changeY*, *x*, *y*, the window and the window's image cell. Figure 15 depicts the effects of such manipulations to the left ImageMover in the window over time. The window form shows the values that would be displayed on the screen as of the most recent moment in time, and the values along

<sup>f</sup>In practice, heuristics and re-use optimizations are used to minimize the number of demands and calculations, respectively.



**Figure 15.** The movement of the left ImageMover causes the generation of a new left ImageMover in the temporal vector, along with new  $x$ - and  $y$ -positions for it. Unchanged data does not cause new values to be generated in the temporal vectors

the arrow show their progression over time. Notice that the only values that are defined along the time dimension are those affected by the user's interaction with the first object. Since that object includes an event receptor (with an event history), new values for that object, as well as its  $x$ - and  $y$ -position and the final image, are defined.

Note that since an event receptor includes in its definition a mask of events to which it should respond, only relevant events will cause any new value definitions—events not of interest to that event receptor do not affect its temporal vector. Also, the user program can obscure an event receptor with some other object or cease displaying its image if the event receptor is no longer useful. When the event receptor is not visible, interaction with it cannot occur, and its temporal vector cannot acquire new elements.

### 6.6. Multiple Active Images of an Event Receptor

Now suppose that, in addition to the ImageMover's image on the screen in some *cellX*, the form defining the particular ImageMover instance is also on the screen. This provides a more detailed view of the same object, which in particular has the same event receptor as one of its parts. This is an example of a single-event receptor which has multiple active images on the screen—ImageMover:*image* and *cellX* both display an image of the same event receptor. The issue is what effects there are on the other if there is an event in one of the two images.

The answer becomes clear when we recall that an event is a value in the system-maintained event queue temporal vector for the event receptor. Because the event receptor contains a reference to that queue, any event in that event queue affects the event receptor, and resultantly the cells on form ImageMover and *cellX*. Since the event receptor's image is the conduit from the user to the event queue, and since these two event receptor images on the screen are both conduits to the same event queue,

then an event in either of the two images affects the event queue and all the objects that depend on it.

## 7. Conclusion

In this paper we have presented a declarative approach to visual data abstraction. Significant features are:

- A data type's appearance is an integral part of its definition, and is controlled by a user-defined formula which can be based upon the value of the data itself. For example, the appearance of type person could be formulated to be dependent on a person's age, hair color, gender, etc.
- A data type's interactive behavior is an integral part of its definition. In other declarative VPLs, although interactivity plays a major role in the *process* of programming, most have not fully supported interactivity in the programs *produced*. We have addressed that problem by presenting a declarative approach to interactive event handling that can be incorporated into a data type, defining the way it behaves under user interaction. The approach is consistent with the high-level, visual process of programming, it is fully declarative, and it is equally suited to non-interactive and user-defined events.
- Information hiding is supported through visibility. When a value can be made visible on the screen, it is accessible logically as well. This visual mechanism replaces the approach to information hiding found in textual languages, which generally rely on a combination of declarations and behind-the-scenes scope rules.
- Concreteness is used to provide immediate visual feedback during the programming process. By using sample values, each calculation is able to produce an answer as soon as it is defined based on sample inputs. Through the static approach to polymorphism the user can concretely create flexible VADTs without the disadvantages of dynamic typing. These concrete features provide the user with an immediate source of error feedback, allowing him or her to notice and correct simple logic errors as soon as they are entered.

The most important feature is that programming with abstract data types is no different than any other kind of programming. The approach to visual data abstraction is simple and small, requiring a minimum number of concepts to use it. Everything is accomplished using declarative formulas in cells and groups of cells. The user does not need to think about variables, declarations, sequencing, control flow, pointers, state modification, event loops, inheritance trees or hierarchical scope rules in order to program. This allows a high-level, declarative, visual approach to programming.

## Acknowledgments

We would like to thank Marla Baker, Sherry Yang, Pieter van Zee and the anonymous referees for their helpful suggestions on earlier versions of this paper, and Sherry Yang for her assistance with the figures. This work was supported in part by the National Science Foundation under grants CCR-9215030/CCR-9396134 and CCR-9308649.

## References

1. S. H. Gutfreund (1990) ManiplIcons in ThinkerToy. In: *Visual Programming Environments: Applications and Issues* (E. Glinert, ed.) IEEE Computer Society Press, Los Alamitos, California.
2. A. Borning (1986) Defining constraints graphically. In: *ACM Proceedings of CHI'86*, pp. 137–143.
3. C. E. Hughes & J. M. Moshell (1990) Action Graphics: a spreadsheet-based language for animated simulation. In: *Visual Languages and Applications* (T. Ichikawa, E. Jungert and R. Korfhage, eds) Plenum Press, New York, pp. 203–236.
4. M. Hirakawa, M. Tanaka & T. Ichikawa (1990) An iconic programming system, HI-VISUAL. *IEEE Transactions on Software Engineering* 16, 1178–1184.
5. G. Rogers (1990) The GRClass visual programming system. In: *1990 IEEE Workshop on Visual Languages*. Skokie, Illinois, pp. 48–53.
6. F. Penz (1991) Visual programming in the Object World. *Journal of Visual Languages and Computing* 2, 17–41.
7. 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.
8. TGS Systems (1989) *Prograph Reference* The Gunakara Sun Systems, Ltd., Halifax, Nova Scotia, Canada.
9. D. N. Smith (1990) The interface construction set. In: *Visual Languages and Applications* (T. Ichikawa, E. Jungert and R. Korfhage, eds) Plenum, New York.
10. P. Haeblerli (1988) ConMan: a visual programming language for interactive graphics. *Computer Graphics* 22(4), 103–111.
11. D. Ingalls, S. Wallace, Y.-Y. Chow, F. Ludolph & K. Doyle (1988) Fabrik, a visual programming environment. In: *Proceedings of OOPSLA 88*. San Diego. Also *ACM SIGPLAN Notices* 23(11), 176–190.
12. F. Ludolph, Y.-Y. Chow, D. Ingalls, S. Wallace & K. Doyle (1988) The Fabrik programming environment. In: *1988 IEEE Workshop on Visual Languages*. Pittsburgh, Pennsylvania, pp. 222–230.
13. N. Wilde & C. Lewis (1990) Spreadsheet-based interactive graphics: from prototype to tool. In: *ACM Proceedings of CHI'90*, pp. 153–159.
14. A. A. Schoberth (1990) Event handling in a demand-driven visual language preserving single assignment. Master's thesis. Department of Computer Science, University of Kansas.
15. A. L. Ambler & M. M. Burnett (1990) Visual forms of iteration that preserve single assignment. *Journal of Visual Languages and Computing* 1, 159–181.
16. M. M. Burnett (1991) Abstraction in the demand-driven, temporal assignment, visual language model. Ph.D. thesis. Department of Computer Science, University of Kansas, Lawrence, Kansas.
17. S. L. Tanimoto (1990) Towards a theory of progressive operators for live visual programming environments. In: *1990 IEEE Workshop on Visual Languages*. Skokie, Illinois, pp. 80–85.
18. M. M. Burnett & A. L. Ambler (1990) Efficiency issues in a class of visual languages. In: *1990 IEEE Workshop on Visual Languages*. Skokie, Illinois, pp. 209–214.
19. W. Wadge & E. Ashcroft (1985) *Lucid, the Dataflow Programming Language* Academic Press, London.
20. M. M. Burnett (1993) Types and type inference in a visual programming language. In: *1993 IEEE Symposium on Visual Languages*. Bergen, Norway, pp. 238–243.
21. Y.-T. Hsia & A. L. Ambler (1988) Construction and manipulation of dynamic icons. In: *1988 IEEE Workshop on Visual Languages*. Pittsburgh, Pennsylvania, pp. 78–83.
22. G. Viehstaedt & A. Ambler (1992) Visual representation and manipulation of matrices. *Journal of Visual Languages and Computing* 3, 273–289.