

A User Interface Toolkit Based on Graphical Objects and Constraints

Pedro A. Szekely and Brad A. Myers

Computer-Science Department
Carnegie-Mellon University
Pittsburgh, PA 15213

Abstract

One of the most difficult aspects of creating graphical, direct manipulation user interfaces is managing the relationships between the graphical objects on the screen and the application data structures that they represent. Coral (Constraint-based Object-oriented Relations And Language) is a new user interface toolkit under development that uses efficiently-implemented constraints to support these relationships. Using Coral, user interface designers can construct interaction techniques such as menus and scroll bars. More importantly, Coral makes it easy to construct direct-manipulation user interfaces specialized to particular applications. Unlike previous constraint-based toolkits, Coral supports defining constraints in the abstract, and then applying them to different object instances. In addition, it provides iteration constructs where lists of items (such as those used in menus) can be constrained as a group. Coral has two interfaces: a declarative interface that provides a convenient way to specify the desired constraints, and a procedural interface that will allow a graphical user interface management system (UIMS) to automatically create Coral calls.

1 Introduction

Many interactive direct manipulation user interfaces can be viewed as graphical editors. The graphical objects represent application objects: resistors and wires in a circuit simulation program, chess pieces in a chess program, chairs and tables in a furniture layout program, etc.

Graphics editing operations, such as pointing, moving, stretching, deleting, and drawing, not only change

the screen images, but also have an application-specific meaning, and must trigger changes in the data structures of the application. For example, when a resistor is moved, all the wires attached to it should be stretched to maintain the connectivity of the circuit; only some of the pieces on a chess board can be moved, and then only to certain locations; and the desks and tables in a room cannot overlap. In addition, displaying the appropriate feedback for the editing operations often requires application specific knowledge (this is called "semantic feedback"). An example of this is "gravity," where the endpoint of a wire being dragged jumps to nearby legal connection points. Therefore, for these programs, the editing operations are not unconstrained as in a simple drawing program like Apple MacDraw.

This paper describes a user interface toolkit, called Coral, currently under development as part of the Garnet Uniform Interface project [10] at Carnegie Mellon University. Coral allows the kinds of graphical interfaces discussed above to be easily implemented. In addition, Coral is able to create interaction techniques (menus, scroll bars, dialogue boxes, etc.) like those in conventional toolkits like the Macintosh Toolbox [1].

Coral facilitates the construction of graphical user interfaces through a synthesis of four techniques: object-based graphics, graphical constraints, active values, and interactors (see figure 1).

Object-based Graphics

The images displayed on the screen are composed of graphical objects such as lines, rectangles, polygons, arcs, ellipses, and strings, as well as groups of graphical objects called aggregates. All graphical objects are organized into an inheritance hierarchy. Subclassing can be used to construct new graphical objects specialized to particular applications.

Graphical Constraints

The values of attributes of graphical objects may be defined in terms of the values of attributes of other graphical objects using constraints. For instance, a resistor can be attached to other elements in a circuit

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

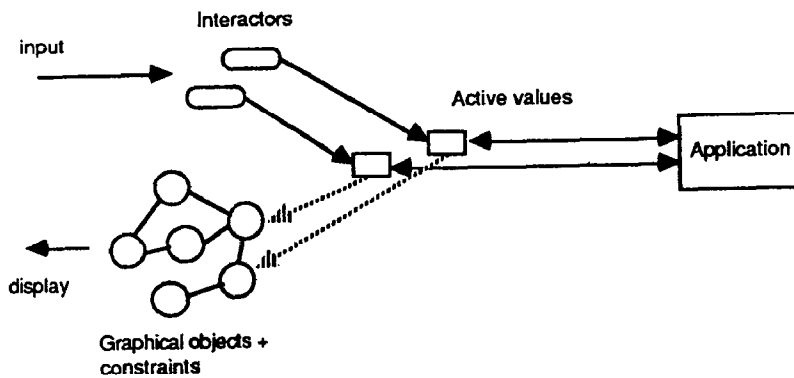


Figure 1: The architecture of Coral programs.

simulation program by defining constraints that force them to remain connected even when one of them is moved. Whenever the attributes of a graphical object change, the relevant constraints are automatically enforced, and the affected graphical objects are automatically redrawn.

An unconventional aspect of Coral's constraints is that they may be associated with a list of items. For example, all of the items in a menu may be constrained to be centered inside a rectangle, and each of the items may be constrained to be below the previous (except the first, of course).

The constraints in Coral are unidirectional, but cycles are allowed. A Coral constraint is like a formula in a spreadsheet, which computes the value of a cell based on the values of other cells. The formulas automatically recompute the values whenever any of the other cells change. Coral uses unidirectional constraints because they can be implemented more simply and efficiently than bi-directional constraints, and as Barth [3] reports, the loss of generality is surprisingly small. Cycles are useful because they allow any part of a set of connected objects to be edited, and have all the other parts updated automatically.

Active Values

An active value is a data value plus a list of objects and procedures that depend on that value. When the data value is set, the objects are informed so they can be redisplayed, and the procedures are called to notify the application program [15,13]. Active values are similar to graphical constraints, except that the constraint is from data values to graphical objects, rather than between graphical objects. Therefore, active values are sometimes called "data constraints."

Coral uses active values to communicate between the user interface and the application portion of a program (see figure 1). The application can register a procedure with an active value that is called when the active value is set, thereby informing the application about the change.

Active values provide a clean separation between user interface and application. The interaction techniques that set the active values can be changed without affecting the application portion of the program. In addition, the application does not have to update the display since that is automatically handled by the data constraints between the graphical objects and the active values. This separation facilitates the iterative refinement cycle required to produce good interfaces [7].

Interactors

In Coral, input from the mouse and keyboard is not handled by the graphical objects themselves, but by special input handling objects called *interactors*. Interactors communicate with graphical objects via active values. Since data constraints can be used to tie graphical objects to active values, interactors need only set the active values to affect the display. Active values separate the input and output portions of the user interface, thus allowing different interactors or the application program itself to drive the output part, and also further facilitating iterative development.

This paper focuses on the object-oriented aspects of Coral, and on how constraints are used to tie the objects together. A future paper will deal with the active values and interactors.

The rest of this paper is organized as follows. Section 2 describes a user interface implemented with Coral. Section 3 contrasts Coral with other systems that use graphical objects and constraints. Section 4 describes Coral's hierarchy of graphical objects. Section 5 describes the declarative language to define graphical objects and constraints, and gives examples of its use. Section 6 describes the procedural language with which the declarative language is implemented, and discusses the implementation of constraints. Finally, section 8 offers some conclusions and directions for future work.

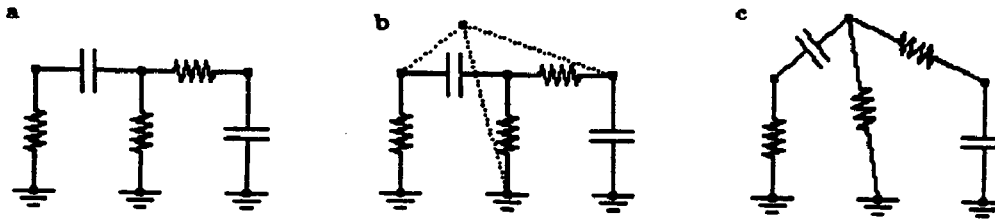


Figure 2: An interface to a circuit simulator constructed with Coral.

2 An Example

Figure 2 (part a) shows a user interface for a circuit simulator constructed with Coral. Each of the circuit elements, resistors, capacitors, ground elements are graphical objects, and the connectivity of the circuit is specified by constraints.

When the user drags a node (part b), the editor displays feedback showing the new position of the circuit elements. When the user releases the mouse button (part c), the feedback is erased, and the circuit elements are displayed at the new position. The constraints preserve the connectivity of the circuit automatically.

The feedback, represented by the dotted lines, is produced as follows: the dotted lines are graphical objects with a data constraint to an active value that represents the position of the point being dragged. Each time the mouse moves, the active value is set, and the constraints force the lines to be displayed at the new location.

3 Related work

The oldest tools to facilitate the implementation of graphical user interfaces are the graphics packages such as GKS [4], consisting of a library of routines to draw lines, polygons, curves, strings, etc. These packages are very general, allowing virtually any kind of image to be created, but are too low level, and constructing user interfaces with them involves a large programming effort. The modern graphics libraries such as the Macintosh Quickdraw package [1] or the X window manager graphics interface [14] are also general but too low level. The Coral graphical objects provide facilities similar to those of the graphics packages, but embedded in an object oriented framework similar to the one used in GROW [3] and Intermedia [9], making it easy to create new classes of graphical objects as subclasses of existing objects.

User interface toolkits such as the Macintosh toolbox [1] and the X toolkit provide an additional layer of services that facilitate the construction of graphical user interfaces using buttons, menus, scroll bars, text buffers and other such generic interaction techniques.

These tools facilitate the construction of the interface elements placed around the application window, (*e.g.* menus and scroll bars), but provide very little support to construct the application-specific interface that takes place inside the application window. The programmer has to fall back to the graphics package to implement the application-specific portion of the user interface. Coral was explicitly designed for this part of the interface, which is often the hardest to implement. Of course, Coral also supports the construction of generic interaction techniques.

Coral derives much of its power from the integration of the graphical object hierarchy and constraints. Constraints have been used in graphical application for long time. They were first used in the Sketchpad drawing program [16], which allowed lines to be constrained to have certain relationships to other lines (parallel, perpendicular, attached to, etc.). ThingLab [5] extended Sketchpad to provide a general simulation environment. The constraints in Sketchpad and ThingLab are bi-directional, which means that if two objects are attached by a constraint, then either can be modified and the other will be updated appropriately. In Coral, as explained below, the constraints are only uni-directional. Animus [8] extends ThingLab to also handle constraints about time, and therefore can support realistic animations of dynamic processes. Borning and Duisberg [6] summarize many constraint-based approaches for constructing user interfaces.

Another difference between Coral and these constraint-based systems is that Coral provides a clean separation between user interface and application. In Sketchpad and ThingLab the user interface and the application are closely tied together, where as in Coral the two parts communicate via active values. This makes Coral better suited to construct user interfaces because the separation facilitates the iteration cycle required to fine tune interfaces [17].

The GROW system [3] is more directly relevant since it uses constraints as part of a object-oriented user interface toolkit. Coral extends Grow, and the other constraint-based systems, by providing iterations and a convenient syntax for specifying the constraints.

Most UIMSs [2] are concerned with gluing together

the interaction techniques that compose a user interface, and hence are concerned mostly with dialogue management. Coral provides two mechanisms to support dialogue management, but does not provide a dialogue management tool per se. The two mechanisms are the constraints, which can be used to tie together graphical objects, and the tree of aggregate objects, which can be queried for the objects that contain a given point.

The biggest influence on Coral is the second author's Peridot UIMS [11,12,13]. Peridot is a graphical tool that allows graphical interaction techniques to be constructed by example. The user interface designer simply draws a picture of what the user interface should look like, and the system generalizes the example to create a general-purpose parameterized procedure. To do this, Peridot infers graphical and data constraints among the objects the designer creates. Unfortunately, Peridot was not built on a good underlying graphical object system, and many of the features were difficult to implement. Coral was designed to provide an appropriate foundation on which future graphical tools could be built—both to make it easier for the tools to generate code for user interfaces, and to make it easier to implement the tools themselves.

4 Graphical Objects

Coral is implemented using the CommonLisp Object System (CLOS) and runs on the X window manager [14]. A CLOS class defines each kind of graphical object. Instances of these classes represent the images displayed on the screen.

4.1 The Graphical Object Hierarchy

The classes that define the graphical objects are organized in an inheritance hierarchy as shown in figure 3.

```

Graphical-Object
  Line-GO
  Spline-GO
  Rectangle-GO
  Polygon-GO
  Polyline-GO
  Circle-GO
  Ellipse-GO
  String-GO

Aggregate-GO
  Aggregate-With-No-Overlapping-GO
  
```

Figure 3: The inheritance hierarchy of graphical objects.

The Superclass Graphical-Object

At the top of the hierarchy is the abstract class¹ **Graphical-Object** (abbreviated **GO**), which defines the properties and methods common to all graphical objects.

Graphical-Object defines the following slots (called instance variables in Smalltalk terminology):

top, left, width & height: these define the bounding box of the object.

visibility: specifies whether the object should be displayed.

draw-function: the drawing function (**xor**, **and**, **or**, **draw**, etc) with which an object is drawn on the screen.

filling: the color for the inside of the object.

outline: the pattern for the outline of the object.

window: the window in which the object is displayed.

dirty: specifies whether the object has been changed, and thus should be redisplayed.

Class **Graphical-Object** also defines the internal slots and methods needed to keep track of changes to objects so that the screen can be efficiently erased and redrawn when a display update is requested. These slots and methods are based on the bounding box of the object, and hence can be inherited by all the subclasses of **Graphical-Object**.

Graphical-Object provides accessors to read and set slots values, and default implementations of the **is-point-in-go** method, which tests whether a graphical object contains a given point, and the **erase** method, which erases objects. **Graphical-Object** does not provide an implementation of the **draw** method, since the implementation of this method depends on the class of object.

The subclasses of **Graphical-Object** define the **draw** methods. These classes inherit the slots and methods defined in class **Graphical-Object**, and provide additional ones if appropriate. For instance, **Circle-GO** provides a **draw** method as well as methods to set the radius and center of the circle. However, **Circle-GO** inherits all the bookkeeping slots and methods from **Graphical-Object**, so its implementation does not have to keep track of changes.

The **is-point-in-go**, which tests whether a graphical object contains a point is often redefined in the subclasses of **Graphical-Object** because the default method just uses the bounding box of the object, which is not appropriate for objects such as lines and polygons.

¹ Abstract classes are used as superclasses. They are never instantiated directly.

Aggregates

Class **Aggregate-GO** defines aggregate objects, which are groups of graphical objects that can be treated as a unit. For example, moving an aggregate moves every member of the aggregate, and drawing it draws every member. Since the members of an aggregate can themselves be aggregates, aggregates organize graphical objects into a tree.

Aggregates provide much more than just a convenient way to move, delete, and draw groups of graphical objects. Aggregates are the managers of groups of objects, and perform the following functions:

- Aggregates manage the redisplay of their children. When a property of a graphical object changes, its parent aggregate is informed. The aggregate determines how to update the screen: it determines how much of its bounding box to erase and which children to redisplay.
- Aggregates manage the **is-point-in-go** method for their children. To find which graphical object contains a point, the point is given to the top most aggregate in the tree of aggregates, which distributes it to the appropriate children, and then propagates it down the tree.
- The constraints that relate groups of graphical objects are defined in terms of the aggregates that contain the objects (see sections 5 and 6).

Coral provides two default classes of aggregates. The class **Aggregate-GO** handles arbitrary overlapping children. It uses simple algorithms to decide which children need to be redisplayed, and simple algorithms to implement the **is-point-in-go** method.

The other class of aggregate is **Aggregate-With-No-Overlapping-GO**. This class assumes that children do not overlap, and hence can handle redispays in a much more efficient way. New subclasses can be created for other special cases.

4.2 Extensibility

Adding new classes of objects is relatively easy, since the complex mechanism to keep track of changes for updating the screen is inherited from class **Graphical-Object**, class **Aggregate-GO** and its subclasses. However, a new class of graphical object can override this behavior if it has a more efficient way of keeping track of changes.

The inheritance hierarchy provides a simple way to tailor the behavior of graphical objects to particular applications. For example:

- In the circuit simulator program it is convenient for lines to have gravity at the end points since this facilitates connecting objects together. Gravity can be implemented by defining a subclass

of **Line-GO**, called, say, **Gravity-Line-GO**, and overriding the **is-point-in-go** method so that a point is considered inside the line even if is some distance away from the end points.

- The class **Aggregate-With-No-Overlapping-GO** is a subclass of **Aggregate-GO** that was implemented to optimize the common case where objects do not overlap. This class overrides the **erase** and **update** methods with more efficient ones that do not have to figure out how objects overlap, and after erasing an object, do not have to redisplay the objects that overlap the erased object.

4.3 Redisplay

An important aspect of Coral is that it decouples redisplay from changing the attributes of objects. A program can change many attributes of many graphical objects, and then call redisplay. This is important, because when a constrained graphical object changes, the attributes of dependent graphical objects will be recomputed, thus causing a cascade of changes. The decoupling allows the screen to be updated after all the changes to objects have been computed, thus avoiding unnecessary screen updates.

One built-in optimization of redisplay is the **Aggregate-With-No-Overlapping-GO** class discussed above. Another one is that Coral treats graphical objects drawn in **xor** specially. When an object is drawn in **xor** mode, and it is not covered by other objects, then it is erased by drawing it again using **xor** instead of by clearing the area of the screen it occupies. This optimization saves many redispays because the objects underneath the **xor** object do not have to be redisplayed.

This simple optimization is very important because it is used for the display of feedback during mouse operations, which typically overlaps many objects shown on the screen, and it has to be produced in real time (see figure 2).

5 The Declarative Interface to Coral

The declarative interface to Coral allows programmers and user interface designers to define new graphical object classes and the constraints among them using a declarative formalism. The language is designed to provide a convenient textual interface into the underlying procedural language that Coral interprets (see section 6). A special compiler expands the expressions in the declarative formalism into expressions in the procedural formalism before they are processed.

The declarative interface to Coral is illustrated in figure 4. The figure shows two check-boxes such as those used in the Macintosh user interface, and a Coral

expression that defines a graphical object class to produce them.

Draft Draft

```
(New-GO Check-Box
 (:super-classes
  Aggregate-With-No-Overlapping-GO)
 (:uses left top selected label-string)
 (:part (New-GO Box
  (:super-classes Rectangle-GO)
  :outline (:thickness 1 :pattern :solid)
  :filling (:color :white)
  :width 15
  :height 15
  :left (sv Check-Box left)
  :top (sv Check-Box top)))
 (:part (New-GO Mark
  (:super-classes Aggregate)
  :visible (sv Check-Box selected)
  (:part (New-GO Line-1
    (:super-classes Line-GO)
    :x1 (- (sv Box center-x) 2)
    :y1 (- (sv Box bottom) 3)
    :x2 (- (sv Box right) 2)
    :y2 (+ (sv Box bottom) 2)))
  (:part (New-GO Line-2
    (:super-classes Line-GO)
    :x1 (+ (sv Box left) 2)
    :y1 (+ (sv Box center-y) 1)
    :x2 (sv Line-1 x1)
    :y2 (sv Line-1 y1))))))
 (:part (New-GO Label
  (:super-classes String-GO)
  :left (+ (sv Box right) 8)
  :center-y (+ (sv Box center-y) 2)
  :string label-string)))
```

Figure 4: Instances and definition of the Check-Box graphical object.

The **Check-Box** class is defined as a subclass of **Aggregate-With-No-Overlapping-GO** since the label, the box, and the marker do not overlap.

The **:uses** clause declares that the check-box object is parameterized with respect to its top and left coordinates, the string used as a label, and a flag indicating whether the check-box is selected. The actual parameters can be either active values or arbitrary Lisp values, and must be supplied when an instance of a check-box is created. If active values are supplied, then they can be changed at run time, the constraints will be re-evaluated, and the display will be updated accordingly. For example, the parameter **selected** must be an active value so that the check-mark in the box can be turned on and off at run time.

The **:part** clauses declare the aggregate parts.

Check-Box has three parts, the box, which is the border of the check-box, the marker that shows whether the check-box is selected, and the label.

The definition of each part declares the type of graphical object, supplies values for any values that do not change at run time (e.g. the width and height of the **Box** part), and defines constraints that compute the values for the other slots. The constraints are Lisp expressions. The meaning of expressions such as **(sv Check-Box left)** is "the value of slot **left** in part **Check-Box**."

The constraints for **Line-1** define the long arm of the check-mark, and the constraints for **Line-2** define the short arm of the check-mark. Note that the check-mark is indented a few pixels inside the box. The **visible** slot of **Marker** is constrained to the value of the **selected** parameter, so when **selected** is set to **nil** the marker disappears, and when **selected** is set to **t** the marker reappears.

The **New-GO** clause not only generates the classes and constraints that define the graphical object, but also a procedure to make instances of the graphical object class. The procedure makes instances of the objects corresponding to the parts, and links them together. The parameters to this procedure are derived from the **:uses** clause.

For instance, the following program fragment creates the first instance of the check-box shown in figure 4:

```
(setf draft-selected (make-active-value nil))
(make-check-box 30 50 "draft" draft-selected)
```

The first statement creates an active value whose initial value is **nil**, and stores it in a variable called **draft-selected**. The second statement creates the check-box: 30 and 50 are the values for **top** and **left**, "draft" is the value of **label-string**, and the active value stored in **draft-selected** is the value of **selected**. Since the value of **draft-selected** is **nil**, the check-box is initially shown de-selected. When the value of **draft-selected** is set to true, by executing **(set-value draft-selected t)**, the mark is displayed.

Figure 5 illustrates the ease with which graphical object classes can be changed. The figure shows a new kind of check-box that uses a black square rather than a cross to show whether the check-box is selected. The new class, called **Check-Box-With-Black-Mark** is a subclass of **Check-Box**. It overrides the definition of **Mark** in class **Check-Box**, and inherits the definition of the other parts.

Figure 6 shows an expression that defines a graphical object consisting of a column of graphical objects. An instance of **Column** that uses a list of check-boxes is shown at the top of the figure.

The **:list** clause declares the constraints that apply to the elements of the list **list-of-GOs**. The **:for-each**, **:for-each-but-first** and **:for-first** clauses declare constraints that apply to the various

Draft

```
(New-GO Check-Box-With-Black-Mark
 (:super-classes Check-Box)
 (:part (New-GO Mark
        (:super-classes Rectangle-GO)
        :filling (:color :black)
        :visible selected
        :width (- (sv Box width) 6)
        :height (- (sv Box height) 6)
        :center (sv Box center))))
```

Figure 5: A subclass of `Check-Box` that shows the mark as a black rectangle.

elements of the list. The definition of `Column` forces each element to be left aligned, the first element to be placed 2 pixels below the top of the column, and each of the other elements to be placed 5 pixels below the previous.

- Draft
- Landscape
- Two columns
- Scale to fit

```
(New-GO Column
 (:super-classes
  Aggregate-With-No-Overlapping-GO)
 (:uses left top list-of-GOs)
 (:list list-of-GOs
  (:for-each
   :left (sv Column left))
  (:for-each-but-first
   :top (+ (sv :previous bottom) 5))
  (:for-first
   :top (+ (sv Column top) 2))))
```

Figure 6: An example of constraints ranging over a list of objects.

6 The Procedural Interface

The expressions in the declarative interface are expanded into calls to the Coral procedural interface. The procedural interface is also intended to be used by an interactive tool like *Peridot* [13], which allows user interface designers to interactively construct graphical objects and their constraints by drawing pictures.

The procedural interface is organized around two concepts: graphical objects, and *formulas*, which are

the expressions that define constraints.

A Coral formula is an arbitrary Lisp expression that returns a single value. The following is an example of a formula.

```
(+ (sv Box left) (av mouse-pos x) 3)
```

The `sv` construct is used to access the values stored in object slots. Its syntax is `(sv name &optional slot-name)`. `name` is a variable, which acquires a value at run time, and hence serves as a parameter. The value of this variable must be an instance of a graphical object. `sv` returns the value of the given slot in the graphical object bound to `name`. If `slot-name` is not provided, then the graphical object itself is returned. So, in the above example, `sv` returns the value of the slot `left` of the graphical object bound to `Box`.

The `av` construct is used to reference values in active values. Its syntax is `(av name part-name)`. Like `sv`, the `name` acquires an active value instance at run time. `av` returns the value of the given part of the active value bound to `name`. So, in the above example, `av` returns the `x` part of the active value bound to `mouse-pos`.

The Coral formula parser recognizes the calls to `sv` and `av` and builds a data structure to record the dependencies of the formula on slots and active values. So, whenever the value of a slot or active value changes, Coral can find the formulas that must be evaluated.

Formulas are used by attaching them to objects, and by assigning values to the variables used in the formula.

The `attach-formula` construct is used to attach a formula to the slot of a graphical object. Its syntax is `(attach-formula object slot-name formula)`. Once a formula is attached to a slot, Coral will use the formula to compute the value of the slot whenever any of the slots and active values in the formula change.

The `set-variable` construct is used to set the values of variables used in formulas. Its syntax is `(set-variable object name value)`. `set-variable` sets the value of the given variable in the given object. Note that the values of variables are specified on a per object basis, and not on a per formula basis. Hence, formulas are state-less, and thus a single formula to be attached to multiple graphical objects.

Specifying a new formula for a slot, or setting a new value for a variable will cause Coral to automatically evaluate the appropriate formulas, so their effects will be immediately visible on the screen.

In Coral, formulas can be defined for both classes and instances. When a formula is defined for a class, then the formula will be used in all instances of the class, unless it is overridden in specific instances. Alternatively, a formula can be applied to only a particular instance, or set of instances.

Associating the formulas with the classes is desirable because it allows formulas that apply to a whole class of objects to be conveniently defined, and allows

the formula data structures to be shared. The **Column** graphical object described in section 5 illustrates the need for the flexibility provided by Coral. The items in the column are forced to be laid out in a column by associating with each of them, except the first one, a formula that computes the top location of the item in terms of the bottom location of the previous item. The first item in the list, even though of the same class as the other items, needs a different formula: one to compute its top location in terms of a slot in the parent aggregate.

This design is more general than ThingLab [5] and GROW [3], which only allow formulas to be associated with classes, and is also more general than spreadsheets, which only allow formulas to be associated with instances.

The ability to include arbitrary Lisp code in formulas is also an important feature of Coral, since it makes it possible to specify formulas with complex behavior. The full power of Lisp is available for defining formulas. In addition, Coral can compile formulas using the Lisp compiler, thus making them efficient.

The use of variable names in formulas to refer to other objects gives Coral a degree of flexibility not found in other systems:

- By referring to other objects through names, formulas do not have explicit references to other objects, and hence can be reused by applying them to many instances.
- The values of these variables can be changed at run time, thus changing the dependencies between objects at run time. The changes to these variables go into effect immediately, causing the relevant formulas to be recomputed.

6.1 Example

Consider a drawing program such as MacDraw. When the user drags an object, the program shows an outline of the object following the mouse. The behavior of the feedback object can be implemented in Coral with a single instance of class **Rectangle-GO**, by applying the following formulas to it:

```
visible: (sv Ob-To-Drag)
width: (sv Ob-To-Drag width)
height: (sv Ob-To-Drag height)
left:
  (cond   ;; keep the feedback object inside
         ;; the drag region.
        ((< (av mouse-pos 'x) (sv Drag-Region left))
         (sv Drag-Region left))
        (<> (sv mouse-pos 'x)
         (- (sv Drag-Region right) (sv self width))
         (- (sv Drag-Region right) (sv self width)))
        (t
         (av mouse-pos 'x)))
top: ...
```

The formulas use the names **Ob-To-Drag**, whose value is the graphical object to be dragged, and **Drag-Region**, whose value is a graphical object which defines the region within which the first object must remain. **mouse-pos** is an active value that stores the position of the mouse, and which changes whenever the mouse is moved.

The mouse interactor that uses this feedback object operates as follows: when the user clicks over an object, it stores that object in the **Ob-To-Drag** variable in the feedback object, and the mouse coordinates in the **mouse-pos** active value. This causes Coral to evaluate the formulas that depend on **Ob-To-Drag** and **mouse-pos**. The formula for the **visible** slot will cause the feedback object to become visible, the formulas for **width** and **height** will cause the feedback to acquire the size of the object being dragged, and the formulas for **left** and **top** will cause the feedback to acquire the position of the mouse.

While the mouse moves, the interactor stores the mouse location in the **mouse-pos** active value. This causes Coral to evaluate the formulas for **left** and **top** causing the feedback to follow the mouse, but remain within the drag region.

When the mouse button goes up, the interactor stores **nil** in **Ob-To-Drag**, causing Coral to evaluate the formula for **visible**, making the feedback disappear.

Moving the dragged object to the new position is performed by defining formulas for it in a similar fashion.

The ability to use a single feedback object and a single set of formulas to display the feedback while dragging any graphical object, irrespective of its size, depends on the use of variables in formulas. Specifying the object to be dragged is just a matter of setting in the feedback object the value of **Ob-To-Drag**, and changing the drag region is just a matter of setting the value of **Drag-Region**.

The formulas can also be changed at run time. For instance, replacing the formula for **width** by `(+ (sv Ob-To-Drag width) 2)` will cause the feedback to be 2 pixels wider than the object. The formula will take effect immediately. No compilation or loading steps are required to see the effects of changes.

6.2 Constraint Satisfaction

The constraint satisfier is implemented by a class called **Constraint**. The Coral compiler uses multiple inheritance to include this class as a superclass of graphical objects that have constraints.

The *constraint satisfier* uses a two-pass algorithm, triggered when an active value is set:

1. Mark as undefined all slots that contain formulas that depend on the active value, and then recursively mark as undefined all slots that contain

formulas that depend on slots that are marked as undefined. This step involves no search because Coral maintains a dependency graph linking each slot to the slots that must be marked undefined each time another slot is marked undefined.

2. Evaluate the formulas for all the slots that were marked as undefined. Since formulas are unidirectional, and only one formula can be stored in each slot, no planning step is needed to decide which formulas to evaluate, and in what order to evaluate them. Thus, Coral does not need expensive planning or relaxation techniques, such as those used in Sketchpad and ThingLab, to solve the constraints.

The evaluator first updates the values of slots that depend directly on active values, and then evaluates the formulas for all undefined slots. If a formula references an undefined slot, the evaluator calls itself recursively to evaluate that slot.

Once all undefined slots are evaluated, Coral calls re-display on the topmost aggregate associated with the window, causing the screen to be updated to reflect all the changes to the graphical objects.

Coral allows the formula dependencies to contain cycles, as long as these cycles are broken by an assignment from an active value. For instance, in Coral it is legal to specify that the width of an object *a* is equal to the width of an object *b*, and vice versa, thus specifying a cycle between the width slots of *a* and *b*. If the user changes an active value that affects the width of *a*, Coral sets the width of *a* from the active value, and then evaluates the formula for the width of *b*. Since the width of *a* was already assigned a value, the cycle is broken, and Coral will not try to compute the width of *a* again. Alternatively, should the user modify the width of *b*, then the width of *a* will be recomputed.

7 Implementation Status

Coral is currently under active development. The graphical objects, the active values, the procedural language, and the compiler for the declarative language have been completed. A tool box that provides menus, scroll bars, buttons and other interaction techniques such as those in the Macintosh tool box is being implemented, and the tool to interactively specify objects and constraints is currently being designed.

8 Conclusions and Future Work

Even though the implementation of Coral is not yet complete, Coral can already be used to easily construct generic interaction techniques such as checkboxes, sliders, and menus, as shown in the figures in this paper. With the same ease, Coral can be used

to construct application-specific interaction techniques such as the ones required in a circuit simulator (connectivity maintenance and gravity).

Coral derives its power from the integrated use of graphical objects, constraints and active values:

- The graphical object hierarchy allows new graphical objects to inherit the complex mechanisms needed to efficiently update the display when objects change. This facilitates the definition of new classes of objects.

Inheritance is used to improve efficiency by creating subclasses that take advantage of special cases, and is also used to define graphical object classes specialized to particular applications.

- The Coral constraints were designed specifically for user interface construction. This involved allowing graphical constraints to be applied to lists of objects (a feature common in spreadsheets, but new in the graphical domain), and packaging constraints so that they can be reused, and so that different sets of constraints can be applied to different instances of the same class.
- Active values provide a clean way of separating of input and output portions of the user interface, and also of separating the user interface from the rest of the application.

Coral already demonstrates the tremendous potential of integrating an object-oriented graphics hierarchy, a constraint system, and active values to produce a powerful toolkit to construct graphical, direct manipulation interfaces. We are also working on the interactors, which will provide a declarative interface to specify the mapping of input events into active values. In the longer term we plan to construct a tool like Peridot that will allow user interface designers to interactively specify both the output and input behavior of graphical user interfaces. Coral should provide an excellent substrate for implementing such a tool, and in the meantime is a good tool kit for programmers to construct highly interactive user interfaces.

Acknowledgements

We wish to thank David Anderson, Ellen Borison, Richard Cohn, Dario Giuse, Bruce Horn and Richard Lerner for their comments on earlier drafts of this paper.

This research was sponsored by the Defense Advanced Research Projects Agency (DOD), ARPA Order No. 4976, Amendment 20, under contract number F33615-87-C-1499, monitored by the Avionics Laboratory, Air Force Wright Aeronautical Laboratories, Aeronautical Systems Division (AFSC), Wright-Patterson AFB, Ohio 45433-6543.

The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the US Government.

References

- [1] Apple Computer. *Inside Macintosh*. Addison-Wesley, Reading, Mass., 1985.
- [2] R. Baecker and W. Buxton. *Readings in human-computer interaction: a multidisciplinary approach*. Morgan Kaufmann, 1987.
- [3] P. Barth. An object-oriented approach to graphical interfaces. *ACM Transactions on Graphics*, 142-172, April 1986.
- [4] P. Bono, J. Encarnacao, F. Hopgood, and P. ten Hagen. GKS: the first graphics standard. *IEEE*, 9-23, July 1982.
- [5] A. Borning. The programming language aspects of ThingLab, a constraint-oriented simulation laboratory. *Transactions On Programming Languages and Systems*, 3(4):353-387, October 1981.
- [6] A. Borning and R. Duisberg. Constraint-based tools for building user interfaces. *ACM Transactions on Graphics*, 5(4):345-374, October 1986.
- [7] W. Buxton and R. Sniderman. Iteration in the design of the human-computer interface. In *Proceedings of the 13th Annual Meeting of the Human Factors Association of Canada*, pages 72-80, 1980.
- [8] R. A. Duisberg. Animated graphical interfaces using temporal constraints. In *CHI'86 Conference Proceedings*, pages 131-136, ACM, April 1986.
- [9] N. Meyrowitz. Intermedia: the architecture and construction of an object-oriented hypermedia system and applications framework. In *OOPSLA'86 Conference Proceedings*, pages 186-201, ACM, September 1986.
- [10] B. A. Myers. *The Garnet User Interface Development Environment; A Proposal*. Technical Report CMU-CS-88-153, Carnegie-Mellon University, 1988.
- [11] B. A. Myers. Creating highly-interactive and graphical user interfaces by demonstration. In *ACM Computer Graphics (Siggraph'86)*, pages 249-258, ACM, August 1986.
- [12] B. A. Myers. Creating interaction techniques by demonstration. *Computer Graphics & Applications*, 7(9):51-60, September 1987.
- [13] B. A. Myers. *Creating User Interfaces by Demonstration*. PhD thesis, University of Toronto, May 1987. (Published by Academic Press, 1988).
- [14] R. W. Scheifler and J. Gettys. The X window system. *ACM Transactions on Graphics*, 5:79-109, April 1986.
- [15] M. Stefik and D. G. Bobrow. Object-oriented programming: themes and variations. *The Artificial Intelligence Magazine*, 6(4):40-62, 1986.
- [16] I. E. Sutherland. SketchPad: a man-machine graphical communication system. In *AFIPS Spring Joint Computer Conference*, pages 329-346, 1963.
- [17] P. Szekely. *Separating the User Interface from the Functionality of Application Programs*. Ph.D. thesis CMU-CS-88-101, Carnegie-Mellon University, January 1988.