Converting an Existing User Interface to Use Constraints

Bjorn N. Freeman-Benson
University of Victoria, Department of Computer Science,
PO Box 3055, Victoria, BC, V8W 3P6, CANADA[1] bnfb@cs.uvic.ca

ABSTRACT
Constraints have long been championed as a tool for user interface construction. However, while certain constraint systems have established a user community, constraint-based user interfaces have not yet been widely adopted. The premise of this paper is that a major stumbling block to their pervasive use has been the emphasis on designing new interface toolkits rather than augmenting existing ones. The thesis of the work described in this paper is that it is possible, and practical, to convert an existing user interface written in an imperative programming language into a similar user interface implemented with constraints. This thesis is proved by example: the conversion of HotDraw into CoolDraw.

KEYWORDS
user interface toolkits, constraints, conversion, HotDraw, CoolDraw, direct manipulation

MOTIVATION
Constraints have long been championed as a tool for user interface construction. The user interface research and development community has developed constraint algorithms, interface toolkits, and even complete systems based on constraints. And, while systems like Garnet [Myers et al. 90] have established a user community, constraint-based user interfaces have not yet been widely adopted. The premise of this paper is that a major stumbling block to their pervasive use has been the emphasis on designing new interface toolkits rather than augmenting existing ones. And, as the real demonstration of a toolkit's utility is its use in full-scale applications, these new, fledging toolkits are handicapped: developers cannot afford to adopt incomplete tools while researchers cannot afford to build full-scale ones. However, if existing tools could be adapted or converted to use the researchers' ideas, these ideas could be explored in full-scale applications for significantly less cost.

The thesis of the work described in this paper is that it is possible, and practical, to convert an existing user interface written in an imperative programming language into a similar user interface implemented with constraints. This thesis is proved by example. The remainder of the paper describes both the general issues of such a conversion, and the specific details of the conversion that my students and I performed. The application we converted was HotDraw, a MacDraw-like object-oriented drawing system [Beck & Cunningham 87, Johnson 92]. HotDraw has three attributes that made it very attractive for this conversion: it has immediate graphical appeal (an important feature for a technology demonstration); the constraints could be used not only in the user interface, but also in the application itself; and HotDraw's structure is similar to other special purpose constraint-based applications we had written in the past. As a result of this conversion, both the application and the user interface have "cool" new behavior, hence the name: CoolDraw.

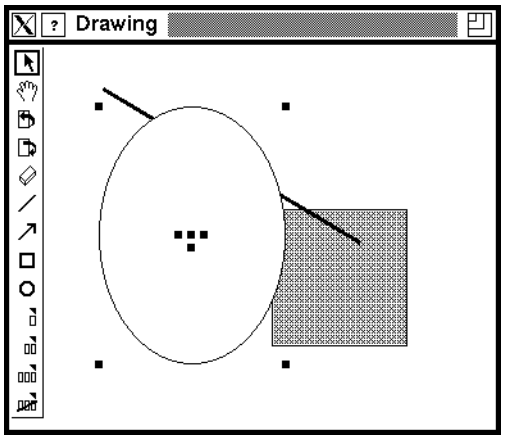

Figure 1. Screen Image

EXISTING HOTDRAW USER INTERFACE
HotDraw is a object-oriented drawing framework written in Smalltalk-80. The application consists of a Drawing object containing Figure objects, e.g., RectangleFigure, ArrowFigure, EllipseFigure, etc. The user interface consists of Tools (SelectionTool, ActionTool, …) and Handles. Tools are displayed in the palette on the left of the window, and are attached to the mouse pointer when active. They interact with figures when the mouse button is pressed. Handles are small black boxes that appear on and around a selected figure. Figure 1. "Screen Image" shows a typical window with an ellipse selected and the select-and-drag tool in use.

1. Current address: Carleton University, School of Computer Science, Herzberg Building, Ottawa, Ontario, Canada, K1S 5B6

To appear in Proceedings of UIST'93
Atlanta, Georgia, November 1993

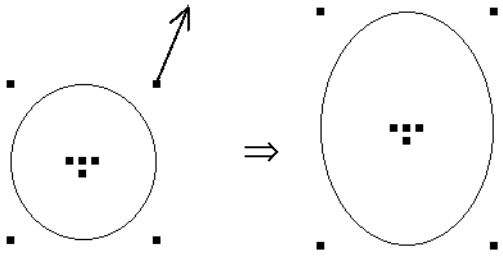

Figure 2. Using a 2D Handle

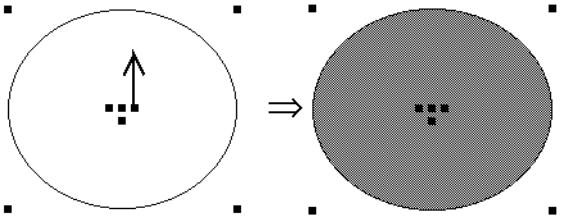

Figure 3. Using a 1D Handle

The standard HotDraw tools include: move-figure-to-front; move-figure-to-back; select-and-drag; delete/erase; create an ellipse; create an arrow; etc. These tools divide into three basic groups: figure creation, select-and-drag, and miscellaneous actions.

These tools behave as expected. For example, when the mouse button is pressed with the select-and-drag tool over a figure, that figure becomes selected and its handles appear. Each handle is a user-interface widget affiliated with one attribute of the figure. Typically there are standard two-dimensional *scale-by* handles affiliated with each of the four corners, as well as four additional handles grouped in the center of the figure. These four additional handles consist of three one-dimensional handles affiliated with the fill color, border color, and border width of the figure, plus one handle that rubberbands a connection line between two objects. When the select-and-drag tool drags a two-dimensional handle, the handle follows the mouse; thus the affiliated attribute (corner point) of the figure follows the mouse; thereby scaling the figure. For example, dragging the top right handle grows or shrinks the figure on the top and right leaving the bottom and left fixed (see Figure 2. “Using a 2D Handle”). When the selection tool drags a one-dimensional handle, the handle remains fixed while the affiliated attribute is adjusted by the vertical component of the mouse movement. In other words, the one-dimensional handle acts somewhat like the thumb of an invisible vertical scrollbar widget for the fill color, border color, or border width (see Figure 3. “Using a 1D Handle”).

COLBALTBLUE CONSTRAINT SYSTEM

The constraint system used in this conversion experiment was the ColbaltBlue incremental constraint solver, an enhancement of the SkyBlue solver [Sannella 93]. While this solver was chosen both because I was familiar with it and because it provides the necessary set of features, most of the design decisions and implementation problems outlined in the following sections are independent of the choice of constraint solver. The ColbaltBlue solver supports constraint hierarchies, incremental computation, plans, inequalities, and multiple output constraints. Note that inequalities are used in constraints such as “anywhere to the left of” whereas equalities will suffice for “two pixels to the left of”.

The Blue algorithms are structure-based rather than value-based, and thus can produce plans (partial evaluations of constraint hierarchies) which can be repeatedly reused to solve the same constraints without the expense of calling the constraint solver. While references [Borning 81] and [Maloney 91] disagree about the size of the performance increase due to plans, they do agree that an increase exists. Additionally, as described in other partial evaluation literature [Launchbury 91], the use of plans can reduce overall runtime even when such plans are used just once.

CHANGES NECESSARY

There are a number of basic changes that convert an imperative user interface, e.g., HotDraw, into a constraint-based user interface, e.g., CoolDraw. These changes can be loosely organized into three categories: those that deal with variables, those that deal with values, and those that deal with the interface.

Constrainable Variables

The most fundamental change was the conversion from normal variables to *constrainable variables*. Constrainable variables are, obviously, variables that can be constrained using the underlying constraint system. The overhead of a constrainable variable will vary with the constraint system used. However all constraint systems need some mechanism for keeping track of which constraints are attached to which variables and vice versa.

Variables are stored in memory cells, where a *memory cell* is one or more contiguous bytes used to store a single value. A memory cell storing one ASCII character would be one byte, whereas a memory cell storing an IEEE floating point number would be eight bytes. The key feature is that a memory cell can only store one value. The extra information needed by the constraint system (e.g., which constraints each variable participates in) usually requires dozens of additional cells per variable somewhere in memory.

Table 1: Constrainable Variable Options

To get the… use…	*Global Mapping*	*Objects*	*Double Storage*
value in var x	`x`	`x.value`	`x`
info about x	`info(&x)`	`x.info`	`#x`
change original application source code	no	yes	no, but recompile source code
performance	bad	reasonable	good

Three techniques for creating constrainable variables are: (listed in Table 1: "Constrainable Variable Options")

i. *Global Mapping.* Applications written in standard imperative programming languages access their variables directly through the memory cells: typical machine instructions include load value from cell at location *l* into register *r*, and store value from register *r* into cell at a location *l*. If the memory layout of the original application is not to be changed, then the per variable constraint information must be maintained using a table that maps addresses to information. The disadvantage, however, is that a relatively expensive table lookup is required each time the constraint information for a variable is accessed.

ii. *Objects.* Another technique is to store a special *constrainable variable object* in the existing memory cell, and store the variable's original value within that object. All variable reads are modified to access the value indirectly, reading either directly from the object's memory or indirectly using a message send. In return for the disadvantage of modifying all the variable reads, this technique provides reasonably efficient access to variables' constraint information.

    Note that modifying the variable reads can either be done at the source code level (as in Table 1: "Constrainable Variable Options"), or via a meta-object protocol [Kiczales et al. 91] which can alter the semantics of variable reads. However, although there are languages, such as C++, that allow the redefinition of many common operations (equality, assignment/variable writes, pointer indirection, etc.), currently only CLOS allows the redefinition of variable reads. BETA [Madsen et al. 93] comes close, but still requires variable declarations to be altered.

iii. *Double Storage.* A third technique is to modify the application implementation language's compiler and run-time system to allocate two (or more) memory cells for each variable. The first cell is for value storage, thus eliminating the need to change variable reads; and the second cell is for constraint information storage, thereby providing very efficient access to the constraint information. Unfortunately, this technique requires that the language's implementation be available and easily modifiable (e.g., to compile `#x` into a reference to the information storage for `x`).

Although source code neutral schemes are preferable, the HotDraw application was typical in that Smalltalk-80's implementation is not readily available for modification. Smalltalk-80 does have a metaclass architecture, however, for pragmatic reasons, we chose to use special constrainable variable objects with explicit messages. Our first task was to convert all variable initializations into constrainable variable object creations, and to convert all variable reads and writes into message sends. E.g., `x := y + 3` became `x value: y value + 3` (in C++ `x = y + 3` becomes `x.value = y.value + 3` or, by overriding assignment, `x = y.value + 3`). Additionally, the ColbaltBlue constraint system required that an instance variable be added to the HotDraw application (the Drawing class) to store general information about the constraint hierarchy.

Mouse Constraints

After the application code has been converted to use constrainable variables for at least those attributes that the user interface can affect, then the user interface itself can be converted to use constraints. In the imperative interface, the control flow is based on sequencing and assignment. An event loop receives and distributes events – key pressed, mouse button pressed, mouse moved, etc. – to handler routines based on the current mode and mouse position. The handler routines do rubberbanding, assign values to variables, update the graphics, etc. In HotDraw, Tools are the event distribution mechanism: the currently active tool receives and process events. For example, the select-and-drag tool handles the mouse down event with the following loop:

```
while the mouse button is down do
    if the mouse has moved then
        ask handle for its interpretation
                        of the movement
        modify the figure using that interp.
    end if
end while
```

In a pure constraint system, there is no explicit control flow: all results are computed by constraints. However, in a real system and especially in an imperative-to-constraint conversion, the overall control flow is dictated by the operating system and/or toolkit. The resulting user interface control flow is a hybrid of event driven and constraint maintenance techniques. The event loop still receives and distributes events to handler routines, however these routines are general purpose rather than specialized to each user interface. For example, all dragging operations begin by creating a medium strength equality constraint between the mouse position

and the constrainable variable affiliated with the widget under the cursor. Each time the mouse is moved, that constraint and any connected constraints are re-solved. When the mouse button is released, the equality constraint is removed.

Re-solving the constraints updates the constrained variables and thus has the same effect as assigning values to variables in the existing interface. Similarly, because the constrainable variables are aware of which figure they are a part of, solving the constraints also automatically updates the graphics.

In CoolDraw, Tools remain the event distribution mechanism. However, the event handling is simplified. For example, the select-and-drag tool's event loop is:

```
add the medium mouse equality constraint
while the mouse button is down do
    if the mouse has moved then
        re-solve the constraints
    end if
end while
remove the mouse equality constraint
```

Clearly, the difference between the HotDraw and CoolDraw event loops is that the HotDraw loop uses object specific interpretation and modification routines whereas the CoolDraw loop does not. In HotDraw, these routines implement the expected interaction behavior. For example, consider the four corner handles of a rectangle: when dragged, each handle should change the width and height of the rectangle appropriately while the opposite corner remains fixed, except when such a change would shrink the rectangle below a minimum size, at which point the opposite corner should be moved. Each of the four handles must implement different, but inherently linked, interpretations of the mouse movement such that this specification of correct behavior is maintained. This duplication of code and the fact that the constraints are only stated in the specification (if at all) has obvious negative implications for program verification and maintenance.

In the constraint-based CoolDraw interface, there are no specialized interpretation routines, and no code duplication. The minimum size restriction is enforced by an explicit constraint: $(width \geq 10) \wedge (height \geq 10)$.

Menu Selections and Assignments

Most constraint systems prohibit direct assignment to constrained variables. All modifications to constrained variables must use, or at least notify, the constraint system so that any affected constraint can be checked and possibly re-satisfied. Note that if the constraints are not automatically maintained, direct assignments can be allowed, however the constraint system may still require notification of which variables have changed, if only to ensure that solving the constraints does not undo those changes.

If constrainable variables are implemented using special constrainable variable objects or by modifying the compiler to use double-storage, then all variable writes can be trapped (either within the write method or by the compiler) and can easily be dealt with. If some other constrainable variable mechanism is used, then techniques such as the following are required:

i. *Watch Points***.** Run-time support similar to watchpoints in debuggers can be used to catch all assignments to constrained variables, or

ii. *Coding Conventions***.** A coding convention can be adopted to ensure that assigning to a constrained variable is followed by notifying the constraint system.

The CoolDraw conversion used constrainable variable objects, thus assignments were easy to detect.

After the HotDraw user interface had been converted, only two types of assignments remained: initializations and menu selections. Menu selections, such as setting the color of a line figure to gray, were implemented as the addition and then removal of an equality constraint:

```
procedure menuGray()
    add constraint color = gray
    // now the color is gray
    remove constraint color = gray
    // and it stays gray until another…
    // …constraint changes it
end
```

Constraint to Imperative Interface

Just as the imperative code must notify the constraint system when variables are assigned to, the converse is also true: the constraint system must notify the imperative code when variables change value. This is especially true for variables that interact with the external world, e.g., graphics, because one would like the graphics and the variable's values to remain synchronized automatically.

The two obvious notification schemes are:

i. *Eager.* Notification is immediate, i.e., in the midst of solving the constraints.

ii. *Lazy.* Notification is postponed, usually until after all the constraints have been solved.

The ColbaltBlue notification system uses both: it has an eager notification for each changed variable, and a lazy notification after all variables have been changed. The eager call-back is used to mark objects as needing updating, and then the lazy call-back actually does the redrawing. The eager call-back is based on which variables have changed, and thus the marking procedure uses an `owner` pointer stored with the per-variable constraint information to find the appropriate graphical object.

Note that for incremental screen updating, the notification actually needs to be done twice per object: once to erase the object from its old location and once to draw the object in its new location. In CoolDraw these actions

are similar: the erase notification marks the object's current bounding box as out-of-date, while the draw notification marks the object's new bounding box as out-of-date. After all notifications are processed, the out-of-date portions of the screen are redrawn, thereby erasing and redrawing the objects simultaneously.

Derived Values

One can distinguish between the *essential* and *derived* values of an object where the essential values are those necessary to uniquely specify the object, and the derived values are those calculated from the essential values. For example, there are four essential values for a rectangle: top, bottom, left, and right. Other values, such as width, height, or center can be derived from the original four. Of course, other choices for the essential four are possible: top, left, width, and height; or top, bottom, right, and center; etc.

In a typical application, the essential values are stored in variables and the derived values are recomputed upon demand. For example, the ellipse display routine calls a function to compute the center from the top-left and bottom-right values. Similarly, when the center of the ellipse is dragged using the select-and-drag tool, the interpretation routine computes the necessary changes to the four essential variables (top, left, bottom, and right) to affect a change in the computed center.

From the value reading point of view, there is no effective difference between reading an essential value (stored in a variable) or reading a derived value (recomputed upon demand). Thus both types of values can be considered to be stored in variables: real variables in one case and *virtual variables* in the other.

However, from the variable writing point of view, there is a large difference between the essential and the derived values in a conventional imperative language. For example, the semantics for dragging the center of a rectangle should be that of assigning new values to the virtual center variable. In practice, in imperative languages, this is implemented by computing and then assigning new values to the four essential values.

In a constraint system, the distinction between essential and derived values, and between real and virtual variables, can be erased. All variables are real and consistency constraints are used to ensure that the redundant values remain synchronized. For example, a rectangle might have eight constrainable variables: top, bottom, left, right, width, height, center x, and center y. These eight variables are attached to four consistency constraints: *top - bottom = height*, *right - left = width*, *top + bottom = center y * 2*, and *left + right = center x * 2*. The multi-directional nature of these automatically maintained constraints means that any of the eight variables can be assigned to and the other variables' values will be adjusted appropriately. The use of constraints results in true virtual variables that can be both read from and written to regardless of their essential or derived status.

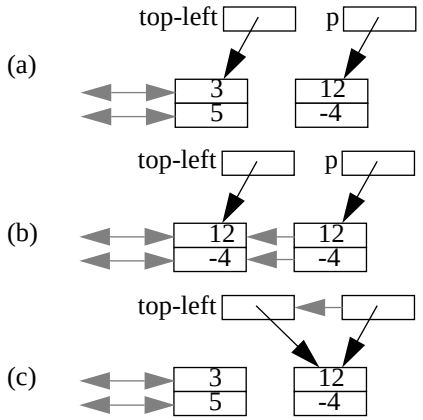

Figure 4. Updating Parts vs. Wholes

Providing all variables with the same status (readable and writable) is the mechanism which allows the constraint-based event loop to be simple and generic.

Of course, not all variables in an application need to be made constrainable. For example, in HotDraw, the PolylineFigure has a boolean variable named `closed` to indicate whether the figure is an open or a closed polygon. We decided that the user interface did not need to constrain this variable, i.e., that no user interface widget was going to affect its value. Similarly, once the number of points in a PolylineFigure is set, there is no user interface widget to add or remove points. Thus the length of the `points` array is not constrainable. However, anticipating their future use in new user interfaces, we endeavoured to make as many variables as possible available to the user interface, i.e., to convert them to constrainable variables with the necessary consistency constraints.

Variables versus Values

In an application with an imperative user interface, there is little or no difference between variables and values. For example, changing the top-left corner of a rectangle figure can be done in one of two ways:

i. *Update X and Y.* Assign new integers into the `x` and `y` instance variables of the point object contained in the `top-left` variable, e.g., in Figure 4. "Updating Parts vs. Wholes" (a) and (b).

ii. *New Point.* Assign an entirely new point object into the `top-left` variable, e.g., Figure 4. "Updating Parts vs. Wholes" (a) and (c).

Either method changes the value of the top-left corner as seen from outside the rectangle. However, when a constraint-based user interface is used, the individual x and y variables of the top-left point object can be otherwise constrained, and thus there is a substantial difference between updating x and y, and creating an entirely new

point object. Updating the values will correctly trigger the constraint solver to resatisfy all the constraints based on the new values. Replacing the point object will create two new, *unconstrained*, variables for its x and y components, as well as removing the existing x and y constrained variables from the structure of the rectangle. (See Figure 4. “Updating Parts vs. Wholes” (c).) Thus, not only will the new values not propagate through the constraints, any other changes which would have propagated through the constraints to update the position of the top-left corner will still propagate, but will not update the position — instead they will update the now unreachable variables that were part of the former top-left point object.

Thus, once a part-whole object structure has been created and constrained, either it should not be restructured, or special care must be taken to reset the affected constraints. Note that this problem exists even when the underlying constraint solver supports pointer variables, e.g. [Hudson 93, Hill 93].

We encountered this problem during the conversion of the HotDraw figures that were subclasses of PolylineFigure. These figures are defined by an internal array of point objects. Whenever the figure was altered, moved, stretched, etc. this array was replaced by a new array of new point objects containing the new values. In CoolDraw, the array of point objects is created once when the figure is initialized, and thereafter only the x and y instance variables of the individual points are updated. Thus CoolDraw replaces:

```
points := new_points
```

with:

```
for i := 1 to size(points)
    points[i].x := new_points[i].x
    points[i].y := new_points[i].y
end
```

A similar problem, also an artifact of the PolylineFigure, is duplicate values (which are a trivial case of derived values). In HotDraw, the RectangleFigure is implemented as a subclass of PolylineFigure, i.e., a four point closed polyline with the implicit constraints that the lines are horizontal and vertical. However, four points contain eight values whereas only four values are necessary to specify a rectangle.

One obvious solution would be to actually create the four consistency constraints: *topLeft x = bottomLeft x, topLeft y = topRight y*, etc. However, as [Borning 81] noted, identity is a stronger and more efficiently computable condition than equality, so CoolDraw uses four constrainable variables (top, bottom, left, and right) to create the four corners — each variable is shared by two of the points.

Splitting Constraints

User interfaces to object-oriented applications should present a wide variety of abstractions to the user. In the case of CoolDraw, this means that the user should be able to manipulate at least the following types of data: a group of figures, a single figure, one corner of a figure, or one dimension of one corner of a figure. Providing this diversity is challenging in a constraint-based user interface because it requires that the constraint solver handle constraints at different levels of the part-whole hierarchy. For example, the solver must be able to handle a constraint on a point object in conjunction with another constraint on just the x coordinate of that point.

The simplest solution to this problem is to lower all constraints to the same level of abstraction, e.g., if some constraints are on points and some are on numbers, then the former should be transformed into constraints on numbers. This transformation is called *splitting* a constraint [Freeman-Benson & Borning 92]. Constraints on points are split into two constraints, one on each of their x and y components. Similarly, a constraint on an entire figure, such as the constraint created when an entire figured is dragged, are split into four constraints for the four corners, which are then split into eight constraints on the eight separate x and y components.

Likewise, the mouse position consists of two numbers (the x and y coordinates). However, when the mouse manipulates one of the special one-dimensional handles (see Figure 3. “Using a 1D Handle”), only one of the two numbers is used. Thus, not only is the mouse-position-equals-variable constraint split into separate x and y constraints, but one of the two is discarded.

Non-Numeric Values

Most of the modifiable variables in a user interface hold numeric values, typically integers. Because the mouse position is a pair of integers, simple equality constraints can be used to connect the mouse position to these variables. However, not all such variables are numeric: other common data types include booleans, colors, and menu items. Thus the user interface toolkit must provide some mechanism to translate between the numeric values of the mouse position and the non-numeric domains of the affected variables.

In a constraint-based system, there are two obvious strategies:

i. *Constraints***.** Use special constraints that relate the two domains, or

ii. *Conversion***.** Convert the application to use numeric variables exclusively.

The use of special constraints is similar to the mechanism used in existing user interfaces: the mouse movement or mouse position is translated into an instance of the appropriate data type. For example, the menu interaction routines on the Macintosh translate the mouse position into a data structure indicating a menu and an item in that menu.

The advantage of using special translation constraints is that no further modification to the existing application is required. For example, in HotDraw, the special handle

for changing the fill color of a rectangle implements an invisible vertical scrollbar (see Figure 3. “Using a 1D Handle”). The interpretation routine for that handle consists of:

```
if the mouse has moved up 10 units then
    get the current fill color
    find the color table index of that color
    increment the index
    get the color from the color table at
                              the new index
    return the new color
end
```

The first version of the CoolDraw user interface used a special constraint that emulated this interpretation routine and thus no change to the rest of the application was necessary.[2] The disadvantage, however, is that the mouse down event handler must have special logic for the creation of these special constraints.

The second technique (converting the application), simplifies the user interface constraints by moving the burden of translating the numeric values into complex objects to the application code. For example, in the released version of the CoolDraw user interface, numeric values are used for the fill color, and the `display` routine has been modified to index the color table. The disadvantages are obvious: the application was modified, and the resulting code runs slightly slower. The advantages, however, are that the mouse down event handler is generic and, specific to CoolDraw, that it was possible to extend the application itself with constraints. For example, the user can constrain the fill color to change in concert with the figure’s width using a simple numeric equality constraint.

PROBLEMS ENCOUNTERED

After the basic design decisions had been made, the conversion from HotDraw to CoolDraw was fairly straightforward: variables were made constrainable; handles and tools were altered; consistency constraints were created for derived values; non-numeric variables were eliminated; etc. However, the conversion was not without problems, and a number of these problems uncovered further design options.

Origin Independent Positions

An interesting interaction between the CoolDraw application and its user interface occurred because of *anchor* constraints. An anchor constraint fixes the position of the constrained variable on the canvas, and causes figures to be *position dependent*. For example, if a rectangle figure is dragged to another position while one corner is anchored, the rectangle will stretch or shrink so as to satisfy both the anchor and the mouse movement constraints. There was no equivalent of anchors in the original HotDraw application and thus, unfortunately, there were at least two places in the HotDraw user interface that assumed *position independent* figures: scrolling the canvas and cut/copy/paste.

The canvas scrolling tool uses a hand icon to push the drawing around behind the window á la MacPaint. HotDraw had implemented this behavior by dragging all of the figures in the drawing at once. With position independent figures, this has the same visual effect as moving the canvas. With position dependent figures, the effect varies with the presence or absence of anchors: unanchored figures move, while anchored figures change shape. This problem was fixed in the CoolDraw user interface by moving the canvas rather than the figures.

Cut/copy/paste in HotDraw had a similar problem: when a group of figures was cut or copied to the paste buffer, they were translated such that the bottom left corner of their collective bounding box was at the origin. This behavior could not be implemented in the presence of position dependent figures, and thus objects are pasted at the same locations they were cut or copied from. In fact, as with scrolling, one could argue that the CoolDraw behavior is a “better” design than that of the original HotDraw.

Additionally, cut/copy/paste had to be further modified to deal with the semantic content of CoolDraw drawings (i.e., both the figures and the constraints between the figures have to be cut/copy/pasted).

Numeric Replacements for Non-Numeric Variables

One of the design issues regarding the replacement of non-numeric values (see section “Variables versus Values”) is the mapping of the infinite and continuous numeric domain to the (usually) finite and discrete non-numeric one. The exact mapping will be application specific, but all systems will have to deal with the issue of limits. For example, assume that the fill color of a figure is a number indexing a color table with three entries. Consider the two possible mappings from numbers to colors in Table 2: “Two Possible Mappings”.

$< 1 \rightarrow \perp$	
$1 \rightarrow$ black	$-\infty \ldots 1 \rightarrow$ black
$2 \rightarrow$ gray	$2 \rightarrow$ gray
$3 \rightarrow$ white	$3 \ldots \infty \rightarrow$ white
$> 3 \rightarrow \perp$	

Table 2: Two Possible Mappings

The mapping on the left requires that the interface include additional consistency constraints to prevent the user from requesting an illegal value. The mapping on the right, however, allows vastly out-of-range numbers. For example, usually when the user moves the mouse down by 10 pixels, the color lightens one step. However. if the variable contained 900 (which displays as white) and the user moved the mouse 50 times as far (500 pix-

2. Note that this *translation constraints* technique requires a constraint solver that can handle non-numeric constraints. All of the Blue derivative algorithms have this capability.

els) the value would become 850 and the fill color would unexpectedly still be white!

RELATED WORK

There are numerous approaches to providing new user interfaces and user interface toolkits. There are many outstanding projects exploring constraint-based user interfaces (e.g., [Hudson 90, Myers et al. 90, Olsen 90] and many of the chapters of [Myers 92]), however they all require that at least the user interface, and usually the application as well, be implemented in a specific language or within a specialized system. Other systems (e.g., [Telles 90]) have the goal of provide a new user interface wrapper for an existing, or legacy, system.

The goal of the CoolDraw project was the exploration of a third alternative: the conversion of an existing imperative user interface. Many, but not all, of the design decisions and implementation problems discussed in this paper occur in a "from-scratch" constraint-based user interface system as well, and thus no user interface development effort would be complete without carefully considering at least these three alternatives: an entirely new system, a new wrapper, or a conversion.

PERFORMANCE OF MAN AND MACHINE

From the user's point of view, the converted CoolDraw application has the same performance and the same user interface as the original HotDraw application. From the programmer's point of view, the user interface event loop code has been greatly simplified, but introducing new figures has become more a bit complex as the designer must consider the semantic behavior of the object vis a vis derived values, non-numeric value replacements, etc.

The conversion itself was fairly straight forward, in our case consuming approximately two person-weeks after the design decisions outlined in this paper had been made. Note that this conversion time assumes the pre-existence of a solid, well documented, constraint solver.

SUMMARY

The CoolDraw application is a superset of the HotDraw application. Not only are the basic functions of HotDraw duplicated in CoolDraw, but the use of constraints both fixed flaws and introduced new functionality.

Two flaws in the original HotDraw user interface were that the one-dimensional handles were based on mouse motion rather than position, and that groups of figures could not be scaled. Both were easily fixed using constraints. The one-dimensional handle problem was due to the fact that the mouse down code asks the handle for its interpretation of mouse movement rather than mouse position. Because the mouse is "down-shifted" by a factor of ten to provide finer control (i.e., the mouse movement is divided by ten), the user had to move the mouse at least ten units in single event before the handle would recognize it as motion. Thus, if the mouse was moved slowly (less than ten pixels per clock tick), no change occurred, even when the mouse was eventually moved a large distance.

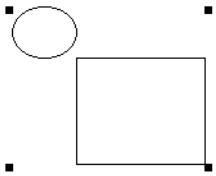

Figure 5. Group Relative Scaling

Constraint systems, being declarative, are inherently based on values rather than changes in value over time. Thus the CoolDraw user interface has the correct behavior regardless of mouse speed.

The HotDraw user interface mechanism of figure-specific interpretation and modification routines restricts its interactions to a single figure at a time. The CoolDraw mechanism of automatically maintained, multi-directional constraints allows the user interface to utilize additional "interaction" constraints without changing its general-purpose mouse equality constraint activity loop. For example, to ensure that the ellipse in Figure 5. "Group Relative Scaling" is half the size of the rectangle, two group interaction constraints are used: *ell.topLeft + rect.botRght = rect.topLeft*2* and *rect.topLeft = ell.botRght*.

A similar benefit of using constraints is the trivial replacement of the two one-dimensional border handles (color and width) with a single two-dimensional handle with the vertical component controlling width and the horizontal component controlling color.

Because CoolDraw has only just been released to the public, no conclusions can be drawn about whether this conversion technique has succeeded in promoting the use of constraints in user interfaces — more time is necessary. However, the fundamental question about the feasibility of conversion as an approach to introducing constraints has been answered positively.

Acknowledgments

Conversion assistance was provided by Ian Perrigo. The HotDraw system was implemented by Pat McCaughry. Thanks to Kirsten Freeman-Benson and Marc Stadelmann for proof reading, and to the UIST referees for pointing out sections that needed more exposition. This project was generously supported by Object Technology International, and by the National Science and Engineering Research Council under grant OGP0121431.

REFERENCES

[Beck & Cunningham 87] Kent Beck and Ward Cunningham. Semantic Drawing with HotDraw. Technical Report CR-87-34, Tektronix Computer Research Laboratory, April 1987.

[Borning 81] Alan Borning. The Programming Language Aspects of ThingLab, A Constraint-Oriented Simulation Laboratory. *ACM Transactions on Programming Languages and Systems*, 3(4):353–387, October 1981.

[Freeman-Benson & Borning 92] Bjorn Freeman-Benson and Alan Borning. Integrating Constraints with an Object-Oriented Language. *Proceedings of the 1992 European Conference on Object-Oriented Programming*, pages 268–286, Utrecht, The Netherlands, June 1992.

[Hill 93] Ralph Hill. The Rendezvous Constraint Maintenance System for Multi-User Applications. To appear, *Proceedings of the ACM Symposium on User Interface Software and Technology*, November 1993.

[Hudson 90] Scott E. Hudson. Adaptive Semantic Snapping — A Technique for Semantic Feedback at the Lexical Level. *Proceedings of the 1990 ACM SIGCHI Conference*, pages 65–70, Seattle, April 1990.

[Hudson 93] Scott Hudson. *EVAL/vite User's Guide (v1.0)*. Georgia Institute of Technology technical report GIT-GVU-93-xx, 1993.

[Johnson 92] Ralph E. Johnson. Documenting Frameworks using Patterns. *Proceedings of the 1992 ACM Conference on Object Oriented Programming, Systems, and Applications*, pages 63–76, Vancouver, BC, October 1992.

[Kiczales et al. 91] Gregor J. Kiczales, Jim des Rivières, and Daniel G. Bobrow. *The Art of the Metaobject Protocol*, MIT Press, 1991.

[Launchbury 91] John Launchbury. *Projection factorisation in partial evaluation*. Cambridge University Press, Cambridge, 1991.

[Madsen et al. 93] Ole Lehrmann Madsen, Birger Møller-Pedersen, and Kristen Nygaard. *Object-Oriented Programming in the BETA Programming Language*. ACM Press/Addison-Wesley, New York, 1993.

[Maloney 91] John Maloney. Using Constraints for User Interface Construction. Ph.D. thesis, Department of Computer Science and Engineering, University of Washington, August 1991.

[Myers et al. 90] Brad A. Myers, Dario Guise, Roger B. Dannenberg, Brad Vander Zanden, David Kosbie, Philippe Marchal, and Ed Pervin. Comprehensive Support for Graphics, Highly-Interactive User Interfaces: The Garnet User Interface Development Environment. *IEEE Computer*, 23(11):71–85, November 1990.

[Myers 92] Brad Myers, Editor, *Languages for Developing User Interfaces*. Jones and Bartlett, Boston, 1992.

[Olsen 90] Dan R. Olsen, Jr. Creating Interactive Techniques by Symbolically Solving Geometric Constraints. *Proceedings of the ACM SIGGRAPH Symposium on User Interface Software and Technology*, pages 102–107, Snowbird, Utah, October 1990.

[Sannella 93] Michael Sannella. The SkyBlue Constraint Solver. Technical Report 92-07-02, Department of Computer Science and Engineering, University of Washington, February 1993.

[Telles 90] Marcy Telles. Updating an Older Interface. *Proceedings of the 1990 ACM SIGCHI Conference*, pages 243–247, Seattle, April 1990.