Graphical Constraints in CoDraw

Mark D. Gross
University of Colorado; Boulder, Colorado 80309-0314
mdg@cs.colorado.edu

Abstract

Constraint based draw programs require users to understand and manage relationships between drawing elements. By establishing constraint relationships among elements the user effectively programs the drawing's behavior. This programming task requires a more sophisticated visual interface than conventional draw programs provide. Users must have available — in a convenient format — information about the structure of the constraints that determine the drawing's interactive edit behavior. This format must support editing and debugging. CoDraw is a constraint based drawing program that can be interactively extended by its users. This paper describes the CoDraw program and its programming interface.

1 Introduction

This paper describes CoDraw, an experimental constraint-based drawing program, and Co, the constraint programming environment in which CoDraw is embedded. With CoDraw a user programs the edit behavior of the drawing by applying constraints to drawing elements. The program enforces and maintains these relationships. Creating, debugging, and editing the constraints that determine drawing behavior requires a more complex and sophisticated user interface than a conventional drawing program. In addition to elements in the drawing, the program must also make visible the relationships that the user has specified. For example, the program must distinguish visually between two objects that have been constrained so that their top edges align and two objects that merely happen to have been placed that way.

In Sketchpad [14], one of the first interactive graphical interfaces, user-defined constraints governed the behavior of elements in a design drawing. Sketchpad inspired a generation of constraint-based graphics and programming environments, including ThingLab [2], Juno [12], and others [1,16] and research on constraint satisfaction and management and language design [7,13,10,9,6].

Two approaches are commonly used for constructing and debugging constraint programs interactively. The first approach, following ideas in Sketchpad, uses a network editor to construct, view, and edit constraint programs. Several systems have employed this approach including ThingLab and its successors [3]. The second approach uses a spreadsheet style interface to program the constraints for objects in the domain. Systems that follow this approach include FINANZ [5], NoPumpG [15], and C32 [11].

In many of these environments, constraints are one-way assignments, pre-defining the dependency structure of the network. Co's (and CoDraw's) constraints are two-way constraints, in which dependencies (between fixed, or anchored variables and derived ones) are not preprogrammed but can be managed by the user. This adds to the expressive power of the system but it places additional demands on the interface. CoDraw provides spreadsheet cards to define objects and program their constraints and it also provides other editable views of the data structures that govern behavior.

2 CoDraw

CoDraw is a drawing program that uses constraint techniques to enforce and maintain simple spatial relationships. In addition to built-in geometric constraints (e.g. alignments, offsets, tangency) users can extend CoDraw's repertoire by defining new constraint types. These extensions are constructed interactively using spreadsheet cards without leaving CoDraw. The user may also define a visual representation for a new constraint, which can be added to the menus of icons and used to display instances of the constraint in the graphical work area.

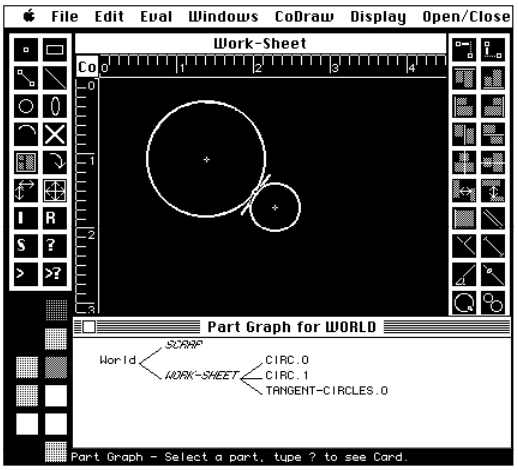

Fig. 1. CoDraw's worksheet and tool palettes.

CoDraw looks like a typical Macintosh draw application (figure 1). It provides a drawing work-sheet with two tool palettes and a color palette. A documentation line at the bottom of the screen offers brief commentary, help, and error messages. Graphs and spreadsheet cards provide additional and more detailed views of the drawing and its objects. For example, the Part Graph in figure 1 dynamically displays the assembly structure of the drawing. These additional views help users to construct and debug constraint-based drawings and models.

Using the command palette on the left the user can enter simple geometric elements: points, line-segments, lines, rectangles, circles, and ovals. The command palette also contains commands for operating on graphic elements. The relations palette on the right contains spatial constraints that a user can apply to graphic objects in the work area. With these the user constrains the behavior of drawing elements. For example, after selecting two circles the user applies the tangent-circles constraint (bottom right icon). CoDraw adjusts the circles to be tangent and keeps them tangent as the user continues to edit the drawing (figure 1).

Constraints in CoDraw are first class objects just as the drawing elements they relate and therefore they are also presented graphically to the user. When a user establishes a constraint relating two objects, the constraint is displayed in the Work Sheet. In this example, CoDraw displays the tangent-circles constraint as a short red line segment at the point of tangency. Notice also in figure 1 that the Part Graph window displays three objects as parts of the Work Sheet — the two circles and the tangent-circles constraint.

As with any other object in the Work Sheet the user can select a constraint and delete it. This removes the constraint from the network of relations that governs the drawing's behavior. For example, deleting the tangent-circles constraint from the drawing in figure 1 would permit the user to move or resize the circles so that they are no longer tangent.

The built-in relations palette contains an assortment of alignments, adjacencies, and offsets. For example, the top-align relation establishes a constraint that keeps two objects' top edges aligned. Other relations on the palette include parallel-lines, fixed-length-segment, point-on-line, and point-on-circle. CoDraw grays the icons of relations that do not apply to the currently selected elements: for example the tangent-circles constraint cannot be applied to two rectangles. Icons in the relations palette can also be used to get documentation about the constraint or to display a spreadsheet card containing the constraint definition.

2.1 Dependency and conflict resolution

When the user asserts a new constraint, such as the tangent-circles constraint above, CoDraw detects a conflict. The new constraint requires the circles to be tangent, but they are not. To satisfy the new constraint, CoDraw selects a constraint or constraints from one or both circles and relaxes it. It can either relax a constraint on a circle's center position or on its radius. In the case of tangent circles, CoDraw resolves a conflict by relaxing the radius of the second circle selected. This circle will grow or shrink to satisfy the tangency constraint.

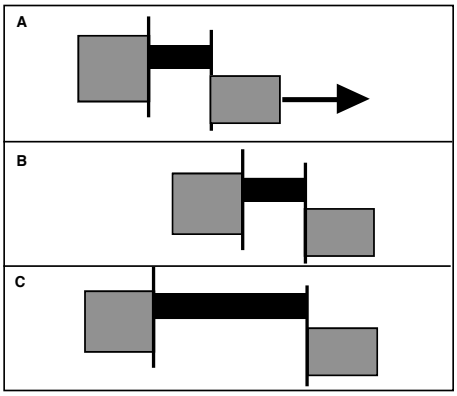

Fig. 2. Rigid and stretchy rectangles.

In the case of rectangles, CoDraw resolves conflicts by relaxing a constraint on a position in preference to relaxing a size constraint. Figure 2a shows three rectangles that have been constrained to be adjacent. (The vertical lines indicate adjacency constraints.) When the user drags the rectangle on the right, what should happen? One obvious behavior (2b) is that the other two rectangles follow along, attached rigidly. Another possible behavior (figure 2c) is that either the middle or the left rectangle stretches, changing its width to maintain the adjacency constraint. The same simple adjacency constraints can produce different interactive behavior.

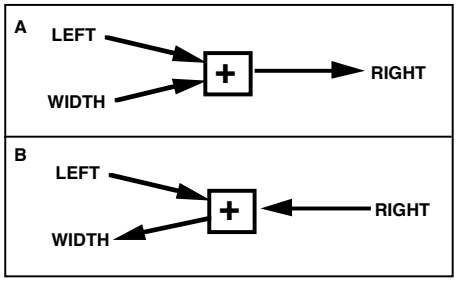

Fig. 3. Rigid and stretchy dependencies.

These different behaviors occur because of different internal states of the various rectangles. Figure 3a illustrates the relevant part of the internal state of a rigid rectangle; figure 3b shows a stretchy one. Arrows show the dependency relation among variables. The widths of the rigid rectangles (3a) are determined by direct constraints. When the user moves the rectangle on the right, the widths remain fixed and the edge positions of the other two rectangles are derived. In contrast, direct constraints fix the stretchy rectangle's edge positions and its width is derived (3b). A special "make-resizable" tool

on the command palette toggles the state of a rectangle between rigid and stretchy.

Often a spatial constraint relates two objects of different classes, for example a line and a rectangle. By default, CoDraw allows lines to control the positions of rectangle. When the user constrains a rectangle to center on a line, the rectangle moves to the line. If the user tries to move the rectangle it snaps back to the line. But when the user moves the line the rectangle follows along. Special "level" variables in the prototype definitions for lines and rectangles determine this default dependency: the element with the lower level number controls the relation. (Two elements at the same level are mutually dependent; when one moves the other follows.)

Default dependencies can be overridden by the user, either for individual instances or for object types. The "make-dominant" tool on the command palette overrides the defaults by installing local values that shadow the inherited level variables. To change the dependency for all instances however, the user must edit the level variables in the prototypes using the spreadsheet cards.

2.2 Grids and constraints

Grids are a convenient way to express placement constraints on graphics objects. A grid in CoDraw is simply another kind of graphics object that can be selected, colored, moved, resized, and assembled in groups. CoDraw's Grid Manager provides a special interface for defining prototype grid objects, and for making and applying instances to the Work Sheet. The Grid Manager displays a catalog of existing grids which a user can select, modify, or use as a basis for a new grid. Grids have two special variables *hseq* and *vseq:* these are lists of integers that describe the horizontal and vertical spacing of grid lines.

Using the Grid Manager the user can specify a placement rule for a particular object (e.g. circle-12)— or a class of objects (e.g. circles) relative to a particular grid or class of grid. By constraining the placement of different object types to different grids a user can make relative positioning rules. This is useful, for example, in architectural design: a rule placing pipes along one grid and wires along another simplifies costly interference problems [8].

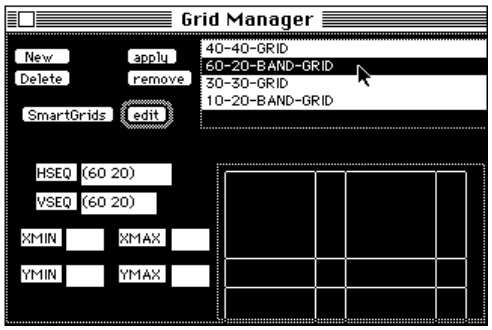

Fig. 4. CoDraw's Grid Manager.

2.3 Other features of CoDraw

CoDraw contains several other experimental features. Although CoDraw runs as a single user system, it is a prototype for a cooperative design program, where different designers controlling different classes of elements work together within the framework of constraints. Each different user can be registered and CoDraw is programmed to permit or prohibit the selection of certain element classes by certain users. For example the structural designer may be permitted to select and move only structural walls and columns while the electrical designer is permitted to select and move only wires and switchboxes.

Another simple and useful feature provides interactive interference constraints. With interference detection in beep mode CoDraw warns the user when two graphic objects coincide. In bump mode CoDraw prevents interference. In solids mode, only solid elements bump.

3 CoDraw's spreadsheet cards

A suite of spreadsheet cards provides the means to extend and modify CoDraw's graphical objects and constraints. These basic programming tools were used to build the initial set of objects and spatial relations that form CoDraw's built-in functionality.

3.1 Variables card

The Variables Card displays the variables of an object or class and their values. The display typeface is coded to show which variables or values are inherited (*italic*) and which are defined locally (**bold**). A similar scheme is used in C32 [11]. The typeface also encodes which values result from a direct constraint (underlined) and which derive indirectly from other constraints (not underlined).

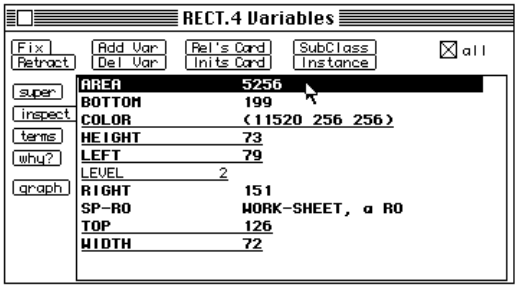

Fig. 5. Variables cards show values.

The user can assign a new value to a variable by typing it in. This action asserts a constraint on the variable's value. The new constraint is passed immediately to the solver and may be reflected by changes in the Work Sheet as well as in the values of other displayed variables. The Variables Card provides buttons for fixing and retracting

values, for making a new subclass or instance and for viewing the prototype of the object, for displaying other views of the object, and to inspect values and their justifications more closely.

3.2 Relations card

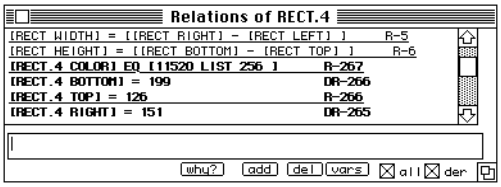

Fig. 6. Relations cards show constraints.

The Relations Card displays an object's constraints using the same typeface code as in the Variables Card. The constraint's name (`R-40, DR-41`) is also displayed. (`R-` indicates a constraint that was applied directly; `DR-` indicates a derived relation). Users enter new relations in the type-in area at the bottom of the card in Lisp syntax (an infix parser was written but proved cumbersome). The Relations Card provides buttons to add a new constraint or delete an existing one; the solver is immediately invoked each time a constraint is added or deleted. The *why?* button displays the justification graph for a selected constraint and can be used to track sources for a derived relation. The *all* and *derived* checkboxes control whether inherited and derived relations are displayed.

4 Graph windows

Several graph windows provide editable views on CoDraw's data structures. The Library Graph displays the inheritance hierarchy of prototypes. The Part Graph displays the assembly structure of the drawing or of subassemblies. The Constraint Graph displays a graphical view of the constraints and variables that define an object's behavior. The Justification Graph displays the chain of dependency that supports a particular value or constraint in the system.

4.1 Library graph

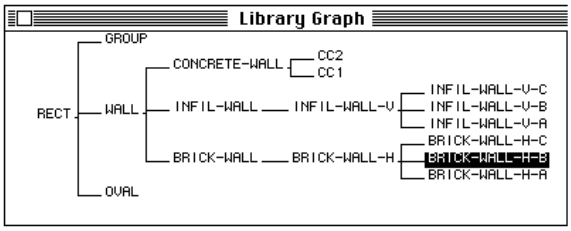

Fig. 7. Library graphs show inheritance structure.

A Library Graph can be displayed on demand for any subgraph of the library of prototypes. Double-clicking on a node in the Library Graph produces the Variables Card for that prototype. The user can also add a new prototype interactively using the Library Graph to specify where in the inheritance structure the new object should be located. The Library Graph can also be used interactively with the Work Sheet as a palette of graphic objects. After selecting an object type in the Library Graph the user can drag a new instance into the Work Sheet.

4.2 Part graph

The Part Graph displays the assembly structure of graphics objects and constraints. All objects, including work sheets and scrap are part of a *world* object. A 'display-related-objects' switch shows red lines on the Part Graph linking objects whose variables are somehow related. When a user selects an element in the Part Graph it is selected in the Work Sheet and vice versa. The user can also cut links and draw new links in the Part Graph, editing the assembly structure of the drawing. Cutting a part link deletes a part relation; drawing a new link creates a new part relation.

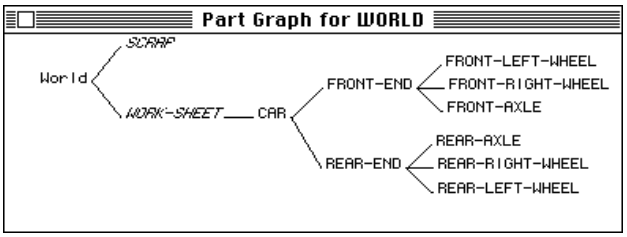

Fig. 8. Part graphs show assembly structure.

In Co, part relations also are represented using constraints. A variable defined in the superpart object has the part object as its value and likewise, a variable in the part object points back to the superpart. The names of these variables are generated automatically using the prefix `SP-` or `PART-` and the class name of the object, i.e. `PART-DOOR.3`. Finally a part-relation in the superpart (it could be located anywhere) associates these variables:

```
(PART  [HOUSE.1  SP-HOUSE]  [DOOR.3  PT-DOOR])
```

Thus when the user creates an assembly of parts, CoDraw asserts a series of part constraints.

4.3 Justifications graph

Justification Graphs display the chain of constraints that support any derived relation in the system. Each node can display a relation name (`DR-251`) or the full constraint expression. The nodes are color coded to indicate whether the relation is inherited or local to its object.

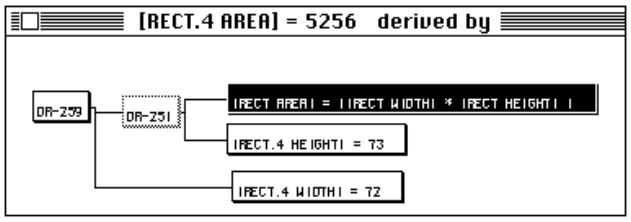

Fig. 9. Justification graphs show dependencies.

In this example a user has requested the justifications that support `RECT-4`'s `AREA` variable. The relation,

```
[RECT-4 AREA] = 5256
```

is supported ultimately by three constraints: two fixing `RECT-4`'s width and height, and the inherited constraint from the `RECT` prototype that defines the relation between height, width, and area.

4.4 Constraint graph

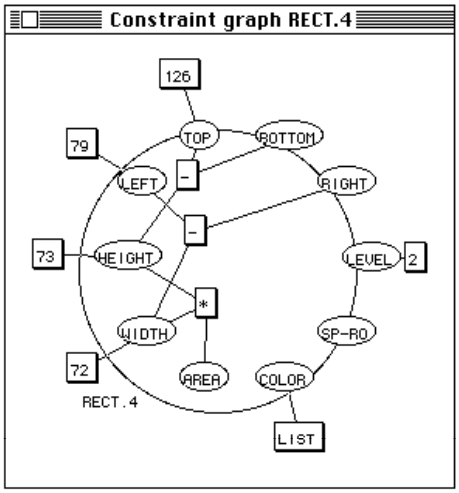

Fig. 10. The constraint graph.

A set of constraints is often represented as a network diagram that indicates variables and constraints and their connections. Pressing the "graph" button on an object's Variables Card generates a constraint graph of this sort (figure 10). Graph layout is automatic and can be adjusted by the user. Constraint operators and fixed values are shown as boxes and variables are shown as ovals. The large circle represents an object boundary; the diagram shows instance variables and constraints for a rectangle object.

5 Defining a new constraint type

Suppose we wish to define a new relation type, a constraint that operates on two objects in the Work Sheet to keep their areas equal. We might begin by choosing *graphic-relation* as a prototype for the new object, (the prototype for other built-in CoDraw spatial relations) and define a new extension (subclass) which we name *equal-areas*. Using the relations card for the new object, we enter the relation:

```
(= (A AREA) (B AREA)).
```

This constraint depends on structured variables to constrain the area variables of two objects (yet to be assigned). When both `A` and `B` are constrained to objects, then the equal-area relation will take effect.

We can apply the new constraint to two objects, `CIRC.0` and `RECT.1` whose areas we wish to be equal. We make an instance of the new *equal-areas* relation, and display its variables card. It shows only two variables (`A` and `B`), which stand for the two objects to be related. We enter the names `CIRC.0` and `RECT.1` for `A` and `B` respectively, assigning these variables to the two objects. Co's solver now evaluates the constraint `(= (A AREA) (B AREA))`, replacing variables by their values, and derives the constraint:

```
(= (CIRC.0 AREA) (RECT.1 AREA))
```

Now the system is overconstrained. Both objects already had dimensions and their areas were unequal. CoDraw, detecting a conflict, asks the user to choose a relation to retract from among the sources of the conflict. In this case the sources of the conflict are the dimensions of the two objects.

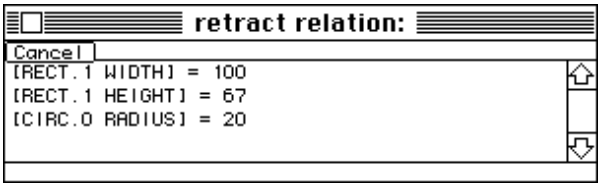

Fig 11. If no resolution has been programmed, CoDraw asks the user.

Alternately, we can program a conflict resolution method for CoDraw to select a relation to retract automatically. In any case, after a candidate relation is selected for retraction, CoDraw will keep the two areas constant through further editing. Because the two objects are mutually dependent (by default) if the user changes the size of either object, the other object will change size to keep the areas equal.

5.1 Icons for new constraints

With CoDraw's spreadsheet cards and graph editors a user can define new classes and subclasses of constraints. The user can use CoDraw's icon editor to draw an icon for a new relation and add it to the relations menu. For example, we can install a new icon on the relations menu that makes and applies an instance of the *equal-areas* constraint. The new relation and its icon remain in future work sessions and become part of the user's customized working environment.

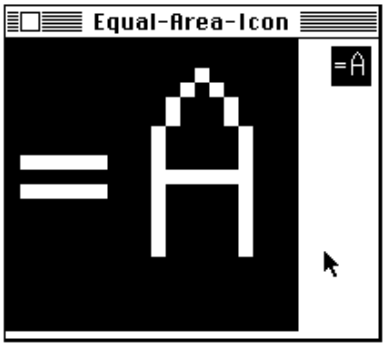

Fig. 12. Users make icons for new constraints.

6 Co: about the language

CoDraw is built using a constraint programming language called Co. Variables and Constraints are Co's atomic structures. These are bound together to make higher-level "relational objects", which model the user's domain (in CoDraw, graphic objects and spatial relations) and they are connected together in networks. Co's job is to support the user in the construction of relational objects, to maintain consistency in networks of constraints, and to help the user program and debug complex sets of constraints.

6.1 Constraints

CoDraw's spatial relations are built from more primitive algebraic constraints on one or more variables. Every constraint is treated as an n-way relation in which any variable can be computed from the others. A table of operators and their inverses and algebraic properties informs Co's equation solver how to produce one-way assignments for each variable in the constraint. Simple polynomial expressions (such as $x^2 = a$) can be handled but Co does not (yet) handle multiple solutions. Inequality relations are also supported; on input Co's constraint parser converts an inequality to an equation with interval values. Although this approach has limitations [4] it allows the system to represent and to begin to reason with inequality relations.

6.2 Variables

In Co a variable's value is defined by the set of constraints that reference it. The constraint *a = 43* asserts that *a* has value *43*. If *x > 4* and *x < 10*, then these constraints define *x*'s value. Often a set of constraints can be reduced by solving to a simpler expression of value. In this case, Co's solver will derive a new constraint, assigning *x*'s value to the interval between 4 and 10, or *x = [4 10]*.

To capture this extended sense of value in Co the value of a variable is represented as an ordered list of partial constraint expressions. The list is called (loosely) a *term-stack* and each partial constraint expression is a *term*. Each term represents a constraint solved for the variable in question. For example, the constraint *a = 2b* places the terms [= *2b*] and [= *(1/2)a*] on the term-stacks for variables *a* and *b* respectively. For debugging, term stacks can be viewed in windows that are updated as the solver works (figure 13).

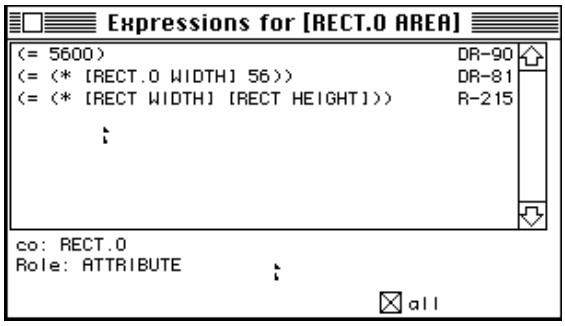

Fig 13. Term-stack shows constraint expressions.

Each term-stack is kept sorted with most simple expressions at the top. Simple is defined as a measure of number of variables, expression depth and length, operator complexity, and whether the term is from an inherited or a local constraint. In figure 13 the simplest term for `RECT.0`'s area, *(= 5600)*, is found at the top of the term-stack, while terms (= (* width 56)) or *(= (* width height))* which describe less specific expressions for the variable will be found lower. If a variable has a specific value it will be found at the top of its term-stack.

6.3 Init-forms

In addition to variables and constraints, every CoDraw prototype has a collection of "init-forms," Lisp code that is run when a new instance is made. This code can be used to assert or retract constraints or to program special user interactions. Like constraints and variables, initialization forms can be inherited; a new instance runs the cumulative set of init-forms of all its prototype superiors up the inheritance graph. An Init-forms Card similar to the Variables and Relations Cards provides an interface for entering init-forms.

Init-forms can determine where a constraint object will get its arguments. For example, the *equal-areas* constraint uses two variables, `A` and `B` to refer to the objects whose areas will be constrained. Suppose we want `A` and `B` to represent two elements selected in the Work Sheet when the constraint is applied. We can write init-forms in the *equal-areas* prototype that assign (constrain) the variable names `A` and `B` to the first and second objects in the selection list:

```
(NAME-SELECTED-OBJECT A RECT)          (NAME-
        SELECTED-OBJECT B RECT)
```

In this example, we also specified that CoDraw should type-check the objects and only allow objects of type *rect*. This argument can be omitted to accept an object of any type.

The init-form is also a good place to specify how to resolve the conflict that is likely to occur when the new constraint is instantiated. For example, we might add an init-form that specifies:

```
(RESOLVE-CONFLICT (RETRACT (B WIDTH)))
```

If a conflict occurs after installing the new *equal-areas* constraint, this init-form retracts the `B`'s `WIDTH` variable, making this object (the second one selected) dependent.

7 Discussion and further work

CoDraw provides a simple point&click interface for making drawings with behavior programmed by two-way constraints. Constructing drawings and applying the constraints is relatively straightforward. Making the constraints visible in the drawing enables the user to predict how the drawing will behave. However, with two-way constraints, the choice of fixed variables or anchors also determines the interactive behavior. A "show-witness-lines" tool on the command palette displays the internal state of a rectangle to help a user see whether the rectangle is rigid or stretchy. Domain knowledge about default dependencies among object types (e.g. lines control rectangles) can also help a user predict CoDraw's interactive behavior. A knowledgeable user can also inspect the state of the network using CoDraw's programming tools. However, displaying information about the internal dependency state of the constraint network in an easy-to-understand way remains a challenge for the next version of CoDraw.

CoDraw's spreadsheet cards interface supports an expert constraint programmer in constructing and debugging constraint language programs. Simpler strategies (such as learning from examples) would support less experienced programmers in defining new constraint types.

CoDraw's solving and conflict resolution mechanisms are also somewhat ad-hoc. CoDraw helped clarify the role of conflict resolution in programming a drawing interface (e.g conflicts must be resolved when the user selects an element for dragging or resizing; default dependencies are useful especially if the domain supports them). However, a more formal treatment would help. It would be valuable to provide CoDraw with a set of solving modules that it selects and applies according to the kind of constraints it is called to manage. Similarly, conflict resolution strategies can be made more modular, and made more accessible to users.

CoDraw began as an attempt to build a cooperative design environment for architectural layout where design team members in charge of different subsystems (structural, partitions, electrical, etc) can work together, guided by placement rules that coordinate their work. In the development of CoDraw, other issues — programming language and interface design issues — became interesting and important. Many of these remain to be addressed in future versions of the software; however, the initial goal of using constraints to coordinate cooperative design work also remains an interesting challenge.

References

1. Barzel, R., and Barr, A. A Modeling System Based on Dynamic Constraints. Computer Graphics (SIGGRAPH 88), 22,4, (1988) pp. 179-188.
2. Borning, A. Programming Language Aspects of ThingLab. ACM Transactions on Programming Languages and Systems, 3,4, (1981) pp. 353-387.
3. Borning, A., and Duisberg, R. Constraint-based tools for building user interfaces. ACM Transactions on Graphics, 5,4, (1986) pp. 345-374.
4. Davis, E. Constraint propagation with interval labels. Artificial Intelligence, (1987) pp. 281-331.
5. Fischer, G., and Rathke, C. Knowledge-Based Spreadsheets. In: Proc. AAAI-88. (1988), Morgan Kaufmann.
6. Freeman-Benson, B., Maloney, J., and Borning, A. An Incremental Constraint Solver. Communications of the ACM, 33,1 (1990) pp. 54-63.
7. Freuder, E. Synthesizing Constraint Expressions. Communications of the ACM, 21,11 (1978) pp. 958-966.
8. Gross, M. Knowledge-Based Support for Subsystem Layout in Architectural Design. In: Artificial Intelligence in Engineering: Design. J. Gero, Ed. Computational Mechanics Press, Southampton, UK, 1990.
9. Leler, W. *Constraint Programming Languages*. Addison Wesley, Boston, 1987.
10. Mackworth, A. Constraint Satisfaction. In: Encyclopedia of Artificial Intelligence. S. Shapiro, Ed. Wiley, NY, 1987, pp. 205-211.
11. Myers, B. Graphical Techniques in a Spreadsheet for Specifying User Interfaces. In: Proc. Human Factors in Computing Systems (SIGCHI '91). (New Orleans, 1991), ACM Press / Addison Wesley, pp. 243-249.
12. Nelson, G. Juno — A Constraint-based Graphics System. Computer Graphics, 19,3 (1985) pp. 235-243.
13. Steele, G.L., and Sussman, G.J. CONSTRAINTS - A Language for Expressing Almost-Hierarchical Descriptions. Artificial Intelligence, 14 (1979) pp. 1-39.
14. Sutherland, I. Sketchpad - a Graphical Man-Machine Interface [Ph.D. Dissertation]. M.I.T., 1963.
15. Wilde, N., and Lewis, C. Spreadsheet-based Interactive Graphics: from Prototype to Tool. In: Proc. Human Factors in Computing Systems (SIGCHI '90). (Seattle, WA, 1990), ACM Press/Addison Wesley, pp. 153-159.
16. Witkin, A., Gleicher, M., and Welch, W. Interactive dynamics. Computer Graphics, 23,2 (1990) pp. 11-21.