TYRO: A Constraint Based Graphic Designer's Apprentice

Ron MacNeil

Visible Language Workshop
Media Laboratory
Massachusetts Institute of Technology
Cambridge, Mass 02139

ABSTRACT

"TYRO", the (very novice) Graphic Designer's Apprentice is a visual programming environment to enhance the exploration and evolution of graphic design concepts. The basic unit of design knowledge, the prototype, is represented as a network of constraining relations with rules for asserting those relations. Similarly, a design is a constraint network of prototypes with its set of assertion rules.

The environment supports multiple simultaneous perspectives on each object (Logical, Spatial and Hierarchical Browsers) and a variation generator used for exploring the design space and discovering the assertion rule breakpoints.

1 INTRODUCTION

TYRO is an attempt to build an environment that parallels the cognitive style of a graphic design team: an art director as team leader with several assistants each with subtasks which are often to generate plausible variations on a theme. The team leader's directions tend to be goal statements using previous work as reference examples, so the system would want to build up libraries of templates or design prototypes which could be reused. The team leader is typically too busy to get involved in the specifics of the execution, so it's the assistant's job to figure out which procedure to use to meet the desired goal. A constraint system that is easy to use is a good match here since it requires a goal statement as its input and decides how to accomplish that goal with its own internal mechanisms. Since part of the design planning process is rule like, TYRO also provides a simple Prolog pattern matcher on condition objects built by demonstration which retract and assert constraining relations to control the variations generated.

2 THE DESIGN EXAMPLE

The design task I chose as the example domain is an award winning package design by the firm Design Elements, Lexington KY which typifies a kind of highly rational corporate design style that complements the product family concept prevalent in modern industrial design. A cursory analysis of the structure of fig. 1 reveals four distinct areas (or zones), each performing a different function in the task of communicating the product's purpose and selling the contents. In order of importance these zones are:

Logo Zone: the name of the product family "Attack" is the largest element in the design, and is underlined with a bold rule. Both the name and the rule are flush against left and right grid margins.

Target-Pest/Product-Name Zone: the name of the poor unsuspecting critter, rendered smaller than the logo and generally above and flush left with the logo.

Product-Description/Illustration Zone: the most idiosyncratic and fun area of the design, tells what the product does, shows to whom it does this.

Product-Contents/Disclaimer Zone: Typically contains two columns of small type, which almost function as gray rectangles at any distance from the package.

3 THE TYRO ENVIRONMENT

TYRO is built in Allegro CommonLisp and ObjectLisp [Dresher86], and runs on a 2meg Mac. The constraint language is a revision of work generously shared by Mark Gross, and uses classical propagation as its solution technique [Sussman,Steele81]. To increase speed the network is solved from the various points of view and the resultant compiled expressions are stored in the spatial-view's points. Selecting a different point...
to manipulate causes only a slight pause as the constraint expressions are reordered.

Intimately connected to the underlying constraint language is the graphical data-flow language. Currently supported primitives are the arithmetic and boolean operations, points, text and rectangles and a printing helper. Following the object oriented programming paradigm [Goldberg83], [Stefik,Bobrow86], the primitives are class objects with methods for constructing instances. Each class object was constructed by hand from a kit of parts using a simple direct manipulation editor.

Every top level object has three perspective views; logical, spatial, and hierarchical (or part/whole). Each perspective has display and interaction methods associated with their particular Editor window.

At least one terminus of each data path is a point object, which functions in the Spatial-View as a handle used to scale the object. In the Logical-View a helper object called the coordinate representative (or coord-rep) attached to each point object sends and receives x and y coordinate values.

Fig 2 is a small network in the making to illustrate how the wiring works. The left panel shows two vertically displaced points with their coord-reps, with a subtractor primitive placed to their right. In the right panel they have been wired up to a printing helper to show that the y distance between the points is 59 pixels.

Output data ports are represented as black squares, and in the case of the coord-rep I have drawn a white line through the middle to indicate the direction of the coordinate being represented (the vertically aligned ports are the x's, the horizontally aligned are the y's).

Unlike Thinglab [Borning79] and HookUp [Levitt86], my data-flow primitives are structured so that the diagram can be read in any direction. Input ports are
represented as white or black circles. Wire-Ors are dissallowed, and the system checks and doesn’t make a connection between two inputs or two outputs, which would cause an error.

Where possible, the function of the object is displayed as a set of graphical symbols inside the object. For example, inside the minus object is a little “help file” instructing one to hook the minuend to the white port and the subtrahend to the black. The numerical difference between the y values of the two points is displayed in the print helper. This object can be used as a probe on a data line, passing the data undisturbed to its destination, or it can be used as a constant holder, injecting that value into an input.

4 ABSTRACTIONS - the Cover Object

Tyro supports several abstraction objects using more or less the same methodology but with different levels of complexity and function within the system: covers, prototypes, conditions and commands. The cover object is the most simple of the bunch, as it holds the code representation of a network and presents at its output the result of processing the inputs.

Covers are different from most other abstraction mechanisms [Borning79, Ingalls88] in that a cover can have a control panel for user input. Fig. 3 shows putting the finishing touches on the ‘between-p’ cover. Between-p’s function is to return T if its input falls between the two variable values, else return nil. Its visual appearance is constructed in the Relation-Editor window where the ‘LO’ cover-variable object is being moved into place to complete the control panel. Option-clicking on either of the cover-variable objects brings up a dialog window asking for a new variable value.

5 PROTOTYPES

The Prototype is the chunk of design knowledge that represents the relational decisions made in the design process. It’s a useful construct for several reasons. From the team’s perspective it has the right cognitive shape, helping the team manage complexity by organizing the parts of the project which are likely to be reused. From the computational perspective, it’s a crucial notion that causes the design search space to have tractable dimensions. Without it, the search space quickly blows up. Graphically, the prototype is an abstraction mechanism that helps manage complicated tasks within the limited real estate of most computer screens.

Let’s start the scenario by defining the Logo Zone prototype. Clicking on the text object primitive in the palette in fig.4 on the left, causes a text-rep and its associated coord-reps to drag under the cursor for positioning in the Logical-View Editor window.

The text-rep object shows the slot information that is relevant to the user of this object, in this case the class name and instance string name. The inputs and outputs were added by hand in the Relation Editor when the primitive class object was made.

In fig.5 the Instance creation is completed by option-clicking the class-def and instance-def areas of
the text-rep and entering the class name and instance string, "Attack" into a dialog box.

When this is done, a text object is created and displayed in the Spatial-View Editor. Objects in this Editor have handles that allow them to be scaled, and mouseable containing rectangles that allow them to be selected and dragged.

The rule rectangle under "Attack" is the next element that needs to be added, and this is done by selecting it in the primitive palette, fig.6 and placing the rect-rep in the Logical window.

As with the text-rep, option/clicking edits the form and completes the class object, making a rectangle object that appears in the Spatial window, fig. 7.

The user, working in other design situations, might need only the Spatial-View Editor to be visible, so TYRO provides a facility to build constraint relations by demonstration [Myers88]. Choosing "left-sides" from the constraint menu, fig.8, and then selecting the two objects (or inverse order) will cause a constraint relation to be asserted between those members and be reflected in the Logical-View where an "equals" primitive is connected to the x output of the bottomleft coord-reps of the text-rep and rect-rep objects. A similar operation is carried out for the right side of the text and rect objects.

The vertical constraints are a little more complicated. We will build them in the Logical-View, shown in fig.10.

The height of the rule rectangle is to be 1/3 the height of the text, and a similar set of operations is carried out to constrain the top-left and bottom-left points of the rect-rep which then generates the desired effect on the rectangle in fig. 10.

The text and rectangle objects are now constrained in such a way that dragging on any of the handles in the Spatial window will cause the group to change shape yet maintain the proportions specified by the constraint network.

All that is needed to complete the Logo-Zone prototype is to add the prototype boundary primitive to the Logical-View and constrain it, fig. 12. Saving the
Prototype writes a file containing the class definition as well as the methods for creating the instance methods on demand. Saving the prototype also creates a Logo-Zone button in the Browser. Clicking this button will make two views of a new instance.

For the purposes of this paper we will gloss over the complicated problems posed by the definition of the pest and product description zones which involve defining a method of iterative exploration through a design space. That subject will be covered in a subsequent paper.

The design begins to take shape and be visible as an entity when a set of zone prototypes are assembled into a 'design prototype' (Fig 13a). The designer now has a way of exploring the design to discover what are the ramifications of changing a parameter, adding or deleting an item, etc. There are two ways of exploring: direct manipulation of the spatial window objects or setting the variation generator off to return a whole set of variations. Let's do the exploring by hand. Figs 13b,c show the result of grabbing the bottom right corner of the container and stretching the container in a range from vertical to horizontal. As the container is stretched the description zone can accommodate more or less text. What becomes clear is that there is no way, given the input data and this set of constraints on the assembled prototypes, to make a satisfactory horizontal, long and thin package type. We need another mechanism to accomplish this and in TYRO it's the design rule.

6 RULES

Defining a Rule is the way in which the designer tells the system the conditions under which to assert certain relationships. They are represented as pairs of a condition and action object, where action objects are generalized networks of prototypes and command objects.

7 CONDITIONS

Condition and action objects are built similar to prototypes by assembling a network in the logical-view editor. Fig. 14 is a network for calculating the ratio of the height and width of the container prototype and comparing that number to the numbers stored in the 'between-p' comparator. These two ratios define the limits within which the condition will succeed and are determined by the user from experimentation with various container proportions by hand in the spatial-view editor.

Before saving the condition and showing it as a new entry in the browser we are asked to choose the most applicable of several methods of generalizing the network. Choosing proto-instance causes the class container-zone to be substituted for the instance found in the network example so that all other instances of container-zone could cause the condition object to see a match.

8 COMMANDS

In fig. 15 we define the command to be executed when the horizontal container is recognized. By experimentation in the spatial-view, the designer determines that if the description-zone and the logo-zone are horizontally-alligned we have a design that at least has a chance of working. Alligning two items horizontally means to have the right edge of the left element abbut the left edge of the right element, and as well line up the tops and bottoms of the elements. The system builds a template using the network shown with the help of the names of the prototype corner points. We can generalize the prototype classes to make this a generic command. Saving the command stores it in the browser along with its sister commands.

Rules are constructed in the rule editor (fig.16) by defining the rule name, select-dragging the relevant parts from the browser into the if, then panes, and finally creating links when applicable.

The rule editor keeps track, and when the elements are in place the appropriate code is added to the prototype's rule slot.
Acknowledgements

Major support for the work described in this paper was provided by NYNEX. I would like to thank Nathan Felde and Sergio Canetti at NYNEX’s Media Lab as well as Muriel Cooper, Nicholas Negroponte, Henry Lieberman of MIT’s Media Lab for their support of this project. The Visible Language Workshop is also supported by grants and equipment from HP, IBM, Apple, DARPA, and Xerox.

BIBLIOGRAPHY

[Borning79], BORNING, Alan, ThinzLab, Xerox Report

[Dresher86], DRESHER, Gary ObjectLisp Manual

[Gentner83], GENTNER, Denise Structure Mapping, Bolt Beranek Newman Inc. Tech Reports

[Goldberg83], GOLDBERG, Adelle, Smalltalk 80, the Language and its Implementation, Addison Wes. '83

[Gross87], GROSS, Mark, Design as Exploration of Constraints, MIT PHD thesis

[Leifer88], LEIFER, Wm. Constraint Programming Languages Addison Wesley, 1988

[Levitt86], LEVITT, David, HookUp, Conference on Small Computers and the Arts

[Myers88], MYERS, Brad, Creating User Interfaces by Demonstration, Perspectives in Computing

[Ingalls88], INGALLS, Dan, The Fabrik Visual Programming Language, OOPSLA 88, SanDiego

[Shu88], SHU, Nan, Visual Programming, Van Norstrand Reinhold, 1988


[Sterling84], STEELE, Guy Lewis Common Lisp the Language, Digital Press

[Stefik,Bobrow86], STEFIK, Mark, BOBROW, Daniel, Object Oriented Programming: Themes and Variations, Al Magazine

[Sussman,Steele81], SUSSMAN, Gerald, STEELE, Guy Lewis Constraints: a Language for Expressing almost Hierarchical Descriptions, MIT Al Lab.

Fig.16
Fig.17

The most typical rule handles the spatial relationship of two prototypes. Let’s call this an arrangement rule, which is what we have just defined. Replacement, insertion and deletion rules are also needed to complete the minimum set.

Now we have defined a rule which allows us to extend the abilities of the design prototype in simple but useful ways. As the designer begins to experiment with dragging the corner of the container in the spatial view to change its proportions (fig.17), the system notices there is a new rule and makes a new entry in our rule data-base. At every new container ratio the tiny prolog tests for a condition match. When a match is found the action object is triggered. As the system begins to assert the constraints it checks to see if there are any constraints already present, and then deletes them before asserting the new constraints. These new constraints get compiled into the point objects in the spatial-view so that the next time the condition is detected the shift happens at interactive rates.

9 CONCLUSION

TYRO proposes a framework for experimentation with encoding design knowledge in the system, creating a system better able to act as an able assistant when needed. Future planned additions include a more natural, higher level interface for building ‘design heuristic’ objects than the design rule object currently in place, and a method for searching for stored prototypes or designs by analogy, using the work on structure-mapping by [Gentner83].

As electronic communication becomes more widespread, the range and complexity of design tasks increases the necessity to have the processor make many of the decisions now made by humans, in the design studio and at remote sites. Graphical constraint languages are a good match to the cognitive and computational resources of the emerging designer and the remote viewer.