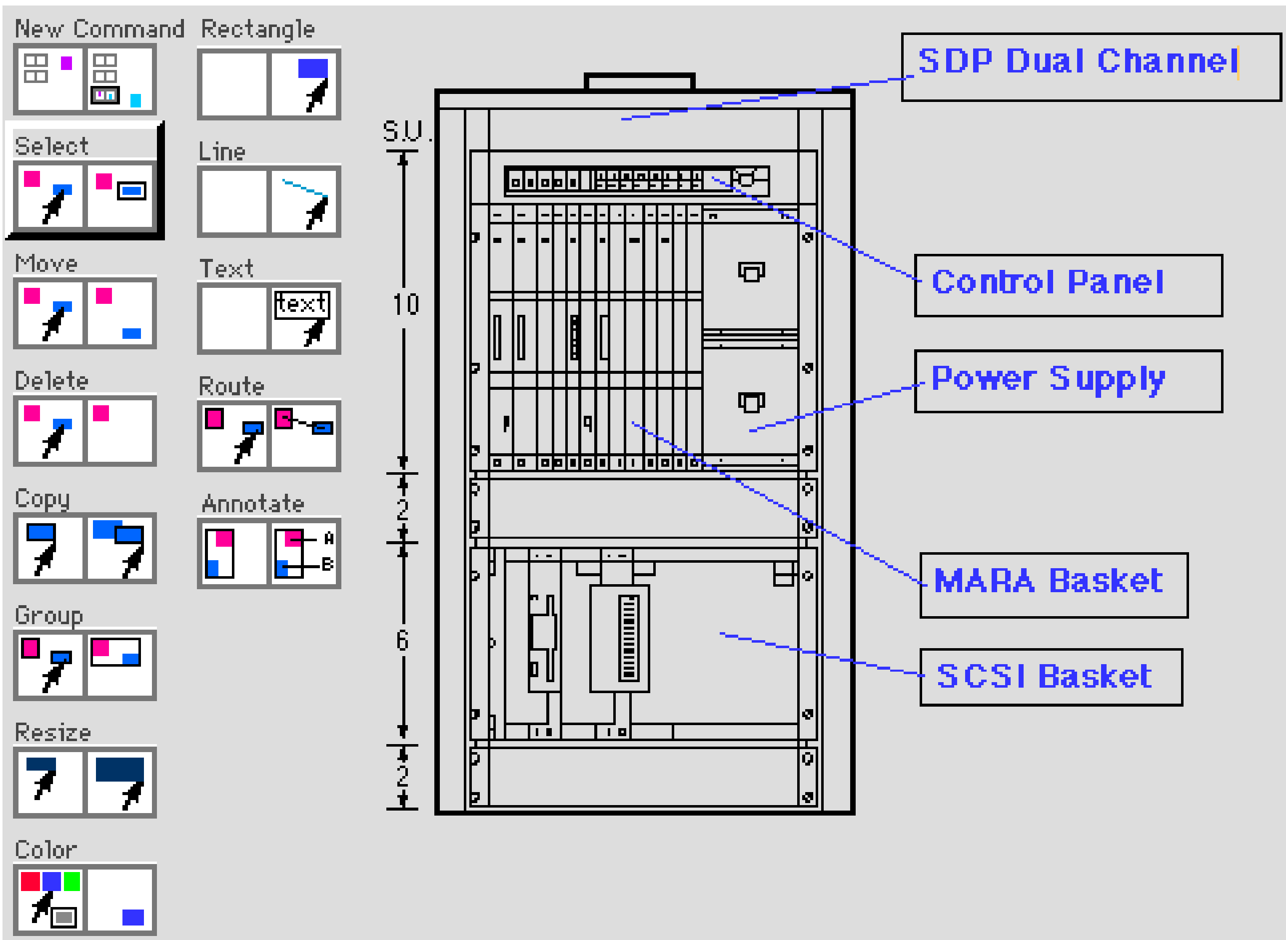
Graphical Annotation as a Visual Language for Specifying Generalization Relations

Henry Lieberman

Media Laboratory
Massachusetts Institute of Technology
Cambridge, Mass. USA
lieber@media.mit.edu



ABSTRACT

This paper explores how *graphical annotation* can be used as a visual language for specifying interpretations of user actions in an environment for programming by example [or "by demonstration"]. Attaching text labels to graphical elements is a natural visual notation that appears in many kinds of hand drawn diagrams, such as those appearing in user manuals, to indicate part-whole relationships.

Programming by example systems have the problem that each user action, such as clicking on a graphical object, is potentially ambiguous. When the domain of the system is extended, the user must communicate to the system what the intended interpretation of newly introduced objects should be.

Past solutions are either to encode a fixed mapping between user interface actions and the problem domain, or to allow the user to supply such knowledge by programming rules and frames in a textual language. In this paper, I present a technique for allowing an end user to add graphical annotation to a drawing to indicate the meaning of its parts. When a

labeled part subsequently appears as an argument to the operation, it is recorded in terms of its part description.

Keywords

Programming by example, generalization, graphical editing, graphical annotation, visual parsing.

Mondrian uses graphical annotation for generalization

Mondrian [Lieberman 93] is an object-oriented graphical editor that has a learning agent that records and generalizes user interface actions. It does so in part by constructing *descriptions* of the objects being manipulated in the actions. In Mondrian's base domain, these descriptions are in terms of geometric properties such as left, right, top, bottom, center, etc. However, Mondrian's domain can be extended where the pictures represent some physical or conceptual artifact, such as in CAD systems. In new domains, Mondrian needs to be informed about the descriptions of each part and their role in the artifact. Traditionally, this would be done by textual programming that would extend the mapping between user interface actions and the semantics of the domain.

Instead, we allow the end user to supply this information graphically, using labels and pointers overlaid on the graphics. We use a visual parser [Lakin 86] to map the graphical relations onto semantic relations. Mondrian, like most graphical editors, allows objects to be "grouped", and we use the group structure to represent the part-whole hierarchy. We use Common Lisp Object System [CLOS] objects to represent the descriptions, and are exploring the user of Framer, a description-based knowledge representation language [Chakravarthy, et. al. 92].

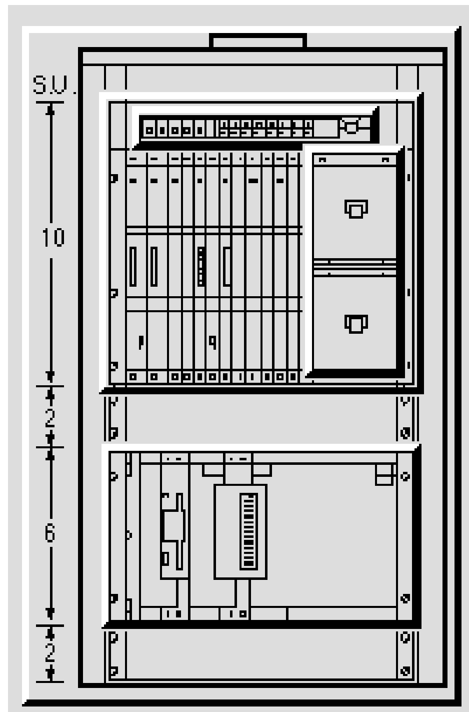
The user may then demonstrate procedures involving the parts of the device [operating its controls, assembly or disassembly procedures, etc.] and the actions will be recorded in terms of the labeled parts of the device.

Graphical annotation is a natural visual language for explanation

In many kinds of explanatory texts, such as users' manuals for tools or appliances, textual labels are attached to pictures of the device to indicate the parts and structure of the device. Natural language descriptions of an operational or maintenance procedure talk about the device in terms of its functional parts, and the user needs to be able to identify what parts correspond to the descriptive terms when working with an actual example. The annotated picture serves as a visual map between the semantic relations and an actual example. When the user is presented with a new device, he or she can try to construct an analogous mapping of semantic relations to parts of the visual presentation of the device that is analogous to the original one demonstrated.

Graphical object structure

We imagine that we have constructed an example image of a device on which we are to demonstrate a procedure, in the object-oriented graphical editor. This means that the parts of the device which play important roles in the operation are available as independently selectable objects in the editor. This would normally be the case if the drawing of the device were originally made using a CAD package, or if the device was part of some simulation software. If the device is represented as a single bitmap image, subsets of the bitmap could be copied as objects to represent the parts.

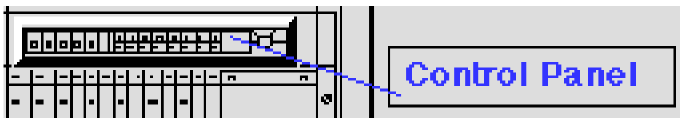


Object structure of drawing

In the industry, graphical editors such as MacDraw or Corel Draw are referred to as being "object-oriented", which often creates some confusion with the sense of "object-oriented" that pertains to object-oriented programming languages. The common factor is that independently identifiable notions such as physical objects or structural concepts are reified. In the case of the visual editor, objects and their parts are manifest visually and can be manipulated with the mouse. In the case of the object-oriented programming language, they are named in the language, and can perform computation. We take advantage of both senses of "object-oriented". The purpose of the annotation is to take the visual object structure of the graphical editor, and to introduce names and structure that create objects in the programming language which support computation about the referent of the image.

Attaching text labels

Meta-graphics are graphics that are "about" graphics. When we draw diagrammatic graphic elements such as text labels, arrows, dimensioning, etc. it is understood that these graphic elements are not part of the image being represented, but serve to communicate relations about the image. In order for the diagram to be understandable, there must be some sort of visual distinction, be it of form, color, or alignment, that enables the viewer to tell which graphic elements are part of the image and which serve to indicate the relational information. These conventions constitute a visual language.



Drawing part labeled by text label

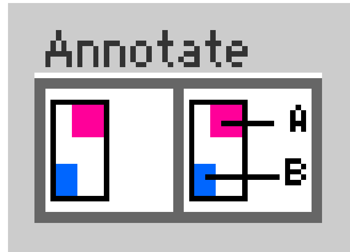
In Mondrian, parts of a drawing can be annotated with textual labels, together with lines that act as pointers connecting the text labels with the elements of the drawing that they describe. The text and lines are themselves graphic elements, and are constructed using the ordinary operations of the graphical editor.

Note that there is no inherent reason why the labels must contain text. Iconic labels would work as well, as long as there was some way to construct textual referents for them in the generated code.

The annotation operation

Mondrian provides an *annotate* primitive as an operation in the graphical editor. This primitive takes either a single text label or a group of text labels, and produces an *annotation*. The annotation is an object-oriented description, and an association between the parts of the drawing with the parts of the description.

Just as a human reader must interpret a diagram by distinguishing the meta-graphics from the base image, so must Mondrian have some way of making that distinction. The annotate operation launches a *visual parser*, a program that interprets the graphical relations between elements of the drawing as semantically meaningful.

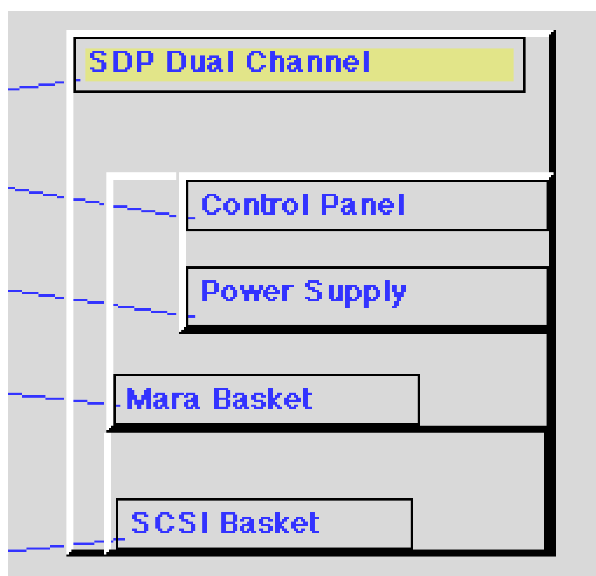


Annotate domino

The icon for the annotation operation follows Mondrian's style of "domino" icons [Lieberman 92b]. Each domino represents a visual example of the use of the operation through "before" and "after" panels that schematically represent the operation. In the case of Annotate, the panels show an object whose parts are labeled "A" and "B".

For each text label, the visual parser looks for a line one of whose endpoints lies within the bounding box of the text label. The other endpoint should lie within a graphical element of the drawing, and the text label is assumed to refer to that part of the drawing. The 2.5 dimensional ordering of the graphical object layers is taken as significant. Endpoints are always assumed to be pointing to the topmost object in the ordering.

Grouping indicates conceptual structure

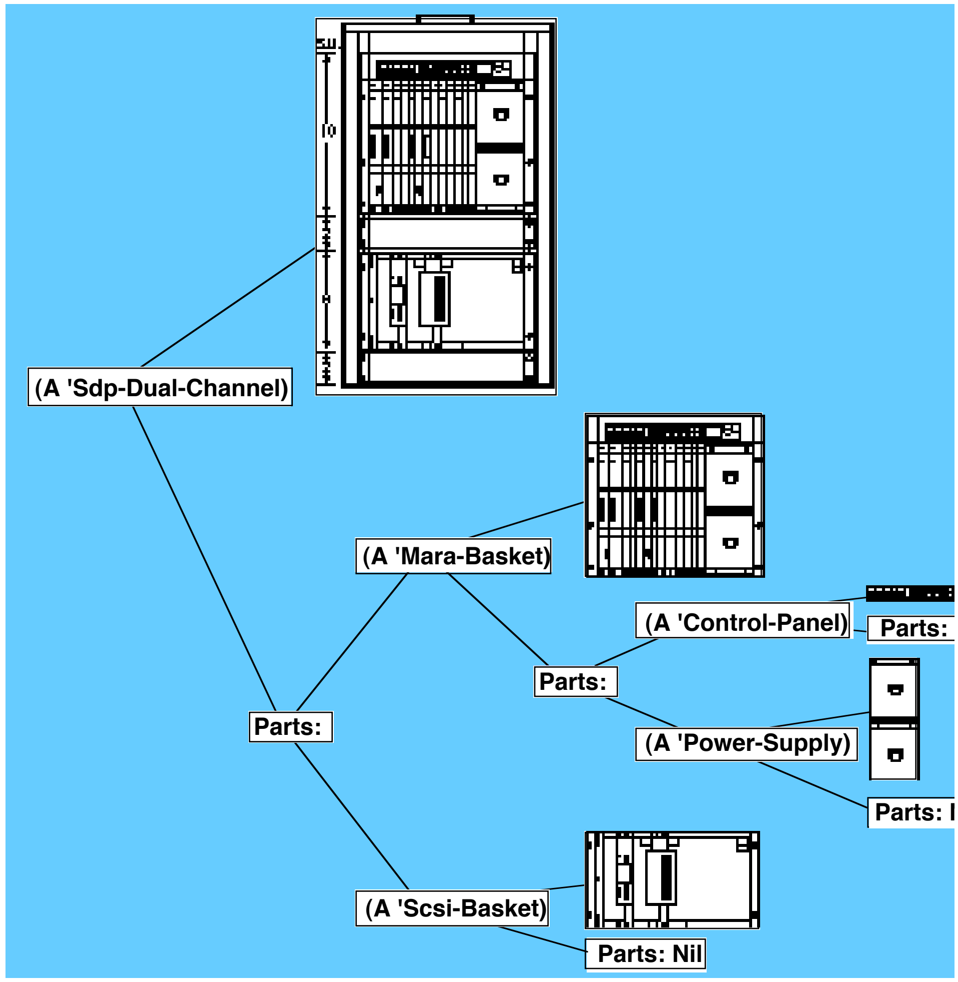


Group structure of text labels

We use the grouping structure of the graphical editor to convey part-whole relations in the diagram. Text labels for parts can be grouped using the Group operation of the graphical editor. Grouping a set of parts indicates that they belong to the same whole. Grouping a single text label with a set of parts means that the single label is the name for the whole.

Attaching this semantics to the grouping structure of the graphical editor has the advantage that grouping is already understood by users of graphical editors and requires no additional operations in the user interface. A possible disadvantage is that this overloads the grouping operation. It would also be possible to introduce another grouping operation specific to the annotation, that acted independently of the usual object grouping structure.

Each named part or whole is mapped onto a CLOS class, either an existing class or a new one created for this part or whole. The objects for wholes inherit from a class that includes an instance variable that holds a list of parts.



A visual representation of annotations

To give visual feedback to the user as to the results of annotation, we display a graph of the conceptual structure of the annotation. We use the Lisp tree grapher, which displays the name of classes and instance variables at nodes connected by lines. This is a standard "icons on strings" representation [Lieberman 92b], but a unique feature is to display the actual graphic elements of a part at the node for that part. This establishes a visual correspondence between the picture that was annotated and the conceptual structure that was produced as a result of the annotation.

Annotation graphics and device parts are grouped

The annotation operation also has two effects on the group structure of the graphical editor.

The annotation graphics, consisting of the text labels and the label pointers are also grouped as a unit. This allows the annotation graphics to be manipulated all at once. One use is to simply delete the annotation graphics, resulting in a "clean" picture of the device. A second use is that the annotation graphics can simply be moved off the original annotated drawing and moved onto another one. The pointers can be reconnected to the parts of a second drawing, and the annotation operation can be invoked again. This creates a structure of the second drawing that is "analogous" to the original, useful for example in annotating two different models of the same device.

Optionally, the parts of the object being annotated are themselves grouped together, to correspond to the structure that was parsed from the grouping of the labels. This permits the user to manipulate the whole object at once, since the part structure is then deemed to be of less interest once the annotation is complete. The part structure remains accessible by un-grouping. Since this would require then extra un-grouping operations in certain situations, we allow the user to turn off that mode, and annotation will not affect the original grouping structure.

From annotation to generalization

Mondrian uses the annotations to control how user actions are recorded during a programming by example session. Mondrian records user actions in a manner similar to a "macro" recorder, but instead of recording mouse positions and keystrokes, Mondrian tries to construct a symbolic description of the action in terms of a Lisp expression. Objects selected at the start of the demonstration become arguments to the recorded procedure.

Mondrian has three basic generalization heuristics. First, it looks for relations that are relative to the objects specified as arguments to the demonstration. Second, it looks for "significant relations" between objects. In Mondrian's base domain of MacDraw-style graphical objects, these start out being simple geometric relations such as left, right, top, bottom, center, above, below, etc. Third, Mondrian tracks the dependency relations between graphical commands and pushes generalization relations through the dependency tree.

Annotation gives the user a means of extending the set of significant relations. Annotation attaches a description to each of the graphical objects, so that these relations are retrieved when the object is to be generalized.

Mondrian has a generalization protocol implemented through a set of functions called REMEMBER functions that dispatch on the type of the object. These functions can also be dependent upon the context of graphic objects visible in the editor, and a generalization environment that supplies a code description for selectable objects. A user who is a programmer can extend the generalization by adding a single new method for each type of object introduced. Annotation provides another means of extending generalization, but through graphic interaction instead of programming.

At the lowest level the description of the object in the graphical editor is simply in terms of graphic primitives, i.e. a bitmap representing the image of the device. Selecting a part interactively, such as the object annotated as the control panel in the example image, constructs a description in terms of the part structure of the annotation. The part-whole structure is linked bi-directionally, that is, from parts to wholes and from wholes to parts. The part-to-whole links are necessary to go from the user's action of selecting a part to a description of the role of that part in the whole. The whole-to-part links are necessary when the procedure is subsequently invoked, to extract the part from a reference to the whole.

Imagine we are teaching the system a procedure to disassemble the computer rack shown in the illustration, which is named SDP Dual Channel. The Disassemble procedure is demonstrated with an operation Open that can be applied to individual parts of our device such as the control panel, or power supply. The SDP Dual Channel itself is selected at the beginning of the demonstration, so it is described as the first argument to the Disassemble procedure.

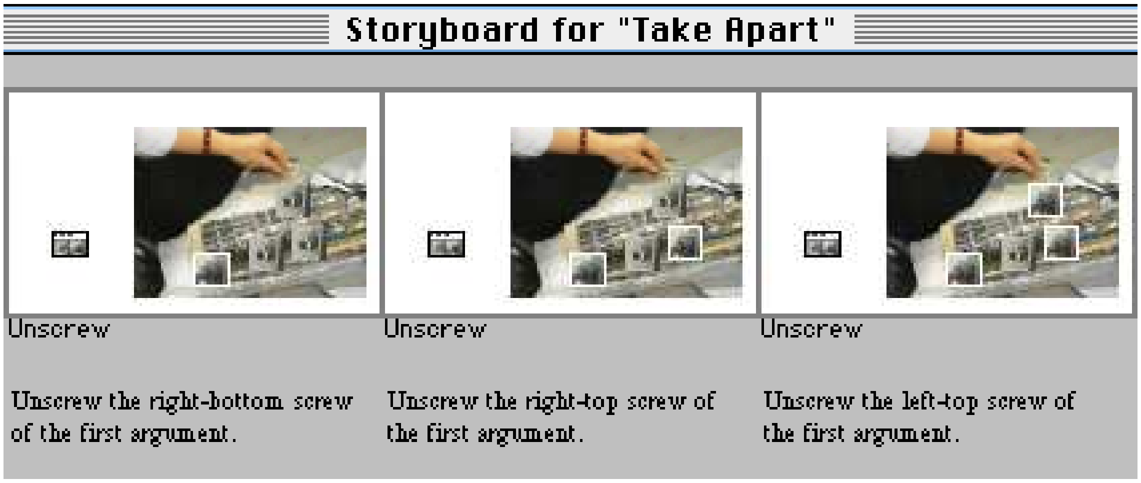
The generalization is represented as a Lisp expression which can be evaluated to access the part through a path through the part-whole hierarchy. Selecting the operation along with the control panel will yield a procedural description of that step as follows. The expression `(nth 0 selection)` represents the first argument to the Disassemble procedure.

```
(open-action
 (find-part-named (find-part-named
                   (nth 0 selection)
                   'mara-basket)
                  'control-panel))
```

In addition to the Lisp code representation, a natural language representation is also generated. We use templates attached to each expression to generate an English sentence describing the action. A speech synthesizer is used to narrate each action as the user performs it. This amounts to "reading the code to the user". It is a form of interactive feedback that allows the user to verify that the machine actually did understand the intended relations, and the use of speech does not interfere with the hand-eye coordination of the graphical interaction task. In the case of the expession above the speech output would be,

> Open the Control-Panel
> of the Mara-Basket of the
> first argument to the Disassemble procedure.

A second use of the natural language is to provide captions for a recorded graphical history of the operations performed. In the example illustrated below, the screws holding in the circuit board have been identified as the left-top screw, right-bottom screw, etc. Each of the Unscrew operations refers to the screw using these relations. If we re-execute the procedure on another circuit board on which screws have been annotated with the same relations, the actions will be applied to whatever objects in the new device that are described by the corresponding relations.



Related work

The definitive reference on programming by example is [Cypher, et. al. 93], in which Mondrian is described, and an earlier version of the annotation facilities [Lieberman 93b]. Halbert [Halbert 93] identifies the "data description" generalization problem for demonstrational systems, which inspired me to look for a more direct-manipulation solution. Lakin [Lakin 86] pioneered the use of visual parsing techniques. Weitzman and Wittenburg [Weitzman and Wittenburg 93] also use visual parsing to recognize layout constraints interactively in a graphical editor.

Acknowledgments

Major support for this work comes from a research grant from Alenia Corp. The research was also sponsored in part by grants from Apple, Digital Equipment Corp., Kansa Corp., NSF, NYNEX, ARPA, and Paws, Inc.

References

1 Chakravarthy, A., Weitzman, L., Haase, K., A Uniform Memory-Based Representation for Visual Information, European Conference on Artificial Intelligence, Vienna, 1992.

2. Cypher, Allen, ed. Watch What I Do: Programming by Demonstration, MIT Press, 1993.

3. Corbeil, J.C., Archambault, A., English/French Visual Dictionary, Facts on File, New York 1987.

4. Halbert, D. Smallstar, Programming by Example in the Desktop Metaphor, in [Cypher 93].

5. Lakin, Fred, Spatial Parsing for Visual Languages, in Visual Languages, S.K. Chang, T. Ichikawa and P. Ligomenedes, eds., Plenum Press, 1986.

6. Lieberman, Henry, Capturing Design Expertise by Example, in East-West Conference on Human-Computer Interaction, St. Petersburg, Russia, August 1992.

7. Lieberman, Henry, Dominoes and Storyboards: Beyond “Icons on Strings”, IEEE Workshop on Visual Languages, Seattle, September 1992.

8. Lieberman, Henry, Mondrian: A Teachable Graphical Editor, in [Cypher 93].

9. Lieberman, Henry, Making Programming Accessible to Visual Problem Solvers, in [Cypher 93].

10. Weitzman, Louis, and Wittenburg, Kent, Relational Grammars for Interactive Design, 1993 IEEE Workshop on Visual Languages, Bergen, Norway.