STYLE-BASED CUT-AND-PASTE IN GRAPHICAL EDITORS

Wayne Critin, Daniel Brodsky,
and Jeffrey McWhirter

CU-CS-706-94 February 1994

STYLE-BASED CUT-AND-PASTE IN GRAPHICAL EDITORS

CU-CS-706-94 February 1994

Wayne Citrin, Daniel Brodsky,
and Jeffrey McWhirter

Department of Computer Science
University of Colorado at Boulder
Campus Box 430
Boulder, Colorado 80309-0430 USA

Style-Based Cut-and-Paste in Graphical Editors

Wayne Citrin and Daniel Brodsky
Department of Electrical and Computer Engineering
Campus Box 425
University of Colorado, Boulder 80309-0425

Jeffrey McWhirter
Department of Computer Science
Campus Box 430
University of Colorado, Boulder 80309-0430

CU-CS-706-94 February 1994

Style-Based Cut-and-Paste in Graphical Editors

Wayne Citrin
Daniel Brodsky
Department of Electrical and Computer Engineering
University of Colorado
Boulder, Colorado, USA

Jeffrey McWhirter
Department of Computer Science
University of Colorado
Boulder, Colorado, USA

Abstract

Although great strides have been made in the last 10-15 years in the development of systems that use graphical representations, very little work has been done in developing systems that help users edit diagrams efficiently. This paper addresses the design of one such feature of a graphical editor, namely cut and paste. We show how knowledge of the syntax and semantics of the language being edited allows us to design a more intelligent cut-and-paste facility.

1.0 Introduction

Over the past 10-15 years, great advances have been made in the design of hardware that handles graphical images and in the software designed to exploit this capability. As understanding of the power of graphical notations and the sophistication of the hardware and software have increased, so have the complexities of the notations used. However, at the same time, there have been few advances in the technology used to *enter* diagrams into the computer, and to *manipulate* them once they are there. This issue has important implications for user acceptance of visual languages, since humans tend to adopt solutions to problems that require minimal effort [12], and when provided with a choice between a familiar keyboard-based text editor and textual language, and an unfamiliar and difficult-to-use graphical editor and visual language, users will probably choose the textual solution regardless of the merits of the graphical representation.

The issue of diagram entry has been dealt with elsewhere [4]. The other problem of graphical editors is related to the issue of diagram manipulation as exemplified by the cutting and pasting of diagrams in graphical editors. The conventional graphical editor, typified by MacDraw, provides a very simple and unsophisticated cut-and-paste facility. When a group of selected objects are cut from a diagram, MacDraw simply removes them from the diagram. The unselected elements are unaffected, and none of the damaged connections are repaired or otherwise altered. Similarly, when a group of objects are pasted from the clipboard back into the diagram, they are simply inserted into the diagram area and perhaps simply overlaid on already-existing elements. No extra elements are inserted to maintain the consistency of the diagram, and the pasted elements are not reconnected to the main body of the diagram. Thus, cutting and pasting on a graphical editor requires a substantial number of additional insertions and deletions of edges to bring a diagram to the desired, or most reasonable, shape. This makes diagram manipulation extremely cumbersome, and graphical cut-and-paste only marginally usable. This situation contrasts with that of text editors. In a text editor, the hole created when text is cut is instantly and transparently closed up. When text is pasted in a textual editor, the new text is similarly not overlaid on the old text, but rather a hole of the appropriate size is automatically opened in the old text, and the new text is inserted in it. Cut-and-paste in a text editor is useful and appropriate for the medium in question.

Figure 1 gives a motivating example of how conventional graphical editor cut-and-paste behavior hinders editor usability. In figure 1, we see a possible desired transformation of a sequence of three statements in a flow chart to a construct where the last two statements become alternatives chosen after a test. On a system like MacDraw, the transformation requires approximately ten editing operations. Note that we generalize a move operation to be a cut operation followed by a paste operation. Disconnected edges must be selected and cut, a decision node must be inserted, and the appropriate edges must be reinserted. (Please note that we are not using flowcharts in our examples because we think that they are a particularly interesting or useful visual notation, but rather because they have a fairly rich semantics, and because their semantics is well understood by readers. The principles described in this paper are applicable to other, more interesting, languages.)

The editing actions required to transform figure 1(a) to 1(b) seem needlessly complicated. In particular, with knowledge of the syntax, semantics, and drawing style of decision nodes in flowcharts, it should be possible to automate the steps in which edges are removed and connections reestablished. However, while it might be relatively simple to design and implement optimal drawing strategies for any particular transformation, it is far more difficult to design a general framework to provide reasonable actions for a large variety of cut-and-paste situations in an editor for a given language.

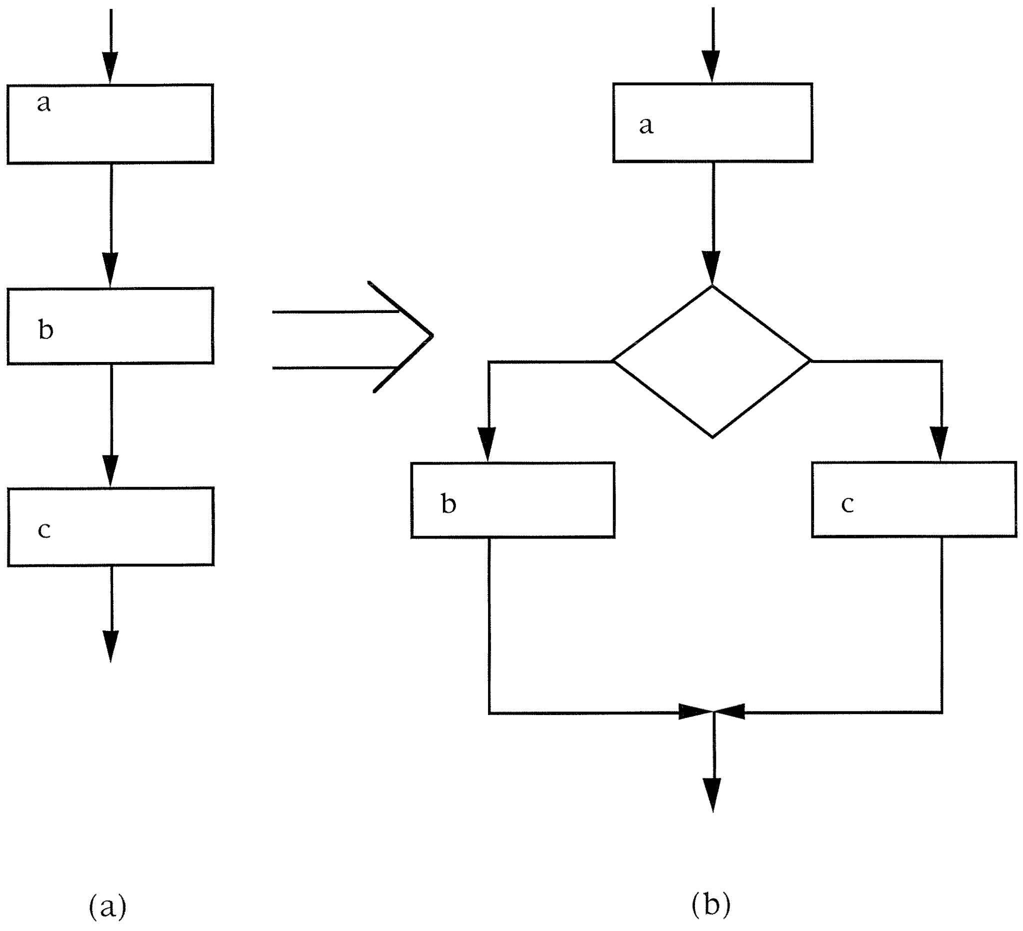


Figure 1: Example transformation

We are currently investigating a number of cut and paste schemes. We give a general overview of the investigated schemes below, and we discuss one particular technique in detail. The scheme that we spend most time on is based on both diagram syntax and semantics, along with a set of style rules that need not be strictly adhered to, but which allow the editor to make certain assumptions about what a damaged diagram represents, and what repairs are necessary to restore its integrity. Our systems have been implemented on a graphical editor specification and construction system called Escalante [9], which is described in some detail later in the paper. Although we concentrate on the flowchart example already presented, we have also implemented editors implementing cut and paste on trees and Verdi [5] programs. These will be presented in a forthcoming long paper. The flowchart example, however, will serve to illustrate our principles.

2.0 Alternative Approaches

In addition to the style-based scheme presented below, we have considered three other cut-and-paste schemes. We do not assert that these are worse than the one we present, and we plan to perform further research on these, but the style-based scheme is simple and easy to describe, and exhibits a reasonable behavior on a number of tests. A forthcoming paper will discuss each of the other approaches in detail, and will include examples.

2.1 Proximity/gravity fields

Gravity fields, or *snap-dragging* [2] is a method for ensuring that diagram nodes and edges become connected without requiring a great precision on the part of the user, who would otherwise have to make sure that the mouse cursor is precisely positioned over a node before choosing an action that causes an edge to be drawn. Instead, nodes are considered to emit *gravity fields*, which attract ends of edges that come within a certain distance. When the end of an edge comes within a node's gravity field, the edge snaps into place on the node. To further increase the usefulness of this approach, Hudson [6] proposed semantics-based gravity fields, in which the attraction is *selective*, so that only certain types of nodes and edges are attracted to each other, and *directional*, so that the attraction only emanates from a node in certain directions. Although the method was proposed for diagram entry, the same principles can be used to support diagram editing and manipulation.

Use of gravity fields for diagram editing has the advantage of being simple, since the cut-and-paste properties of all the nodes can be simply and concisely described with a few rules. However, performing transformations using this technique may require a few operations that may not seem natural. In particular, transformations may require creation of intermediate configurations that are more "damaged"; that is, that contain unconnected components or otherwise represent configurations that do not constitute a portion of a syntactically legal graph, at least in the way a user would ordinarily draw it.

2.2 Semantic nets

Von Känel [11] has proposed a scheme for implementing graphical cut-and-paste in which diagrams are interpreted as semantic nets. Some objects in the diagram represent entities, and some represent semantic relationships between them. In addition, spatial relations may represent semantic relationships. The corresponding semantic net is a graph in which entities are nodes, and relationships are edges. Von Känel proposed that cut and paste operations should not damage the semantic net corresponding to the graph being manipulated any more than necessary. Unfortunately, he did not propose a metric for damage, and the repairs that he envisioned are fairly simple and are not of much use.

In general, what Von Känel proposed was that when a group of graphical objects is cut, it should be replaced by instances of the most generic graphical object at all connection points at which the original group of objects was connected. Unfortunately, Von Känel did not propose an accompanying paste behavior, but we may imagine one where groups of objects are pasted in place of generic symbols, and are connected where possible in order to preserve the semantic qualities (again, using an undefined metric). For example, we may have the editor attempt to preserve (or reduce) the number of unconnected edges in the graph before and after the paste.

The behavior of semantic-net-based cut-and-paste resembles in many ways the behavior of structure (that is, syntactically-based) editors for graphical languages in that elements may only be added to the graph in limited ways, and in limited places. This provides the advantage of allowing the editor to employ syntactic and semantic information in the language but not in the graph itself in reconstructing the graph after cuts and pastes.

However, it also contains many of the disadvantages found in structure editors - namely, the fact that elements may only be introduced into the graph in a very restricted way. A semantic-net-based approach is a bit more flexible than the syntax-based structure editor approach in that we are not restricted to entering elements in the order of a syntactic derivation, but even restricting the intermediate graphs to ones that are semantically valid (or whose degree of invalidity is preserved) is more restrictive than typical human drawing patterns appear to be.

2.3 A syntax based on editing operations

Arefi *et al* [1] proposed to simplify graphical parsing by defining diagram syntax not in terms of spatial relationships among elements, but rather in terms of the allowable sequences of editing operations that may be used to create the graph. Since editing operations are a stream, conventional one-dimensional, or textual, parsing techniques may be applied to determine syntactic validity. Also, because the grammar is based on ways in which the diagram is drawn, rather than on the hierarchical structure of the diagram, we may base this diagram on ways in which people actually draw diagrams. For example, experiments suggest that people generally do not draw directed edges whose source is not anchored in an already-existing node [10]. Thus, to draw two nodes connected by a directed edge, users may draw the two nodes and then draw the edge connecting them; or they may draw one node, then the edge with the source end anchored at that node, then the other node placed at the sink end. What they will almost never draw is the edge followed by the two nodes, or one node followed by the edge with the sink end anchored to it, followed by the node placed on the source end. These characteristics may be captured in a grammar of allowable editing operations. This grammar may be ambiguous, since there are still numerous correct ways to draw a diagram, but for our purposes, this ambiguity is unimportant.

Editing operation-based syntax may be employed in a graphical cut-and-paste scheme. If we assume that we have a graphical editor that only allows syntactically legal sequences of editing operations, then we may assume that any intermediate diagram must be syntactically correct up to the point at which drawing stopped. In other words, the editing operations required to produce a valid intermediate diagram constitute a *legal prefix* of a sequence of editing operations that will produce a syntactically correct diagram. It is possible, however, that a cut operation will produce an illegal intermediate graph (i.e., one that requires an illegal sequence of editing operations for its production). In this case, it should be possible to repair the resulting intermediate graph and restore its validity through the use of syntactic error recovery techniques. In this case, the invalid intermediate graph would be converted to a (syntactically incorrect) sequence of editing operations that would produce it, syntactic error recovery would be applied to that sequence to produce a valid sequence of editing operations, and a new, presumably valid, intermediate graph would be redrawn from this sequence.

A cut-and-paste scheme based on syntactically correct sequences of editing operations shows great promise, but more work needs to be done. Further investigations must be made in both the specification of graphs through syntactically valid sequences of editing operations, and in the application of syntactic error recovery techniques to incorrect editing sequences. Finally, it is possible that a system using this scheme will produce unanticipated results; this should be investigated. We plan to pursue these investigations, but in the meantime, we have implemented a simplified version of this scheme, called "style-based cut-and-paste", which exhibits many of its properties. We discuss this scheme below.

3.0 Style-based cut-and-paste

Semantic and syntactic rules are not the only criteria upon which one can base cut and paste rules. Although it is often useful and even necessary to use these criteria, often times it is not desirable or possible. On these occasions, there is a more generic approach to determining cut and paste rules. By using rules based on stylistic or aesthetic characteristics of a visual language, an editor that makes more intuitive sense can be generated than one using semantic or syntactic rules, for certain languages. Such a scheme not only makes intuitive sense, but allows a simpler implementation than syntax-based cut-and-paste schemes.

Often these style rules are based on conventions. If the convention for some language specifies that some node that appears below another node follows the latter node, a useful rule can be inferred. Using this example, if three nodes appear in a column, and the middle node is cut, the top node is still above the bottom node, and thus precedes it, indicating an edge should be inserted from the top to the bottom. Also, if a node is pasted between two others, it is obvious that since the new node is below the top node, that new node should follow it, and since it is above the bottom node, the new node should precede it. The complete list of style rules for flowcharts will be explained in full later in the paper.

4.0 Escalante

We have used the *Escalante* system [9] to construct the prototype environment described in this paper. Escalante supports the rapid construction of highly functional visual language environments with a minimal amount of manual programming. The target domain of Escalante is graph-model-based visual languages. This characterization of the domain refers to the underlying language constructs, not any particular graphical representation (e.g., nodes and edges).

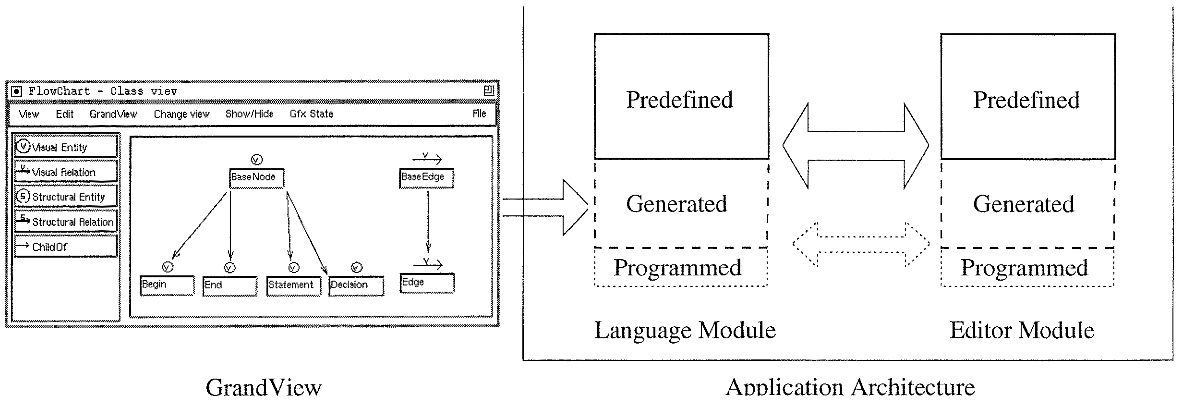


Figure 2: Escalante architecture

Escalante is an object-oriented system composed of three components: a base language module, a base editor module, and the *GrandView* language specification environment. Figure 2 shows the development process and a conceptual view of the target application architecture. Applications built using Escalante are composed of a language (or data) module and an editor (or control) module. The *language module* encapsulates most of the language-specific functionality required within an application, including the application data model and its representation. The *editor module* consists of a built-in editor model that offers a rich set of interaction mechanisms and can be adapted by the language designer to support language- or application-specific interaction techniques. We have taken a language-centered approach for the principles underlying Escalante, meaning that visual applications are defined around the underlying specification of the visual language; as a consequence, the system tends to focus on the language module rather than on the editor module (or other application-specific modules that might be added manually).

Escalante has been used to construct a wide variety of visual language applications including the system discussed in this paper. Systems built with Escalante typically require very little manual programming to realize the desired language and editor behaviors.

The editor module of Escalante encapsulates a wide range of visual program editing capabilities including: the creation, deletion, and copying of language elements; graphical editing capabilities such as moving, resizing, scaling, alignment and simple layout; and grouping and manipulating groups of elements. There is a framework provided for creating on-line help. N-level undo/redo of element creation, deletion and movement is supported. One can copy/paste and export/import components of a graph. Very flexible mechanisms also exist for multiple views, viewing subgraphs, and filtering out the display and selection of elements. The generated editor module is composed of a set of template classes derived from the base editor classes. These template classes can be tailored to fit the particular needs of an application.

4.1 Escalante Support for Cut-and-Paste

Beyond the ability to rapidly construct visual language environments, Escalante aided this research effort in two ways.

The first involves *event propagation*, a mechanism encapsulated in the language module. A relation (e.g., edge) can define the propagation of certain predefined editor events between its tail, itself and its head. These events include the moving, picking, and deleting of elements. The event propagation mechanism causes the particular event to be propagated to other elements. For example, one can define that when the tail of some relation is copied, the relation and its head are also copied. The event propagation mechanism enables the semantics the language to affect the interface/editing behavior.

The second aspect of Escalante that facilitated this work is the ability to declaratively define the default addition of relations between elements based on element type and spatial positioning. This specification is accomplished in the GrandView environment.

5.0 FlowChart Editor

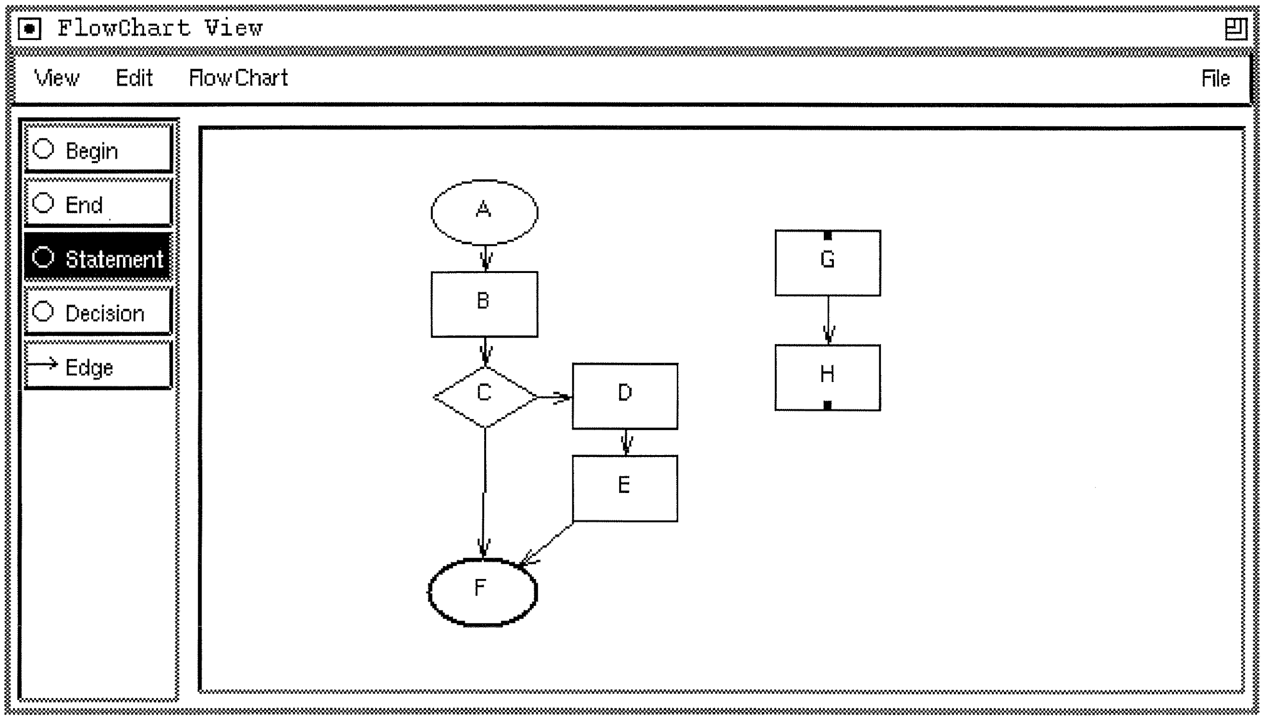


Figure 3: Flowchart editor

We have developed a prototype editor for creating flowcharts. In this editor, we have explored the role of style-based rules for driving the cut/paste behavior of the interface.

Figure 3 shows the FlowChart editor. This system allows the user to create and edit simple flowcharts. A flowchart is made up of *Begin*, *End*, *Statement*, and *Decision* nodes. These nodes are linked together with relations of type *Edge*. A Decision node may only have at most two outgoing edges. A Statement node may have at most one outgoing edge. Each node shows a small black dot at its top and bottom when there are an incorrect number of input or output edges. For example, in Figure 3 the Statement node labeled ``G'' has an incorrect number of incoming edges and the Statement node labeled ``H'' has an incorrect number of outgoing edges.

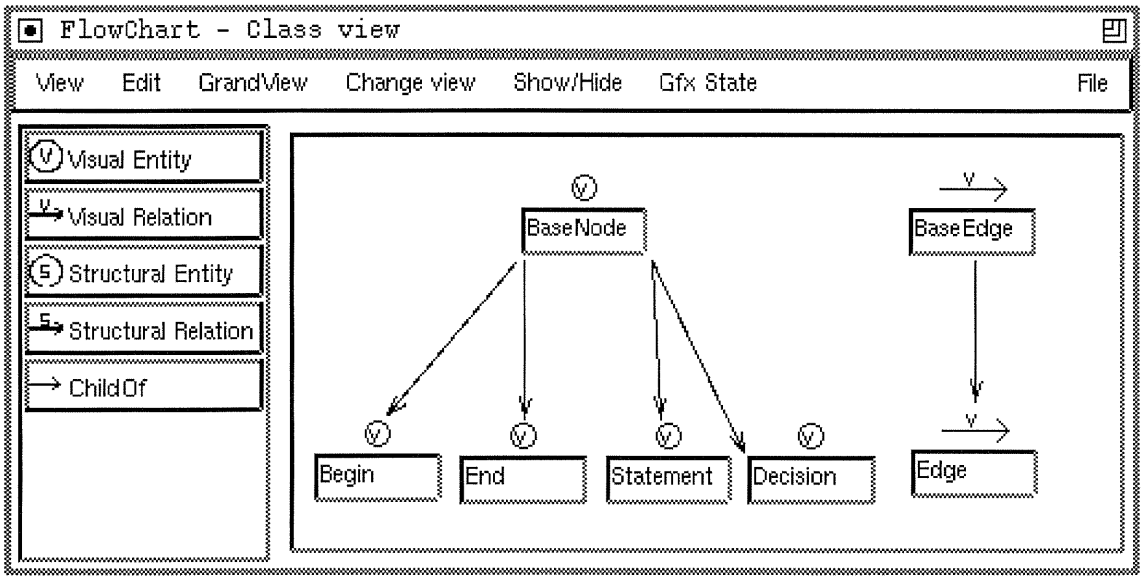


Figure 4: Flowchart specification

5.1 FlowChart Editor Implementation

The Escalante system was used to construct the FlowChart editor. Figure 4 shows the Class View of the specification for the FlowChart editor. Escalante supported some aspects of defining the cut/copy/paste behavior of the FlowChart editor. However, approximately 200 lines of code had to be written to implement the application-specific interface behavior required of the FlowChart editor.

5.2 FlowChart Cut-and-Paste Rules

Rules concerning the appearance of flowcharts fall into two categories: primary visual rules, which directly reflect semantics of the underlying programs, and which must hold; and secondary visual rules (also called style rules) which should hold if possible, in order to improve readability of the diagrams, but which are not necessary to produce a semantically correct flowchart. In addition, style rules may be ordered by priority, so that, in case of conflict, certain rules should be accommodated before others. An example of a primary visual rule is that if a statement A immediately precedes another statement B, there must be an edge in the flowchart from A's node to B's node. An example of a secondary visual rule is that if A precedes B, then A's node should be placed above B's node in the flowchart, unless other style rules, or other edge connections (such as those in a loop or conditional) make this impossible. Style rules may also be applied backwards, in the sense that if the semantic relation between two objects is otherwise unknown, style rules may be used to help determine the semantic relationships. For example, if a node A is located above a node B after a diagram manipulation (i.e., a cut or a paste), and the semantic relation between the two nodes has not been specified, consultation of the style rules may indicate that A should precede B, and causing an intelligent editor to draw an edge from A to B (thus enforcing the primary visual rule). This reverse application is how the style rules are used by our editor.

Our flowchart style rules fall into three classes: vertical ordering (including the rule described above), endpoint holding, and proximity. The vertical ordering rule described above is actually more specific. Namely, it states that if A is above B, there are no other nodes between A and B, and A and B are within a certain number of units of each other, A should be interpreted as immediately preceding B, unless the contrary is specifically known.

Another style rule describes the joining of branches of conditionals. If a node B is located below a decision node, and another node A is located to its right (again, with no intervening nodes, and if A and B are within a certain threshold distance), then A is assumed to be a branch of the conditional, and it rejoins the main execution stream at B. In other words, A precedes B. Figure 5a shows a diagram configuration that satisfies the style rule's condition, and figure 5b shows the result of applying the style rule. Note that the determination that node A is part of a branch is done locally, through its position relative to B, and not through tracing execution paths back to the decision node. A similar rule applies when A is to the left of B.

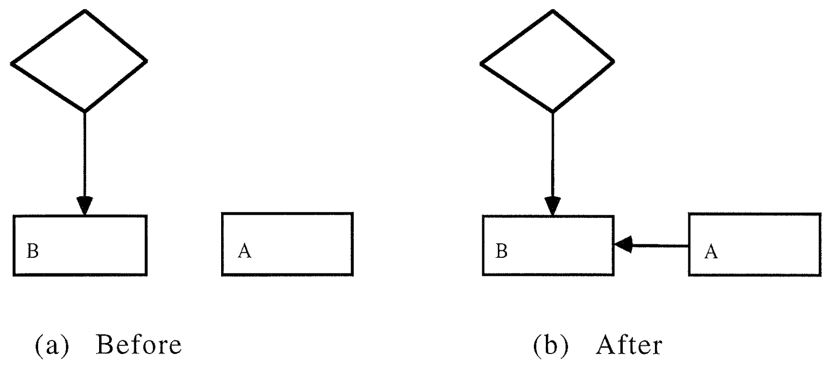


Figure 5: Rejoining a branch

Other style rules concern the interpretation of the relations between decision nodes and other nodes. If A is a decision node, and B is immediately to its right (with no other intervening nodes and within a certain threshold proximity), A is considered to precede B, assuming that nothing to the contrary holds. In particular, B is considered to be the first node on A's "Yes" branch, but we are hot considering that level of semantics here. A similar rule holds when B is to the left of A (except that it concerns the "No" branch). A lower priority style rule indicates than when B is located immediately below the decision node A (with no intervening nodes and within a certain proximity), A precedes B, and B is on A's "No" branch. This rule may be overridden by the preceding rule, as figure 6 shows.

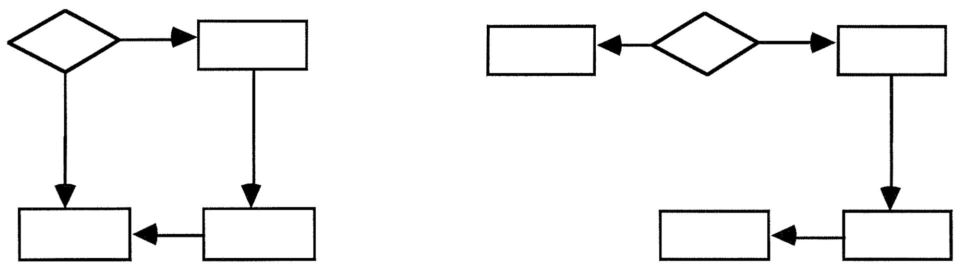
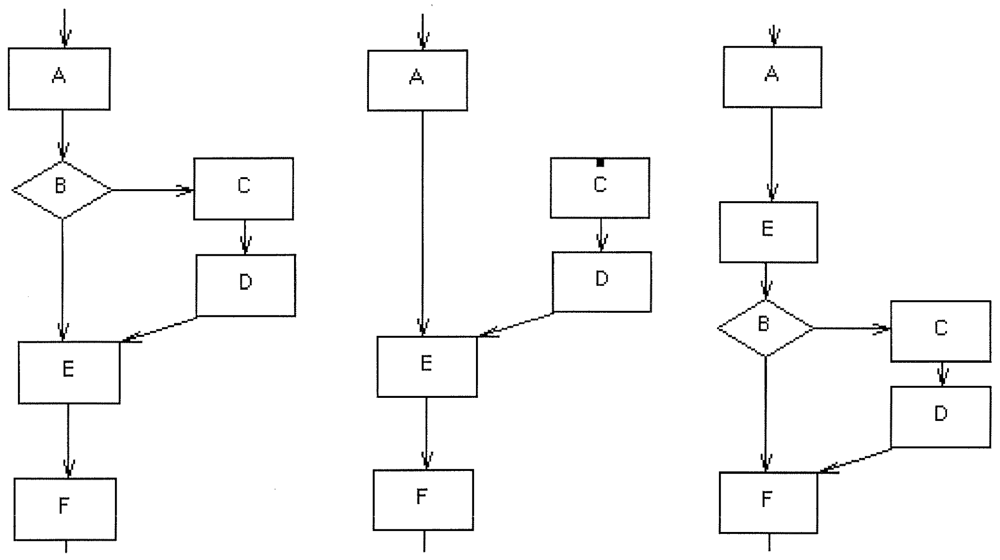


Figure 6: One style rule overriding another

There are a number of other style rules concerning start and end nodes, but they are similar to the rules already discussed, and will not be discussed here.

Finally, we provide a rule that suggests that endpoints of edges cut as part of a selection and copied to the clipboard be preserved, unless other rules are violated. This rule, which is not an interpretive rule like the others discussed above, allows us to move elements while maintaining their connections. This is often what users require, and yields expected editing behavior in the vast majority of cases..



(a) initial (b) cut B (c) paste B

Figure 7: Style-based cut and paste example

Figure 7 gives an example of some of these rules in action. Let us assume that we wish to transform the flowchart in figure 7a so that the decision node comes *after* the node E, not before. We cut the decision node, resulting in the flowchart of figure 7b. The decision node has been copied to the clipboard, but the clipboard also preserves the connections that possessed by the decision node before it was cut. Note that the style rules force the connection between A and E.

When we paste the decision node back into the flowchart (figure 7c), the connection between the node and statement C is restored, and an edge from E to the decision node is automatically constructed, as is a connection from the decision to node F. In addition, the style rules force the connection from node D to node F, since the node below the decision node is assumed to be a join node.

6.0 Conclusions and future work

In this paper, we have introduced the issue of graphical cut-and-paste, which has generally been neglected by investigators. We have considered four approaches and discussed their relative advantages and disadvantages. One approach, style-based graphical cut-and-paste, seems to strike a balance between simplicity of specification and implementation and sophistication of effect. However, a cut-and-paste scheme based on a syntax of editing operations merits further study, and we plan to investigate this approach as well as the style-based approach.

In addition to further investigation of the two above mentioned approaches, the work presented here suggests a number of other directions of research. A number of methodology issues present themselves, particularly in the case of the style-based approach. At the moment, derivation of style is in most cases a rather *ad hoc* enterprise, although it is sometimes helped along by the presence of style books such as those available for flow charts [3,7,8]. Further work should investigate the automation of style rules, either through questionnaires presented by an editor generation system, or through the use of examples.

In the technological area, we plan to investigate enhancements to Escalante that will support the specification and implementation of graphical cut-and-paste behavior. We note that although the specification of cut-and-paste behavior for a tree editor and for Verdi (not discussed in this paper) were fairly straightforward, specification of the behavior of the flow chart editor required over 200 lines of new code. This pushed the limits of the behavior of Escalante and was of an order of difficulty not envisioned when Escalante was designed. Further enhancements should make Escalante more usable as an environment for generating editors for visual programming languages.

References

[1] Arefi, F., C. E. Hughes, and D. A. Workman, "Automatically Generating Visual Syntax-Directed Editors," *Communications of the ACM,* 33:3, March 1990, pp. 349-360.

[2] Bier, E.., and C. Stone, "Snap-Dragging," *Computer Graphics,* 20:4, August 1986, pp. 233-240.

[3] Bohl, M., *Flowcharting Techniques,* Science Research Associates, Chicago, 1978.

[4] Citrin, W. V., "Requirements for Graphical Front Ends for Visual Languages," *Proc. 1993 IEEE-CS Symposium on Visual Languages,* Bergen, Norway, August 1993, pp. 142-150.

[5] Graf, M. L., "A visual environment for the design of distributed systems," in *Proc. 1987 IEEE-CS Workshop on Visual Languages*, Linkoping, Sweden, August 1987, pp. 330-343.

[6] Hudson, S., "Adaptive Semantic Snapping - A Technique for Semantic Feedback at the Lexical Level," *Proc. CHI '90,*, Seattle, April 1990, pp. 65-70.

[7] Lehner, J. K., *Flowcharting: An Introductory Text and Workbook,* Auerbach Publishers, Princeton, NJ, 1972.

[8] McInerney, T. F., and A. J. Vallee, *A Student's Guide to Flowcharting,* Prentice-Hall, Englewood Cliffs, NJ, 1973.

[9] McWhirter, J. D., and G. J. Nutt, "Generation of visual language environments," in *InterCHI '93: Conference on Human Factors in Computing Systems, - Short paper session* , Amsterdam, May 1993.

[10] Van Sommers, P., *Drawing and Cognition: Descriptive and Experimental Studies of Graphic Production Processes*, Cambridge University Press, Cambridge, UK, 1984.

[11] Von Känel, J., *Cut and Paste of Complex, Interrelated Objects,* Verlag der Fachvereine Zürich, Zürich, 1992.

[12] Zipf, G. K., *Human Behavior and the Principle of Least Effort: An Introduction to Human Ecology*, Hafner Publishing, New York, 1972.