Massachusetts Institute of Technology
Artificial Intelligence Laboratory

Working Paper 219 July, 1981

Presentation Based User Interfaces

Eugene C. Ciccarelli

This research will develop a methodology for designing user interfaces for general-purpose interactive systems. The central concept is the **presentation**, a structured pictorial or text object conveying information about some abstract object to the user. The methodology models a user interface as a shared communication medium, user and system communicating to each other by manipulating presentations.

The methodology stresses relations between presentations, especially presentations of the system itself; presentation manipulation by the user; presentation recognition by the system; and how properties of these establish a spectrum of interface styles.

The methodology suggests a general system base providing mechanisms to support construction of user interfaces. As part of an argument that such a base is feasible and valuable, and to demonstrate the domain independence of the methodology, three test systems will be implemented.

CONTENTS

1. Introduction

Increasingly many programs today may be considered to be **presentation programs**: they offer a continual pictorial or text view of some application domain and allow the user to manipulate the visible domain objects and see the results. Much of their bulk and complexity results from offering the user an interface that allows great viewing flexibility. In fact, sometimes more bulk and complexity is in the user interface than in the management of domain objects. These user interfaces are complex in design and use. At the same time they are often functionally inadequate.

For example, consider any of the large display-oriented mail reader programs: the actual management of sending and filing messages is often the simple part, compared to the tasks of displaying a message, displaying summaries of a set of messages, and showing option and command documentation. Those are not simple tasks -- because they talk to a person, not a low-level operating system program.

Many programs that let a person interact with a world of domain objects are small and simple -- but inadequate. I argue that every presentation program should provide certain common capabilities generally missing in these small, simple programs. This is not necessarily a bad reflection on the designers of these programs: it is very difficult to build complex presentation facilities. This research attempts to make that job easier, by providing a methodology for designing user interfaces to presentation programs.

The methodology introduces a model of user interface mechanisms that encourages the development of systems in which many domain-presenting programs co-exist and share presentation mechanisms. This system approach makes individual programs more understandable and easier to build. The importance of the **presentation system** concept grows with increases in the number of presentation programs and domains a person uses and a consequent need for uniformity among these domain interfaces.

1.1 Human Engineering Principles

Some major human-engineering principles of presentation underlie the methodology:

1. **The user should be able to see things and manipulate or reference them "*directly*", rather than be forced to talk *about* them.** The user is aware of, affects, and is affected by different kinds of objects, and these should *all* be visibly accessible to the user. Seeing what is being manipulated and seeing the effects can reduce the complexity of the user's task.

2. **The user should be able to see what the system is doing, has done, will do, and is capable of doing, and to see what it knows.**

3. **The user should be able to see objects in *different ways* and be able to specify *when* objects are to be presented.** This allows a user to adapt a presentation style to a particular problem, personal tastes, or personal problem-solving style.

4. **The user should be able to interact through presentations.** Presentations should be a *shared medium* of communication between user and system. The language used to talk to the system and the language it uses to talk back should overlap as much as possible. This reduces the complexity of the interface, as well as increasing the power and efficiency of user interaction.

5. **The user should be able to escape distractions.** The number of things a user *must* remember in order to use the system should be minimized.

This last is essentially a rule for evaluating human engineering principles. In the rest of this section I discuss some ramifications of this rule, which to me seems fundamental and easily broken, and include a discussion of the other four principles focusing on how they incorporate the design goal of minimizing distractions.

Presenting what the system is doing, for example, means the user may not have to keep a conscious mental model of the system's action and can be reassured that nothing has gone wrong. Presenting the history of interactions can function as notes to help the user keep track of the current problem solving task.

Language is simplified and generalized by allowing the user to indicate objects directly (actually accessing their presentations directly), rather than be forced to describe them or denote them by names -- the visible presentations *are* names, unforgettable ones while visible. Though sometimes it may be most convenient (and therefore least distracting) to use a compact name or description for frequently referenced or very easily described objects, the user should not *have to* remember names -- or naming conventions -- for infrequently referenced ones. (The exact meaning of "infrequently" is dependent on personal tastes and abilities as well as on the particular problem and the amount of concentration it requires.)

By sharing a presentation communication medium with the system, the user is not forced to remember two different languages, one for input and one for output. Furthermore, the system's output continually reminds, briefs, or introduces the user to at least a relevant part of the language.

Using an unfamiliar part of the system, a new domain say, requires learning fewer new commands if the language is based on a domain-independent "core" language for manipulating presentation objects on the screen: objects of different domains are all presented as text or graphical forms constructed out of the same basic elements. This is analogous to a specialized human language that is an extension of standard English and its basic syntactic mechanisms.

Minimizing distractions does not restrict flexibility -- there may be many different possible presentation styles, for example, that *could* be considered, though the user should not *have* to consider this large list just to pick one. In fact, flexibility in presentation is necessary to minimizing distractions: a person trying to solve some problem is already using mental presentations that are natural to the problem. If these must then be translated to very different presentations that the system understands, the user is distracted. However, the goal of flexibility is a difficult one to meet; if specification of the desired presentation from a wide range of possible ones is unnatural, difficult or tedious, the specification itself becomes a distraction.

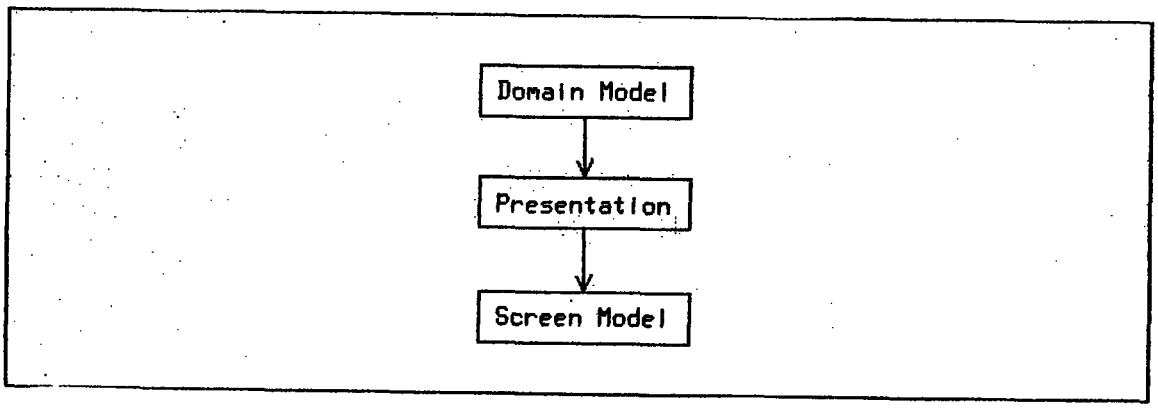
Finally, the principle of minimizing distractions argues against having separate presentation programs for each application domain, and for presentation systems containing many domains. If a person must use separate presentation programs for different domains, separate languages must be remembered. At the very least the user must remember the small but annoying and important differences that crop up when only conventions bind similar program interfaces.

1.2 The Basic Presentation Concept

This section introduces the concept of a **presentation**, a high-level model of output. The rest of this paper proposes a methodology for designing a user interface by organizing it as a system of presentations. The presentation concept is drawn from work done at Bolt Beranek and Newman by Norton Greenfeld, Martin Yonke, and Austin Henderson in a project to study general, knowledge-based methods of presenting information. Recently, the BBN group has published a report of their work [Zdybel, Greenfeld, Yonke, Gibbons 81]; that paper and this discussion differ only slightly in interpretation and terminology.

Note that the presentation-based system is not a *replacement* for a graphics package model (such as discussed in [Newman&Sproull73]) -- it *includes* one as a low-level component, mentioned in more detail below.

Output conveys information about some domain of interest to the user. Any kind of output conceptually contains two kinds of representation object (sometimes several levels of each kind): abstract information-conveying objects and objects that realize them. For example, consider the following diagram displayed on a terminal:



Here the abstract information-conveying object is the diagram which comprises text, boxes, and connecting arrows. It is clear to the user that the boxes comprise lines, as do the arrows, and the text comprises characters. This diagram presents some ideas about presentations connecting domains with display screens; these ideas are the abstract domain objects being presented. The user can thus visualize the abstract idea in this diagram. (Sometimes domain objects, e.g. Lisp objects, do not seem so abstract. The difference in tangibility does not concern the user-interface system: the idea is intangible, but its description is not.)

When the diagram is displayed on a bitmap graphics terminal, the lines and characters are realized by another level of representation: pixels in the bitmap. On a terminal without graphics vertical-bar and hyphen characters may be used to realize the lines, and these in turn may again (at a still lower level, inaccessible to the computer) be realized by pixels.

Non-graphic output also has levels of information-conveying and realizing objects. Consider a Lisp interpreter printing a Lisp object: text is the information-conveying form, and it has conventions concerning the use of parentheses, indentation, and other text mechanisms that allow the user to visualize this text as a structured Lisp object. The levels and conventions are more complicated (also more controversial and less explicit) when Lisp functions are presented as text.

In any kind of output there are three levels of object that concern a user interface system: abstract **domain**

objects, abstract **display forms** that are used to present domain objects, i.e. convey information about them so the user can visualize them, and **device-specific features** used in realizing display forms. The user conceptually recognizes the first two kinds of objects; these are the objects of discussion and the objects of the "language". The user is not generally aware of device-specific features; if the user is aware of them, they are probably distracting, e.g. it is obvious that the diagram's boxes are constructed from text characters. The job of the box-characters is to construct a box -- immediately recognizable as such and as nothing else.

In my model of a user-interface system, domain objects and display forms are described in a semantic network with one collection of processes that watch the domain's world and update its description, and another collection of processes, **screen managers**, that watch the display form description of the screen and realize this in terms of device-specific features. I do not plan to study the operation of screen managers; these simply invoke standard **graphics package** functions. [Newman&Sproull73]

A **presentation** is a display form that is conveying information about a domain object, a "view" of that object in a wide sense. A presentation may contain other display forms that are presentations, either strictly (as a part) or as attributions (e.g. a display form connected to this one); I shall call a hierarchically contained presentation a **subpresentation**. Properties of a display form may be presentations too: the color of a road line in a map indicates the size of the road; the position of map symbols indicates the position of the real world objects.

Not all the parts and properties of a presentation are presentations: some serve to improve the communicativeness of the whole presentation, by more clearly defining the kind of display form or making it more visible. For example, the boxes that surround text in the diagram above serve to visually identify the contained text as all part of *one* display form, as opposed to different pieces of text that happen to be adjacent; the boxes allow quick visual grasp of the structure of the diagram. These non-semantic parts and properties of a presentation are part of its **template**, rather than its **content**: the box is the template; the contained text is the content. Another simple example of a presentation template is the common in-text bibliographic reference style showing the author's last name, the publication year, and surrounding brackets, e.g. "[Carroll72]". The brackets (and the specification for including the text) are the background template, and there are two attached content presentations, "Carroll" and "72". This template has a little flexibility in content style: the author's name is sometimes abbreviated or replaced by a topic name; the year may be two or four digits. But the template itself is rigid and clearly recognizable.

1.3 A General System Design Methodology

The thesis of my research is that the **presentation** can serve as the central concept of a general user interface design methodology.

The methodology models user interfaces as a system of presentations. Certain processes convey information to the user by *creating or changing* presentations of domain objects or actions. Others convey information from the user by *recognizing* presentations in user-constructed display forms. A semantic network describes domain objects, display forms, and presentation decisions. The user manipulates display forms through the use of an editor, the user's extension inside the system.

I have described the general problem as twofold: first, presentation programs are common and hard to build, and second, users face an increasing need for some sort of interface uniformity, at least in philosophy, among different presentation programs. Both of these problems apply not only to switching from one domain application to another, but also to switching to a newer version of a single application program made as new

interface styles are developed and more intelligent presentation or recognition capabilities are added. Both kinds of changes represent active research areas, and any serious application will be affected by them, undergoing incremental changes or replacement. The proposed methodology helps builder and user, providing a design aid and a philosophy that encompasses a range of applications and interface styles.

While there will always be application programs whose user interfaces have good human engineering as well as rich functionality, I think most application programs in use will be systems with unexceptional capabilities since these will be produced under economic pressures to provide what functionality is necessary as soon as possible. Even the well-done programs cause trouble: other programmers are influenced by the good ideas and try to build similar programs -- and find it difficult to get right. To have an impact on the great bulk of computer users, it must be *easy* to build a good user interface.

I doubt the proposed methodology eases this job enough to affect most computer users; however, it might be able to foster the development of a "prefabricated" user interface support mechanism, *a standard presentation-system base for building application programs.* Such a base would contain domain-independent parts of the user interface system, standard processes for presenting parts of the system, and common mechanisms needed for domain-specific presentation programs. To this base the domain builder would add all the domain-related parts for the particular application. There would be no need or chance to skimp on system presentation or editing facilities since they would already be provided -- it would in fact be *easier* to build the application system out of this base than to build it separately and skimp. The methodology thus becomes subsumed by a programming environment.

The EMACS editing system [Stallman81] supports this notion of a standard presentation system base: EMACS has been successful in providing a range of applications within it (e.g. mail readers, aids for editing several programming languages, a facility for viewing tree-structured documentation). This is largely due to the ability EMACS has to provide a general base on which to build application programs with relative ease. The programming language for EMACS extensions (ITS Teco) has some deficiencies; most serious are its lack of list data structures and the fact that it can only be interpreted. However, my experience in building application programs for EMACS has been that these deficiencies have been far outweighed by the advantage of having the support facilities, especially "presentation" and environment support.

A domain-independent user interface helps the user as well as the designer, by adding consistency among interfaces to different domains. In addition to the philosophy, or model, of the user interface being consistent from domain to domain, many system presentations and the basic display form editor capabilities (embodying general display form knowledge) remain the same. Since the editor has general commands to manipulate display forms, the user does not have to learn a host of special commands for manipulating domain specific presentations. The domain application may offer commands that are especially convenient for manipulating that domain's presentations, but the general display form editing commands will suffice. Rather than determining special object manipulating commands that affect a certain transformation, the user can in effect draw a picture of what the result should look like.

1.4 Relation to Other Research

Other research projects are studying problems of presentation programs. By noting the differences between the goals of these projects and mine, the reader can better understand the aims of my methodology:

There is much interest, for example, in developing presentation styles and methods for particular domains, or to better aid users with particular levels of experience, especially novices, infrequent users, and experts.

([Bolt79], [BOXER81], [CCA79], [Moriconi80], and [XeroxStar-Seybold81].)

My interest is in providing a methodology that lies below these different presentation styles, approaches, and domains -- a lower-level but more general methodology that encompasses them -- and attempting to find some common presentation support mechanisms between such alternative styles. With such a methodology, a designer may more easily *offer, experiment with, develop, and compare* different presentation styles. The methodology should serve as a *framework* within which different styles can be viewed as tradeoffs between different principles and user values as implemented with the common mechanisms. This follows the modularization principles of *separating policy from mechanism* and *isolating common mechanisms.*

The BBN group is also emphasizing domain independence. They are developing methods for intelligently constructing presentations (for example, a map or a table) from a domain description, and consequently study the way maps, tables, charts, and other presentation styles convey information and when they are most useful [Zdybel, Greenfeld, Yonke, Gibbons 81].

What is original in my work is the way the presentation concept is used to organize and understand a user interface constructed as a *system of presentations.* For example, different styles of interfaces can be partly understood by seeing what kinds of presentations the user is allowed to create (section 2.5 discusses such alternatives). My aim is not to discover how to build programs that intelligently create effective presentations; it is to show how to organize a system of such programs and provide domain-independent mechanisms to support the interactions between them. However, some techniques for creating a presentation will be studied, namely techniques which create a presentation by interrelating existing presentations; these techniques are relevant since they contribute to and take advantage of the overall coherence of the user interface as a system of presentations.

My work and the work at BBN complement each other: a complete domain-independent presentation system base not only needs a system-level organizational methodology such as I am proposing; it also needs, fitting into this framework, powerful presentation programs that embody general domain-independent knowledge about presenting information.

While most of my work will stress domain-independence, I will be emphasizing one kind of domain-specific presentation: presentations of the system itself, such as presenting the current state of the application program's activity. Every presentation system can include such presentations, so this domain is a special one, and a general methodology should specifically discuss how these presentations relate to others. Thus, other research investigating sophisticated presentations of programs and systems of programs, such as discussed in [CCA79] and [Moriconi80], complement my work in the same manner as the BBN work.

This proposal discusses presentations as a purely visual communication medium (e.g. using display screens and printers), having nothing to do with other user input/output media, such as sound or touch (e.g. speech, special whistles or rings, or pressure or vibration affecting a joystick). The BBN group's work on presentations has from the start assumed that presentations can take other sensory forms, but has initially concentrated on bitmap graphics as the most important and tractable medium. I have restricted my thesis to studying display presentations for largely the same reason: it is comparatively well-understood, yet rich. In addition, my interest is how presentations combine to form a user interface system, and I want to demonstrate how existing and future interfaces can fit into my model; the other media have only scattered and *ad hoc* implementations in existing systems, outside a few very recent, experimental research vehicles (such as that of the Architecture Machine group [Bolt79]).

1.5 Implementation Plan

I plan to implement three different application programs. Two special properties to be investigated are implementation problems and the methodology's applicability to different domains and interface styles. Using three domains will support the proposition that a domain-independent base can be constructed.

The scenario in section 3.2 illustrates the first application: a system to aid a user writing a paper, capable of presenting the text, outlines, and changes. One capability of particular interest is aiding *two* authors to communicate about a shared paper and see each other's changes. Currently, I do not see how this differs from what a single author needs, except in degree: a lone author can tolerate less presentation help by keeping more of the presentations mentally, but two authors are more likely to have to communicate these presentations explicitly. The two-author paper is a common problem, and it may also offer insight into a possible future extension of the presentation user interface ideas -- *user-to-user communication through presentations* (analogous to user-to-domain-manager communication through presentations). The two-author case will therefore be kept in mind and emphasized when it is different from the single-author case.

The second application is a directory manager, allowing the user to present a directory of files in various ways and having knowledge about those files, including user comments and inter-file dependencies.

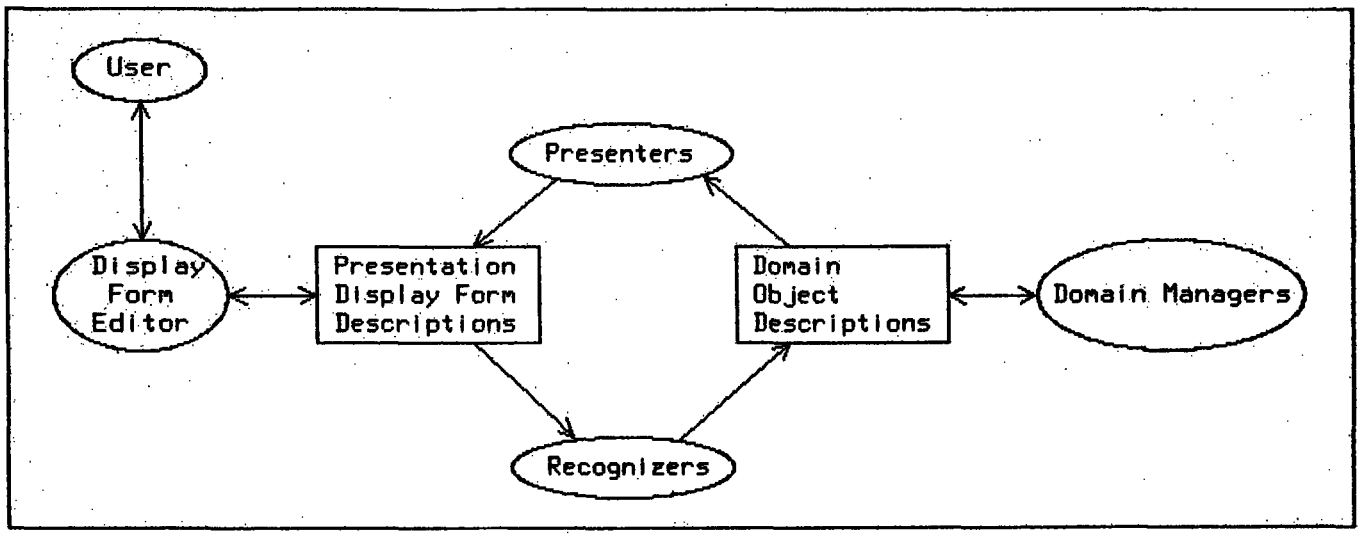
The third application is a computer-mail message manager, allowing the user to keep track of messages through various presentations. This is similar in purpose to the file manager, though the difference in domain knowledge is substantial. The similarity of purpose, however, may allow strong similarities of the interface style, perhaps of some presentations used, and thus might offer insights into simple domain-independent presentation mechanisms.

2. User Interface Design Methodology

The previous section introduced the concept of a presentation in isolation; here we consider how presentations fit together into a larger system of user-system communication.

2.1 User Interface System Components

The following schematic diagram illustrates the kinds of components in a presentation-based user interface and the basic interactions between them. Processes are shown by ellipses, descriptions in the semantic network by boxes:



For each domain there is a **domain manager** (perhaps several -- it doesn't matter here). These manipulate and observe the special kinds of domain objects, whether they are internal (as for a Lisp robotics simulation) or external (as for any computer network application). The domain manager maintains a collection of **domain object descriptions** in a semantic network. (As mentioned in section 1.2, the concept of a domain object includes relations, plans, and other pieces of domain knowledge.)

The semantic network also contains **descriptions of presentation display forms.** These constitute a conceptual model of the screen, describing the currently visible display forms and how these present domain objects.

Connecting this screen model with the domain model are **presenters** and **recognizers**, two kinds of processes deciding how display forms on the screen constitute presentations of domain objects and building the descriptions of these relations. A presenter decides when and how to create or update a presentation of some particular class of domain objects. A recognizer observes the structure of display forms created by the user and decides how that structure can be considered as a presentation of a domain object. (Note that the recognizer's job is easier -- incremental -- when the user has modified a presentation the system constructed.) The importance of this design methodology considering *both* presenters and recognizers derives from the general goal of providing an interface in which the user and domain managers *share one communication medium* -- a medium of presentation passing. The presenters pass presentations from the domain managers to the user; the recognizers pass presentations from the user to the domain managers. For many current systems the recognizers are trivial, as the class of presentations the user can construct is highly restricted.

The **display form editor** is the user's direct agent through which the user can access, reference, create and manipulate conceptual objects on the screen. The user can give the editor a wide variety of kinds of commands; this will be the topic of a later section. Occasionally the editor might communicate directly with domain managers, though this kind of interaction is not shown in the diagram above. There are two reasons for this kind of non-presentation communication: First, a domain manager might tailor the command language of the editor to make it more effective at manipulating certain kinds of presentations for that domain. Second, the ability of the editor to directly signal a domain manager is one way of modeling whatever commands or signals the user gives that go to the domain manager *not* through presentations. I will hereafter ignore that kind of input and pretend that all input is in terms of presentations to discover the generality and limitations of this model.

There is another kind of simple signal not shown in the diagram, and which can take various forms: some way for the editor to signal a recognizer that a user constructed presentation is ready to be recognized and "passed" to the domain manager. This may not be required in every kind of interface, as will be discussed later in regard to different user interface styles.

2.2 EMACS and ZWEI

The EMACS and ZWEI editors will provide several examples referred to in the rest of this proposal. [Stallman81] [Weinreb&Moon81] This section introduces them briefly, concentrating on those aspects involving presentation issues. EMACS and ZWEI are very similar (by design); what is said about EMACS will generally apply to ZWEI as well. Readers familiar with the use of these editors and their subsystems DIRED and Edit Options should at least skim this section to be sure they are familiar with these issues.

The most important presentation aspect of EMACS is **redisplay**, the automatic updating of the screen so that it continually displays the text in the editing **buffer**. Normally, the buffer contains more text than can be viewed on the screen. EMACS displays a portion of text around the current point in the buffer.

Redisplay is an activity largely independent of the editing activity: commands that change the buffer text do not generally dictate how the screen should be updated. This not only allows a sharing of redisplay mechanism; it allows the redisplay activity to be optimized. For example, if the user gives a command that changes the current point to another part of the buffer, any unfinished redisplay of the old part of the buffer will be aborted and the new part displayed instead. This is particularly helpful when redisplay takes a large amount of time, as with a slow terminal or a heavily loaded operating system. Redisplay also tries to minimize the amount of output, typing only lines (and frequently only part of the current line) which have changed with respect to what is on the screen. Thus, after an editing command, the user typically sees only a small portion of the screen change. Redisplay has a simple model of the screen, allowing it to check whether a line of text is displayed correctly by a line on the screen. Thus the redisplay activity, buffer text, and screen model correspond to simple kinds of **presenter, domain description,** and **display form description.**

Not all of the screen is a presentation of the buffer text. A few lines near the bottom of the screen are reserved for other purposes; the most important of these is the **mode line.** This line presents information about the editing environment, for example, the editing modes that are currently turned on, and the name of the file being edited.

One EMACS subsystem the user can enter is **Edit Options**, which presents the environment in much greater detail and allows the user to tailor it by editing its text presentation. A sample Edit Options display is

shown below. It displays variables that control EMACS's operation (e.g. Comment Column), their values (32), and their documentation ("Column to start comments in"). The user edits the values displayed; when the user exits Edit Options, the values are changed.

```
Comment Begin:  "; "
        * String for beginning new comments
Comment Column: 32
        * Column to start comments in
Comment End:    ""
        * String that ends comments
Comment Start:  ";"
        * String that indicates the start of a comment
```

DIRED is another EMACS subsystem; it presents a file directory for editing. It is primarily designed to help the user delete files carefully, though it offers a few other file operations. The user edits the text of the directory listing, marking files for deletion. The files are not immediately deleted: the user is in effect editing a *plan of deletions*, by annotating the directory listing with "D"s, each marking a file to be deleted. When the user is satisfied with this plan, and confirms it, the files are deleted. Here is a sample DIRED listing, with the files +DHP 3 and +DHP 4 marked for deletion (the reader unfamiliar with this style of directory listing can safely ignore its details, though note that *only* the "D"s are DIRED annotations; all other indicators, e.g. "!" and "$" are standard parts of the listing):

```
AI   ECC
FREE BLOCKS  1=137  5=65  14=393  13=124  15=446  2=215  4=0  3=142
D 3   +DHP   3        0 +257 !  5/19/81 15:30:33  (5/19/81)
D 3   +DHP   4        0 +683 !  5/19/81 17:14:56  (5/19/81)
  2   +DHP   5        1  +20 !  5/19/81 17:49:20  (5/19/81)
  13  AR3    DHE      7 +607  $ 3/05/81 12:48:07  (3/05/81)
  13  DHP    110     17 +994  $ 3/09/81 16:11:35  (5/15/81)
  3   DHP    119     18 +224 !  5/19/81 15:26:34  (5/19/81)
  5   DHP    DICT     0 +120    5/17/81 22:03:36  (5/17/81)
  3   DHPDIA 4        0 +815    5/13/81 22:47:04  (5/13/81)
```

Unlike the commands for marking files for deletion, some DIRED commands have an immediate effect on files -- for instance, the command to move a file from a primary disk pack to a secondary one. This acts immediately and DIRED's directory listing is updated to show the file on the secondary pack. If the user had made a mistake, another command moves the file back to the primary pack. It is safe for these commands to take effect immediately because they *can be undone.*

The DIRED and Edit Options subsystems both illustrate what I believe is an *inherent ambiguity* of presentation programs, an ambiguity the designer must at least realize, if not resolve: If the user is allowed general editing capabilities, some transformations of the presentation will have more than one natural meaning. For instance, what does it mean if the user deletes a line from the DIRED directory listing or the Edit Options environment listing? Does the user intend that file to be deleted or that variable to be unbound? Or does the user simply wish to eliminate that line from view -- and from editing consideration. Both subsystems adopt the latter interpretation, as it is safer and offers the user more convenient editing. The user can protect certain files from deletion that way, for instance, or consider only a relevant subset of a directory or environment. The DIRED listing shown above was filtered to contain only lines containing the string "DH", so that only files relating to my thesis would be included. The Edit Options listing was filtered to show

only certain options.

This ambiguity is not just apparent to designers: I have been asked by users new to DIRED whether files can be renamed by editing their names in the DIRED listing. The idea is plausible and natural, given DIRED's general philosophy, though its implementation in DIRED appears very difficult at the least.

ZWEI is the EMACS-like editor for the M.I.T. Lisp Machine. One important difference between EMACS and ZWEI is their programming environments: the EMACS environment includes only the EMACS job, whereas the ZWEI environment includes the entire Lisp Machine environment. Thus, as will be discussed further in section 2.5, the user can create a Lisp function in a ZWEI editor buffer and invoke a command to evaluate (or compile) that text in the general Lisp environment. The reverse is also possible: the user can ask for the text representation of the value of a Lisp function to be inserted into a ZWEI editor buffer. The ZWEI editor can act as an interface to the rest of the Lisp environment.

In both EMACS and ZWEI the primitive (and most frequently used) commands are **single-character commands**: typing the letter "a" is a command to insert an "a"; typing Control-D is a command to delete one character. While most of the single-character commands always perform the same action, some commands change meaning (sometimes only slightly) depending on the kind of text being edited: for instance, the Tab command indents according to the programming language style if the user is editing a program source.

These single-character commands are augmented by **extended commands**, which have mnemonic names, such as Edit Options -- the command to enter the Edit Options subsystem. The user invokes extended commands by typing the command name into a **minibuffer** at the bottom of the screen. (A primitive single-character command first signals the user's intention to enter an extended command.) Any of the single-character commands may be used to edit this name as it is typed. The user signals that the command is finished (and should be executed) by typing Return. (This explanation is strictly correct only for ZWEI -- in EMACS, the user does not have the full complement of editing commands while typing an extended command. For the purposes of this proposal, this distinction is not important.)

2.3 Presenting the System

The user interface system itself is a domain that can be presented in many different ways, for various purposes. These presentations and user interactions through them constitute an important part of the user interface model: they provide an ability to see what the system is doing, to see how it is constructed, and to direct or change its operation. Here is a brief description of the various kinds of system presentations:

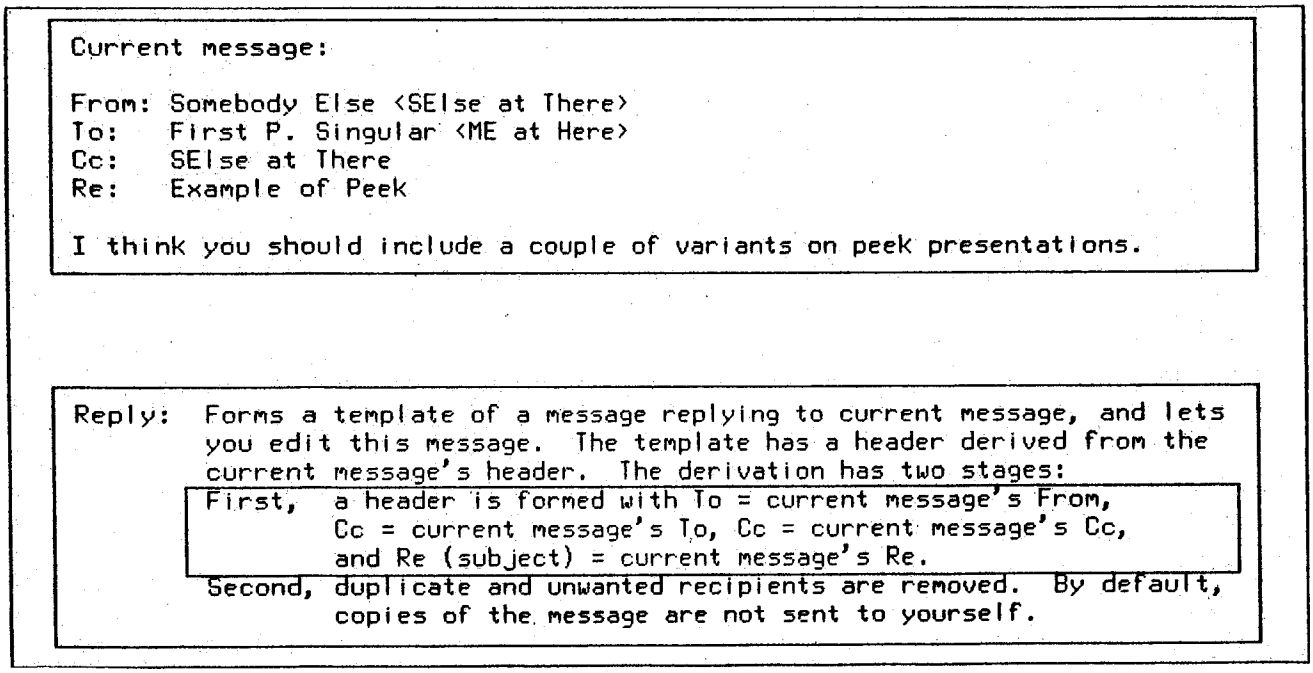
1. **Self-documentation**: The semantic network can be presented. In this class are general techniques that can present any description expressed in the semantic network's language; examples of this are formal network printers (e.g. printing a tabular or Lisp-like text representation) and more intelligent mechanisms for producing natural language descriptions of parts of the network. Also in this class are more specialized techniques that can present only certain kinds of descriptions.

2. **Activity Presentations**: These can be further classified as **history** (past actions: what happened, when, why, etc.), **peek** (current actions: what is happening, how far along it is, etc.), **plan** (future actions and goals), or **command documentation** (possible future actions) presentations. In each kind there are three important

items that can be described: input objects, the activity's programs and goals, and output objects (results). Typical activity presentations will present only some of these, depending on the purpose of the presentation, what the user is assumed to already know, and constraints of the presentation. In particular, constraints can be especially restrictive for peek presentations: the presentation may have to be computed quickly, both to avoid using resources that could be used by the monitored activity and to ensure that the peek is current and does not lag behind the activity. These constraints make it important that these presentations use and augment existing presentations whenever possible; this is frequently possible, because the user just gave a command presentation (see below) which initiated the action and is still visible (or quickly retrievable), or because the objects affected are currently presented.

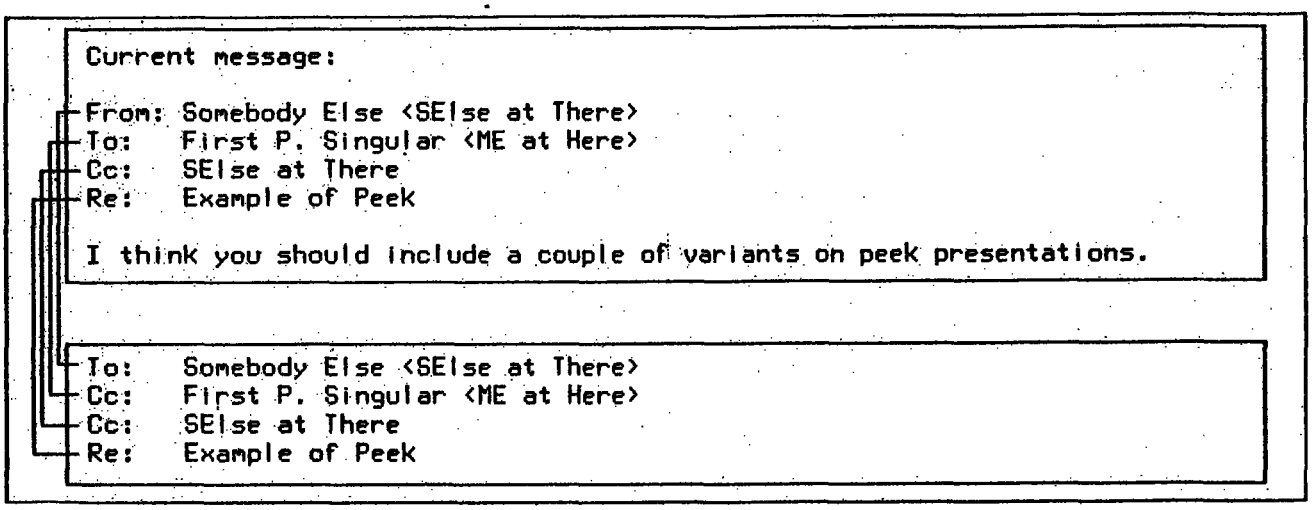
Below are two examples of peek presentations, differing in which of the three kinds of items (input, program, output) are included in the presentation. In both examples the user is viewing the current message with a mail reader program, and invokes a reply command. This forms a message template, which the user can then edit and complete (e.g. add the text of the reply). The activity of forming this template is presented in the two peeks.

In the first example the peek presents the program (by showing its documentation) and highlights the description of the current stage of the activity (by drawing a box around it); thus, the activity is currently in its first stage:

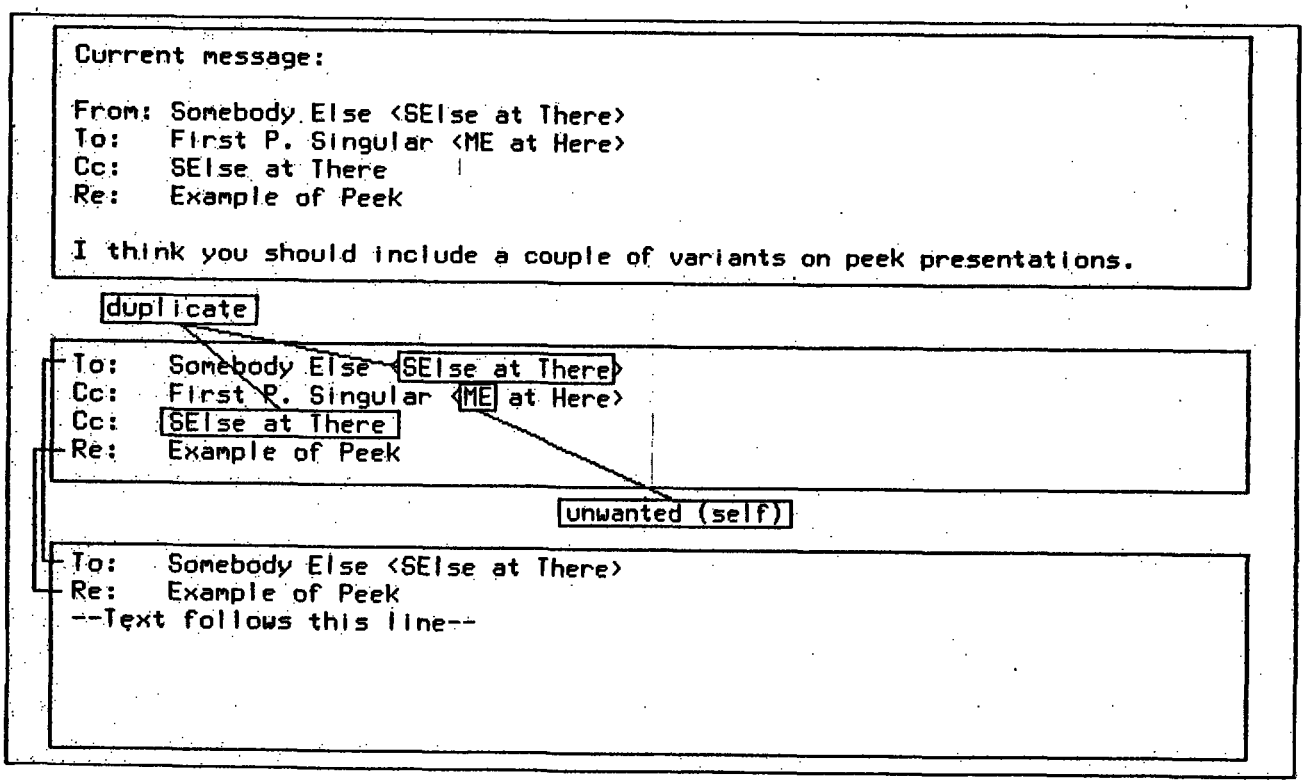


This kind of peek would be especially appropriate if the user, before giving the reply command, had first asked for its documentation. The peek presenter would notice that the program is already being presented, and might use that as the basis for its peek presentation.

The next two figures show another example of a peek presentation of the same activity. Below, the first stage of the template formation is also being presented, but by showing its output (the first guess at the template's header) and how it relates to the input (the current message). Each header field in the template has a connecting line presenting its derivation from a field in the current message.



The diagram below presents the completed second stage of template formation. The field derivations are presented by connecting lines to the fields of the first stage, since they are the input objects for the second stage. In addition, the rejected fields are presented by being labeled with the reason for their rejection.



3. **Presenter Presentations** and **Display Form Presentations**: The decisions that led to a presentation's particular form, and the kinds of manipulation expected of them can be presented. Since presentations are descriptions in the semantic network, self-documentation provides a simple form of this capability. However, this is a particular case that needs some special treatment because of its central nature in the user interface, and there are some general techniques that may be used in a wide variety of systems. One common example

is the legend for a map, which presents the map's presentation styles.

Note that a presentation of a presenter may be considered to be one kind of presentation of the **presentation**; it is not, however, a presentation of the **display form** part. Presentations of display forms are important too: first (as in self-documentation) to document the graphics "language" used and answer questions about structure in display forms that may not be visually evident; and second to construct non-semantic transformations of the presentation display forms such as zooming in on one area, rotating, or showing a Fourier transform. (These transformations can but do not have to disturb the display form: they can be shown in another part of the screen.)

Another kind of presenter presentation is showing its operation, treating it like any other activity. This is particularly appropriate when the presentation takes a noticeable time to complete or undergoes changes. The user may wonder whether the presentation activity has completed or whether the presentation is yet just a partial one. Or, when a presentation is changed, the user may want to know what parts were changed. Many current systems have **implicit presentations** for these purposes, often used but usually not consciously included in the system design. It is important to understand and catalog these kinds of presentation so that making changes will not cause unexpected loss of information.

For example, consider EMACS redisplay as a presentation update. It proceeds from top to bottom, so the user at least knows that the part of the screen above the cursor is a consistent part of the presentation. If redisplay were reordered, say to update the most important lines first, the user would lose this information, and perhaps an explicit presentation of redisplay would have to be added.

Or consider a special case: redisplay completion is frequently presented implicitly by the cursor moving to a recognizably final position, e.g. the top of the screen or the current point in the text. But this can be ambiguous, most especially if no redisplay at all needs to be done.

For example, in one EMACS mail reader, moving to the next message causes that message to be displayed followed by redisplay of the mode line, which has summary information about the message and is guaranteed to change as it includes the message number. Hence, the user can tell when the message has finished displaying by the fact that the mode line changed. And in a mail reader, duplicate messages and slow system response are common enough to cause this implicit presentation to be important. Unaware of this, I recently tried changing the order of display -- to mode line first -- so that users with slow terminals would benefit by first seeing the summary information. The loss of the implicit presentation triggered an immediate complaint.

Problems with implicit presentations seem to be bound up with widely varying terminal and operating system constraints. (Note that having *very fast* terminals can *also* cause such problems; these problems will not necessarily go away as technology improves.)

4. **Debugging and error presentations**: these are in a sense special cases of peek presentations -- the current activity is suspended and these presentations frequently are forms of detailed snapshots which do not suffer from the normal peek constraints -- and have very special (and implementation dependent) purposes. I am not sure how much of a general nature can be said about these in this thesis, as they seem largely separate and difficult areas. One aspect that may be worth identifying and studying is **user-error presentations**, reports of errors in the choice of the user's command rather than in the implementation of its execution. The error presentation is largely a matter of presenting the difference between the current situation and the intended situation, perhaps simply describing, highlighting, or referencing a particular feature or object in the current situation. The intended situation presentation could be similar to part of the command documentation presentation.

2.4 Relations Between Presentations

A study of the kinds of relations that can exist between presentations will help to clarify the notion of a user interface constructed as a system of presentations, by illustrating the kinds of relations between displayed presentations. These relations control the degree to which different presentations can be used together to present something as a whole, say by one large presentation using a subpresentation (a part) of another. The following section, which discusses command presentations (input), will discuss a different kind of interpresentation relation -- not between displayed presentations, but between classes of presentation styles.

As illustrated by the map example a large presentation may be hierarchically constructed from smaller presentations of its parts; more generally, one presentation may also use another as an **attribution**. A typical example is connection: two presentations connected together, though neither is hierarchically a part of the other. Another example is the dependence of one presentation's position on another's. A large presentation may have many presentations interrelated by attribution within it. In addition one large presentation may have (typically only a few) relations to other large presentations. Consider, for instance, two large presentations being displayed, a map and its legend. The user asks where some particular legend symbol is used in the map, and the system responds by drawing a line between the legend symbol and a use of it in the map. That connecting line is attached to instances of that symbol in both the legend and the map -- the two large presentations (and the two small ones, the symbols) are connected by this connecting line presentation.

Presentations may also be related explicitly in the conceptual description by relations other than presentations. Two typical and contrasting relations that must always be used in conjunction within a large presentation are **consistency** and **differentiation** of presentation style. Consistency is a measure of how often or how closely the same kind of display form is used to present the same kind of domain object. Differentiation is a measure of how often or how distinctly two different kinds of domain object are presented with different kinds of display form. The two are not independent: presenting every map domain object by a circle is very consistent but is poor differentiation between, say, a city and a road; similarly, presenting every individual object in the map with a randomly-chosen symbol is good differentiation but no consistency.

More importantly, the context within which consistency and differentiation are evaluated must be considered. A particular kind of large presentation style may imply or suggest styles to be used within it for component presentations; to be consistent between two large presentations may be a mistake and confuse more than help the user if each large presentation has a well established context of styles. One way of looking at this effect is to conclude that consistency and differentiation carry information not only about the kind of domain object but also about its presentation, such as the reason it is being presented, the importance it has (the most important components should be easily differentiated from the rest), and reinforcement of the user's identification of the presentation style in use within a large presentation.

Augmentation of a presentation can increase differentiation without necessarily implying decreased consistency. One example is the use of highlighting, e.g. increasing the intensity of a symbol, outlining it (e.g. changing a star to a star within a circle), or blinking it. Another is adding component structure, such as changing a circle to a circle with a name within or next to it. In both techniques the new kind of display form is readily identifiable as being the same kind of display form at some level and yet easily distinguished visually.

Another interpresentation relation is **redundancy**: the same thing presented twice, perhaps in two different large presentations using different styles. Since presentations in use are explicitly described, one presenter can notice that another has created a presentation the first can use. (Perhaps the first may even decide to do nothing further.) Taking advantage of redundancy between presentations is particularly important when time

or space is constrained, perhaps to limit resource use or perhaps to minimize the distraction of slow response or large presentation size.

For example, consider a peek presentation, continually presenting the state of some activity. The time to create and update this presentation (assume a one processor system) adds to the time the activity takes to complete. Also, it is often undesirable for the peek to use much of a crowded screen. Thus the peek is both time and space constrained; it should either be small and simple, or use as much of existing presentations as possible.

2.5 Command Presentations

Up to this point the discussion has largely been concerned with *output*; this section discusses the *input* action of the user interface system. Particular attention will be paid to the relations between presentations of domain objects and presentations of *operations* on them. Different styles of interface result from different such relations, giving the user a feel of directly controlling domain objects, at one extreme, or of indirectly controlling them by describing what should happen to them, at the other. The section will end by considering another determinant of interface style: the kinds of presentation knowledge contained in display form editor commands.

The user has some direct signals to the display form editor which direct its activity; these will be called **command signals**, the most primitive kinds of commands in the system. Through the editor the user edits a presentation which affects a domain manager; such a presentation will be called a **command presentation.** (Note that since the editor is itself a domain manager, for the domain of the user interface, the set of primitive editor commands may be augmented by command presentations; command presentations directed to the editor will be called **extended display form editor commands.)**

ZWEI and EMACS can serve as models of a display form editor, restricted in that the presentations they can manipulate are only text presentations and no graphics. A single-character command is a command signal; an extended (i.e. minibuffer) command is an extended display form editor command, a text command presentation passed to the editor. Command signals do not have to have fixed meanings to the editor: some depend on context, and an important context is the presentation being edited.

We may further characterize command presentations by relating them to presentations of the domain objects the commands affect, and there are basically two kinds of relations: a command presentation may be the same as or similar to a presentation of the object it affects, or it may be very different. The latter kind are typical in current systems, and are usually presentations of the action to be taken on the object; I will call these **abstract command presentations**. Consider a ZWEI scenario: The user creates a text representation of a Lisp object and gives the ZWEI command to evaluate it; the text is read by the Lisp reader (thus creating a real Lisp object) and evaluated, affecting some objects in the Lisp environment. In this scenario the user has edited an abstract command presentation and passed it to a domain manager (the Lisp evaluator). The Lisp reader is the recognizer for this presentation. Note that in general the text read and evaluated is not similar to a text representation of the Lisp objects affected. (On the other hand, an abstract command presentation can serve as the basis for an activity presentation of its execution, e.g. augmented by highlighting showing what step in the command is currently being executed.)

Now consider how a command presentation may be similar to a presentation of its affected object:

The most immediate and similar command presentation is one which is constantly a presentation of the

object exactly. As the user edits this object presentation, the recognizer immediately edits the object description so that it remains presented by the edited presentation.

A special case is editing a presentation, not to change that object, but to change how it is presented: the edited object presentation becomes a command to its presenter. Such editing tells the presenter that the user wishes to see the object differently, as the presentation is after editing; the relations that derive the presentation are being edited. For example while in the EMACS DIRED subsystem, the user can edit the buffer so that only relevant parts of the directory are presented, thus offering fewer distracting subpresentations and also protecting against deleting other files accidentally -- it will only be possible to delete the files which are under consideration for deleting.

A variation on this, less immediate but also using an exact object presentation: the user starts with a current presentation of an object, but edits it without the recognizer following it with object changes, creating a presentation of a new object similar to the current one. The user then signals the recognizer that the new object presentation is complete and the object should now be changed to reflect it; the object then becomes what is currently presented. Unlike the immediate command presentation, this one allows *intermediate stages* in the presentation editing during which the presentation is not strictly a possible presentation of a real object of the kind that will be affected, thereby allowing the user to manipulate the presentation display forms more easily.

An example of this style is the EMACS Edit Options subsystem, which lets the user edit a presentation of the current option variables in the EMACS environment.

Another variation is to create a command presentation which is not strictly an object presentation but is an "annotated" object presentation, the annotations being small abstract command presentations. Again there are immediate and delayed styles of this kind of command presentation.

An example of this style of command is the EMACS DIRED subsystem for marking files for deletion: the user edits what starts out as a directory listing, putting "D"s at the beginning of each line whose file should (later) be deleted. When the user signals that the command presentation is complete, DIRED makes a list of the files that ended up marked with "D"s and offers to delete those files.

2.5.1 Knowledge used by the Editor

The degree of similarity between a command presentation and its affected object's presentation not only establishes a range of interface styles -- it also establishes a range of requirements on the different kinds of knowledge used by display form editor commands. If a command presentation is to constantly present a domain object, editor commands must have **knowledge about presentations**, in particular about the presentation style in use, to be sure that only display forms of that style are constructed. On the other hand, if a command presentation is allowed to have intermediate stages, during which display forms do not have to present anything, editor commands need only have **knowledge about display forms**, and need know nothing about presentations, leaving it up to the user to be sure that the display forms eventually fit some presentation style.

Knowledge about display forms provides *generality*, since commands using this kind of knowledge can construct or manipulate any kind of presentation, for any application domain. These commands have a general understanding of the kinds of display forms that the system provides, how composite display forms

are constructed from more primitive display forms, how display forms may be manipulated by changing various attributes such as position or size. Some may take advantage of special knowledge appropriate to particular display forms, such as ways to take advantage of spatial arrangement to find a reasonable attachment point for a connecting line. The domain builder thus is freed from having to provide special editing facilities as long as the intermediate-stages style of interface is acceptable; the domain builder need only provide domain descriptions and ways of presenting them.

But by additionally providing display form editor commands that use knowledge about presentations, the domain builder does more than just enable the development of other interface styles: the user gets *convenient editing tools*, which ease the task of creating or manipulating particular presentations. Such tools are certainly necessary for a constant-presentation style; but they are also very convenient for the intermediate-stage style -- not necessary, but convenient. To use them the user must keep track of more commands than if only general display form editing commands are used, but for the user who has to edit certain kinds of presentations frequently, that is no problem: frequently used commands are easily remembered. (Having the generality of commands with display form knowledge is vital for the opposite case: for those presentations which are new or infrequently used.)

Three ways of solving a typical EMACS editing problem will illustrate these two kinds of command knowledge. The problem is how to delete the text representation of a Lisp object at the current location in the text of some source code. (These same three kinds of solutions apply to deleting other kinds of presentations that EMACS is aware of, e.g. text paragraphs.)

One way to do it uses only general editing commands: the user "marks" the current position (the beginning of the Lisp object's presentation), moves to the position in the text that the user knows is the end of that presentation (moving there by forward-line and forward-character commands, say), and finally uses a general editing command that deletes text between the current location and the marked location.

Another way is to use the special EMACS command for deleting Lisp object presentations. This command uses knowledge about how Lisp objects are presented as text. It is a convenient, frequently used command for those editing Lisp code.

A third way is a hybrid: use a special EMACS command for "marking" the Lisp object's presentation (it "marks" the current position and then moves forward over the Lisp object's presentation), but then use the general command used in the first method for deleting marked text. The Lisp object marker uses presentation knowledge, but the deleter does not. This method extends to other operations on Lisp object presentations: use the special marker to identify the presentation, and then use a general editing command that does something to the marked text. This method allows the user to minimize the number of special commands used or remembered, which may be desirable in infrequent situations, while still retaining the ability to refer to the whole presentation at once.

The need to remember per-presentation-style commands can be eliminated entirely in the kind of system proposed here, if the system created the presentation or if a recognizer has already determined its extent: the display form descriptions indicate which display forms constitute the presentation, and thus the display form editor can offer a *general* command for marking (for instance) a presentation's display forms. Using the example just discussed, this command can see that several lines of text (perhaps just parts of some of the lines) taken as a whole are a presentation of a Lisp object -- and that same command can mark paragraphs and other presentations, without needing to know the syntax of those presentations. Another general command can be offered for deleting one presentation's display forms.

Another use of command knowledge is the augmentation of display form descriptions. During creation or editing of a presentation the display form editor adds to the description of the screen's display forms, based on the kinds of editing commands the user has been giving; these descriptions help presentation recognition -- whether concurrent or delayed -- by offering more information than that needed just to draw the display forms. Again, this can be general display-form information: using an EMACS-like "center this line" command would describe the line as centered while the "indent under" command would describe a certain point as being indented under some other text and perhaps also add that it now happens to be centered. Or it can be presentation information: the "fill paragraph" command, which understands how paragraphs are presented as text, would include information that the affected text is a paragraph presentation (that might not have been known before, especially if it were created by the user or is an ambiguous presentation -- Lisp "code" can contain text paragraphs, inside a string object, for example, or in large comments).

These descriptions of changes in display forms help the recognizers: they can get an idea of the user's purposes by the commands used since some commands come "ready made" for a particular purpose and kind of presentation. They are thus implicit declarations by the user of the kind of presentation being manipulated. Other commands may only carry display form information, but can still help the recognizer: if the user copies some part of a presentation to another presentation, that suggests that the copy is also a presentation of the same kind, perhaps even of the same object. Similarly if some display form is moved within a presentation, it suggests that it is still the same subpresentation. The user may also use commands explicitly for declaring presentation (or just display form) structure -- examples of commands that have general knowledge about the operation of the presentation system. Explicit presentation declarations are command signals to the recognizer that reference display forms.

Some domains could have recognizers which are intelligent enough to know when a presentation is "done", having watched and incrementally recognized it during construction. This would probably require a carefully constructed presentation style and conventions concerning the domain actions taken; for instance any presentation which appears complete and implies some action to be taken may have to require that that action be reversible, in case the user further edits the presentation.

3. Implementation

My research will include the development of some test systems with different styles of interface and different application domains. These will help support claims to the degree of applicability of the user interface design methodology and study problems in its implementation. This section will give some idea of how these test systems will be implemented. The first part introduces a technique for creating presentations, something which is not strictly the concern of this thesis (which is to study how such presentations are used to form a user interface system), but obviously must be addressed in any system with a presentation based user interface. The second part explores a scenario of one of the test systems in use. The scenario shows a system to aid a user writing a paper; the system includes various presentations of the paper's text, outline, and changes. The scenario serves in two capacities: to clarify some of the principles of the methodology and the implementation techniques of this section; and to give an idea of the actual implementation that will be constructed in this research.

3.1 Presentation Frameworks

Large presentations will be built from a large standardized (i.e. "canned") presentation, which I will call a **presentation framework**, plus a a flexibly chosen set of smaller presentations that are combined with the larger framework; I will call these **attachment presentations**. (I will sometimes abbreviate these terms to just **framework** and **attachments**.) By "flexibly chosen", I mean to suggest that the attachment presentations may be individually chosen for particular purposes, with some care, from a variety of presentation styles, and that they may be attached to any subpresentation of the framework, to fit the situation. A presenter has the flexibility of varying the amount of consistency and differentiation in the set of attachment presentations, whereas those properties of the framework alone are fixed. Attachments can either be subpresentations (adding detail to the presentation framework) or presentations that are attached to two different frameworks, relating them. A presentation framework is itself a presentation, and may be all that is needed to serve some purpose, especially a very common one requiring limited flexibility. The framework gives the attachments something to contrast against, refine, provides context, and gives the whole presentation overall structure, and coherence. In addition, a framework might supply heuristics for evaluating framework-wide properties such as consistency or differentiation, and it might supply suggestions for preferred attachment styles. Deciding where to place subpresentations of a large presentation are often difficult if a large presentation is constructed from scratch; the presentation framework scheme eliminates much of this difficulty by restricting the choices: attachment presentations take their positions (roughly) from the subpresentations of the framework that they are attached to.

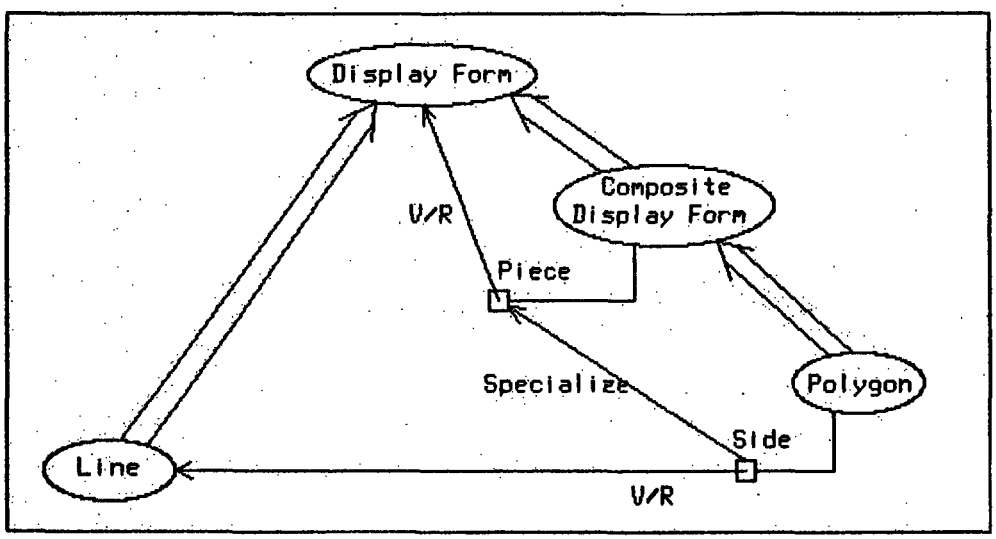
Proofreader marks and other penciled editorial comments on a draft of a paper are an example of attachment presentations (the marks and comments) on a framework (the paper). The author and formatting program have already gone to a lot of work to present the ideas in the paper as effective text. The reader can present descriptions of changes in or comments on the ideas or the text presentation of them with relative ease by such marks. Placement of the attachment presentations is simple: a comment about a sentence to be changed goes near that sentence. And there is no need to decide on an order for these presentations: the order comes from the presentation framework.

Furthermore, this example illustrates that these sorts of attachment presentations for paper changes are in fact much more general: comments, deletions, and so forth may be attached to other kinds of large presentations, even non-text presentations. A map might have penciled comments attached to its various features. Or a diagram might be commented and parts crossed out in a manner very similar to that of

marking edits to text.

From the point of view of my implementation, a presentation framework is "canned": it can be a "canned presentation", as for example an already drawn diagram, or it can be a "canned procedure", an algorithm for presenting, for example a text filler. In either case, though, no real reasoning performed. The framework is the presenter's way of letting some other intelligence (in this case human) do the hard work. The presenter uses the resulting presentation, but without understanding the decisions behind it. It *does* know what the parts are presenting, though; for instance, it would know that a collection of lines presents a paragraph of a paper, and that a larger collection of these lines presents a section.

For an example where the ability to assign some of the presenter intelligence to a person can be very helpful in a difficult task, consider drawing a part of a semantic network; a small example is shown below. This style is used at Bolt Beranek and Newman for drawing the semantic networks of the KL-One language.



The style of these diagrams typically has a complicated set of conventions. For the style shown here, each conceptual idea is presented by the concept name within an ellipse. Thick arrows present the concepts' inheritance structure. The attribution structure is shown by thin arrows coming from small boxes placed near their concept ellipses; various properties of an attribute, such as its name, are indicated around the box. Arrows may be labeled as to the kind of relation they present (e.g "V/R" (value restriction) and "specialize" in the above diagram).

Typically, much of the finer detail is left out (e.g. arrow labels and attribute properties), but even so, considerable skill is needed to decide how to place components of large diagrams so as to emphasize the kind of knowledge represented by that portion of the semantic network and make the diagram readable. If the user decided where to place the ellipses and boxes, the presentation system could fill in much of the more local detail and the connecting arrows. The user could experiment with the presentation by moving the ellipses and boxes around, while the system observed and recorded the placement without understanding the presentation reasoning behind it -- but always able to flesh out the diagram with the finer detail. Furthermore, since the system can handle the bulk of the detail and can postpone doing so, the user can edit a much simpler diagram -- first just the ellipses, and then in a separate stage the box placement. This allows the system to be more responsive than if it continually had to deal with the complete diagram, and that in turn allows the user better concentration. Production of the diagram is made much easier and more successful than it would be either by hand or with a system that understood none of it or tried to do all of it.

In addition to adding attachments, a presenter can decide to *present only part of the framework*, so as to control the scope of the presentation or isolate a relevant part. (In the semantic network diagram example above, for instance, just the concept ellipses and attribute boxes are shown for a particular part of the subnetwork.) As the user interacts with the presenter or if the presenter decides domain conditions for relevance have changed, it may update the presentation by presenting some of these previously unpresented parts, or removing currently presented parts. The full presentation framework may thus be considered a *schematic presentation.*

The presentation framework scheme also organizes the task of the recognizers. They first recognize the framework and then (perhaps examining the framework's suggestions) decide what attachment presentations it has. Once the framework is known, as for instance in the case of a system-constructed presentation, user modifications change its attachment presentations and the set of framework parts which are presented, but the basic framework type remains the same.

3.2 A Scenario

The following scenario illustrates major points discussed in this proposal concerning the design methodology and the presentation framework implementation technique. It also shows one of the test systems that will be implemented as part of this research: a system to aid an author writing a paper, allowing presentations of the paper, its changes, comments about it, its outline and its goals.

Some readers may not be interested in the implementation discussion. I suggest they skim the scenario, concentrating on the small discussions of presentation issues, which are headed by italicized sentences.

The following table summarizes the major concepts illustrated in the scenario. The numbers indicate the scenario figures illustrating the concepts. ("Etc." means that other figures also illustrate the concept, but in the same way.)

Alternate presentations: 1, 8, 19, 22.
Presentation redundancy: 16, 18, 24.
Showing only part of a presentation: 9, 12, 13, 19, etc.
User editing a system-created presentation: 6, 15.
User creating a presentation: 1, 2-3, etc.
Presentation passed to domain manager: 3, 6, etc.
System editing a user-created presentation: 7.
The user interface system as a domain:
    Presentation of activities:
        of presenters: 7, 9-10.
        of recognizers: 2, 4, 5, 15.
    Presentation of semantic network: 9, etc.
Presentation frameworks:
    Outline of paper: 1, etc.
    "English" recognizer declarations: 2, etc.
    Semantic network documentation: 9, etc.
    Text of paper: 19, 22, etc.
Attachment presentations: 2, 3, 4, 5, 9, 14, 16, 18, etc.

Figure 1: The user types some text into window W1. The display form editor creates one display form containing several lines of characters: this text is a display form, but not yet a presentation (except in the user's mind). It will become a presentation when some recognizer determines what the large presentation (i.e. the presentation framework) is for this display form, and what the smaller presentations are that are attached to the framework. In this case the user considers the text an outline (the framework) with some comments and a little of the sections' text (the attachment presentations) typed in between the section titles.

Recognition can in general be done either as the user creates a display form or later after a large display form has been created (maybe completed, maybe just at a good point to pause). The former could happen if recognizers are concurrent and smart enough to work on their own or if the user helps out continually, say by using display form editor commands that help create particular kinds of presentations. E.g. the user gives a "create an outline title here" command, which asks for the title to put there or watches as the user types a line at that point. That is one style of interface, but it can be a distracting style for the user who wants to concentrate on the ideas of the paper that are forming -- later, when the ideas stop forming (for a while!) the user will pause and tell the system what has been happening. Since I want to emphasize that both of these styles are compatible with the presentation methodology, and since the continual-declaration style is fairly common in existing systems and more obviously fits the methodology, I have decided to illustrate the delayed-declaration style here.

Note that no matter how intelligent the recognizers are, they still cannot in general fully recognize the presentation without user help -- even a person watching would not necessarily know if a sentence or two attached to an outline is part of that section's text, a comment, or a reminder to the author. On the other hand, if the user is able to delay presentation declaration, perhaps the system will get information later that will help it to recognize more than it can now. By delaying, the user may end up not having to explicitly declare as much.

Figure 1:

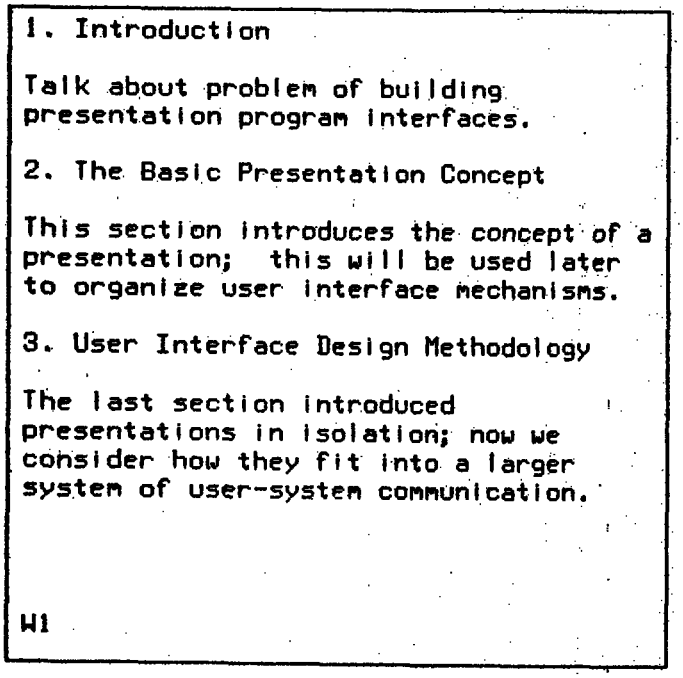


Figure 2:

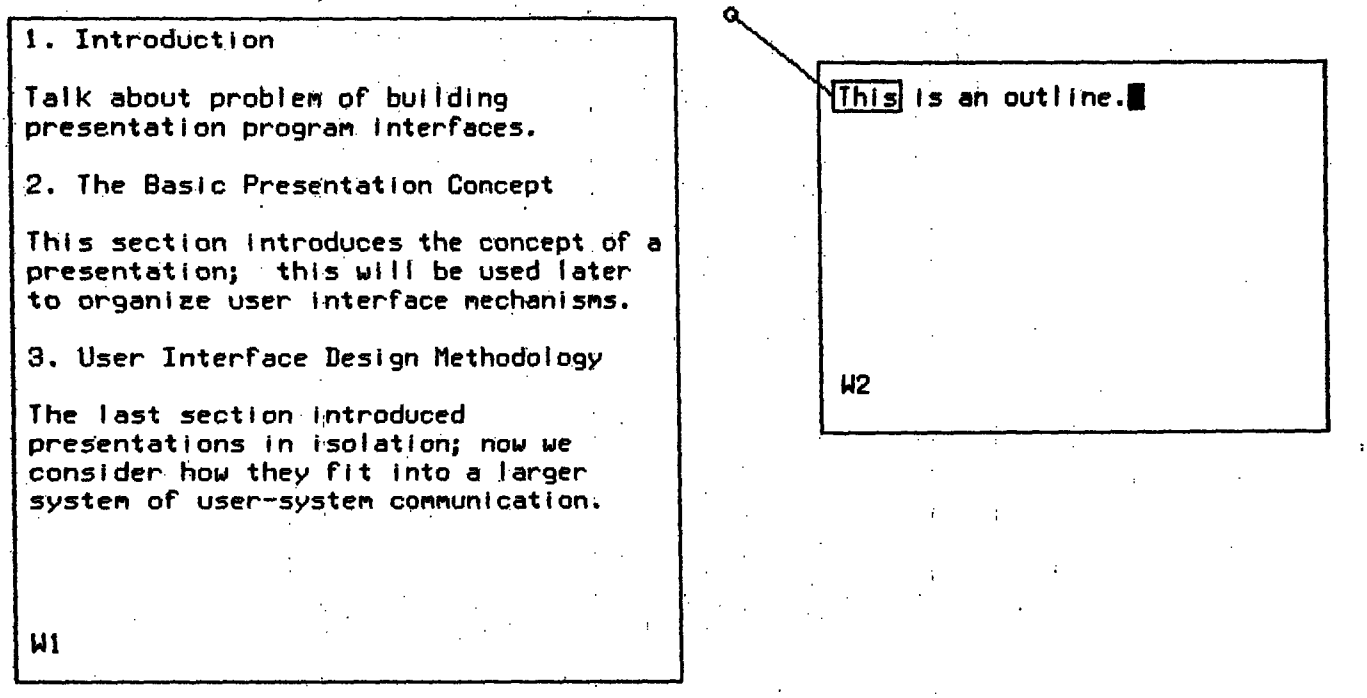


Figure 3:

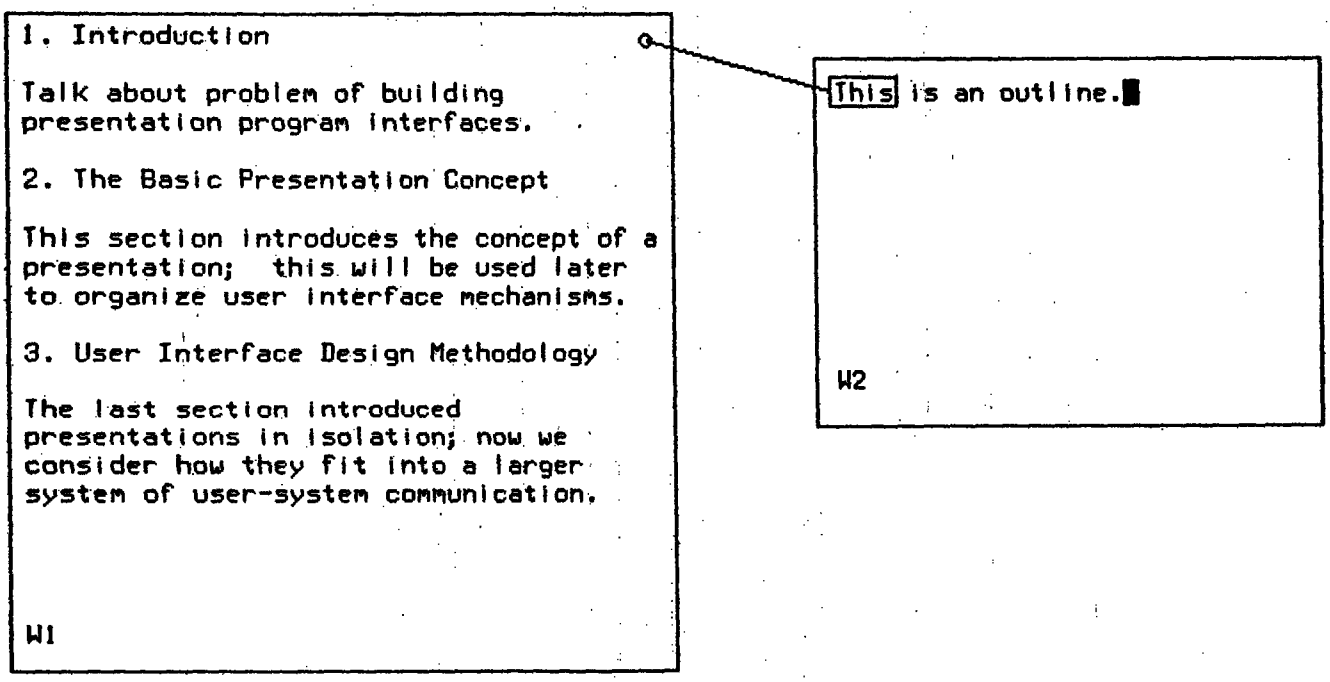


Figure 2: The user signals for an abstract command window, W2, for declaring presentation structure; W2 will contain a presentation in a stylized "English" through which the user and recognizer for W1 will communicate -- each presenting to the other in "English". (I have used an English-like presentation here for three reasons: First, some day it really could be English and this scenario serves to show that powerful recognition could be accommodated; until then it could be a simpler language's parser and generater. The fact that it seems tediously verbose might not be a concern to a fast typist new to the system, nor would it be a concern if the user could speak and have the window record what it heard. And the fact that a full sentence is displayed does not mean the user had to type the whole sentence in -- the editor can offer many sorts of abbreviation and completion aids; perhaps the user only typed "tia outline", "tia" being an abbreviation for "this is a[n]". Second, other styles that are faster and easier some people -- e.g. menus or a host of one-character commands for declaring -- are harder to illustrate here with clarity; the "English" serves as captions. And third, the difference between these methods is not critical to the issues in this scenario.)

Unlike W1, W2's presentation structure has been pre-declared by the user (by asking for this kind of abstract command window), and since the recognition of these sentences is simple and incremental, the recognizer for W2's presentation style can begin work as the user types. Note that W2's recognizer parses "English"; it is a *different* recognizer from the one (yet to be determined by the system) W1. Yet W1's recognizer is the *domain manager* to which the W2 presentations are passed. W2's recognizer helps form a channel to W1's recognizer.

As soon as the user types "This is", W2's recognizer knows that some display form referent will have to be found for "this", even if it cannot at the moment deduce it. It therefore presents this partial W2 recognizer knowledge by attaching a box and connecting line to "This" and leaving the other end (the small circle "knob") unattached, out in empty space. The knob will be attached to the referent for "This" -- the connecting line presents that reference relation.

This connecting line (with box and knob) is the first example so far of an attachment presentation on a presentation framework. (The text presentation inside W2 is the presentation framework in this case.) This general attachment style of box highlight and connecting line will recur throughout this scenario, attached to different kinds of frameworks. This style uses a spatial relation (connection by a line) to show correspondence, replacing what is commonly a temporal relation. (E.g. the user types "This is an outline" and then identifies the outline before giving other commands.) Here, for instance, the user has a choice of attaching the knob now (after "This is") or waiting until the end of the sentence -- or even later after other sentences. Perhaps if the user waits, the system might figure it out and make the attachment itself.

Figure 3: The user takes the knob, moves it over, and connects it to W1. The knob does not just overlap W1, but is actually (by display form editor command) described in the display form structure as being connected to the window, just as the box at the other end of the connecting line is attached to the string "This" in W2. This distinction of connection versus coincidental placement is important to display form editing and to presentation recognition: During editing, moving a display form will drag along any other display forms connected to it. And the W2 recognizer, which created the connecting line as a referent presentation and connected the box end to what needed a referent, now finds the other end attached to W1, implying that the whole display form in W1 is being referred to. The W1 presentation is recognized as an outline presentation framework plus some smaller attached presentations yet unrecognized. The recognizer for the outline framework analyzes the outline to find the parts of the framework, the title lines.

Figure 4:

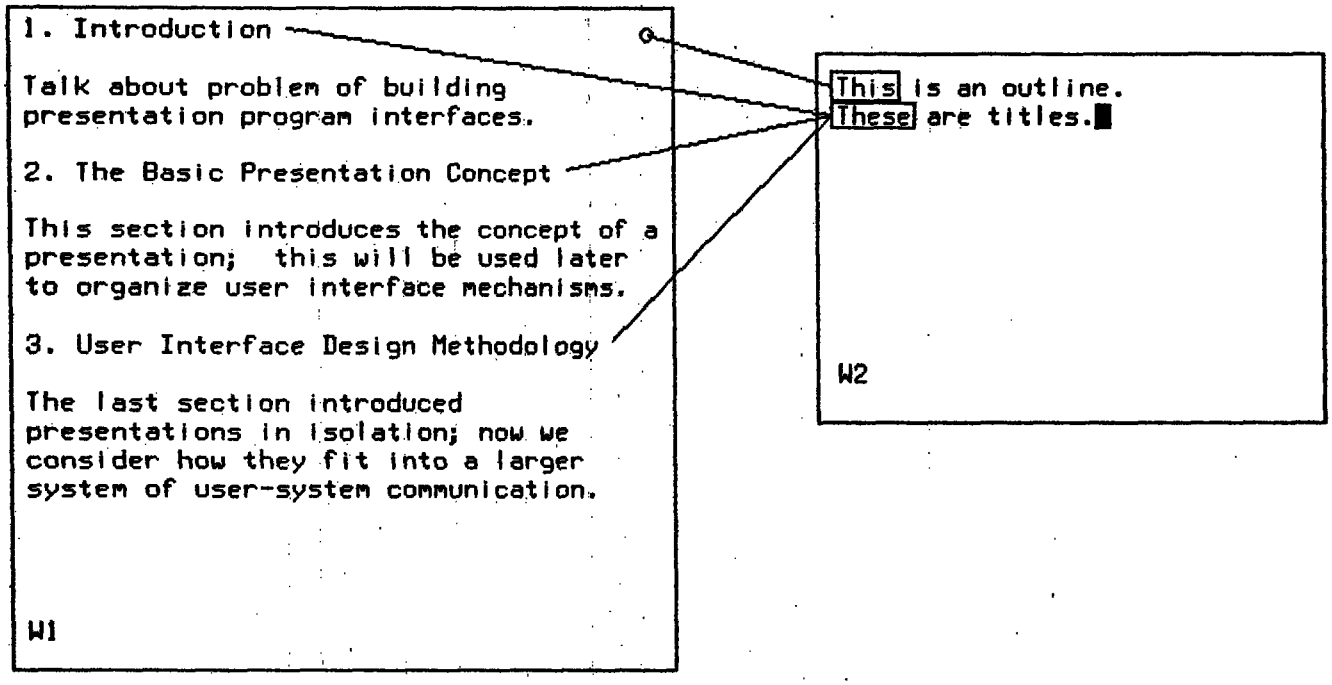


Figure 5:

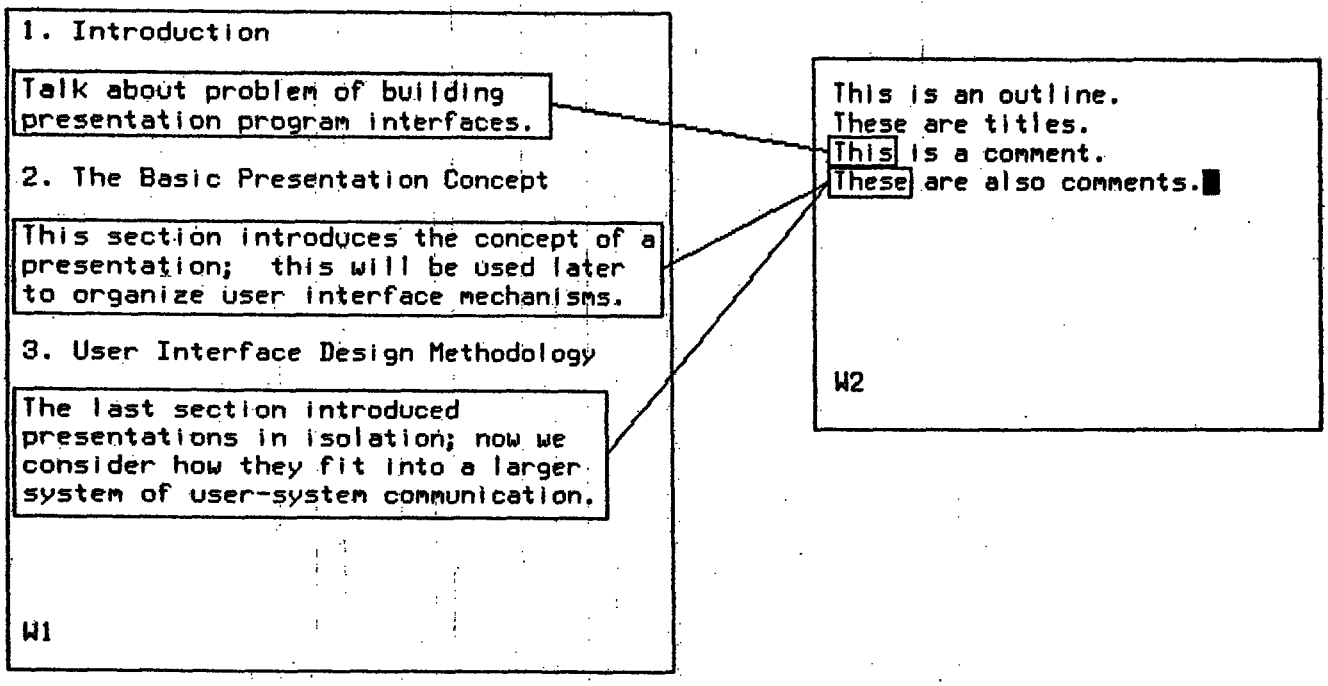


Figure 4: The W1 recognizer has determined more of the outline framework than the user explicitly declared. It decides to present this fact so the user knows that declaration of the titles is not necessary (unless the recognition was incorrect); in other words, the W1 recognizer decides to present the history of its activity, by presenting its results. Noticing that W2 is already presenting W1's presentation structure, it augments W2 -- W1's recognizer is now talking back to the user through the same presentation channel (W2) that the user has been talking through. The W1 recognizer (acting as a *domain manager*) creates a domain description -- saying that the title lines (display forms) present titles (abstract objects). The W2 presenter presents that with the sentence "These are titles", and connecting lines to the actual title lines.

Note the symmetry in W2 between what the system did and what the user did. This reflects the fact that the conversation between user and system is in a shared communication medium -- the W2 presentation.

Figure 5: The user again types a declaration ("this is a comment") and the W2 recognizer provides an unconnected knob for "this". The user creates a box around 2 lines of W1 with a editor command; the box's description includes the fact that it contains these lines. (By using a command to create a box around those two lines, the user not only implicitly specifies the box's contents but gets help in drawing the box -- the command can determine the coordinates of the corners. This is an example of a useful editor command that has pure display form knowledge -- no knowledge of presentations -- but serves an important presentation-declaring role.) The user then attaches the connecting line knob to the box.

Since an outline is a presentation framework for presenting things either contained in a section (e.g. the text of the section) or referring to it (e.g. a comment about it), the W1 recognizer decides that the indicated text presents a comment attached to section 1.1 of the paper. The W2 recognizer has been aided by the decomposition of the W1 presentation into a large, simple framework and attachments which are smaller, but more flexible in their possibilities; once the framework is known, it supplies suggestions about typical attachment presentations: a comment is presented as text after its section's title line.

The recognizer can make still more progress by assuming consistency and differentiation in the W1 presentation's style of attached presentations: it decides that the other text attachments also present section comments.

Figure 6:

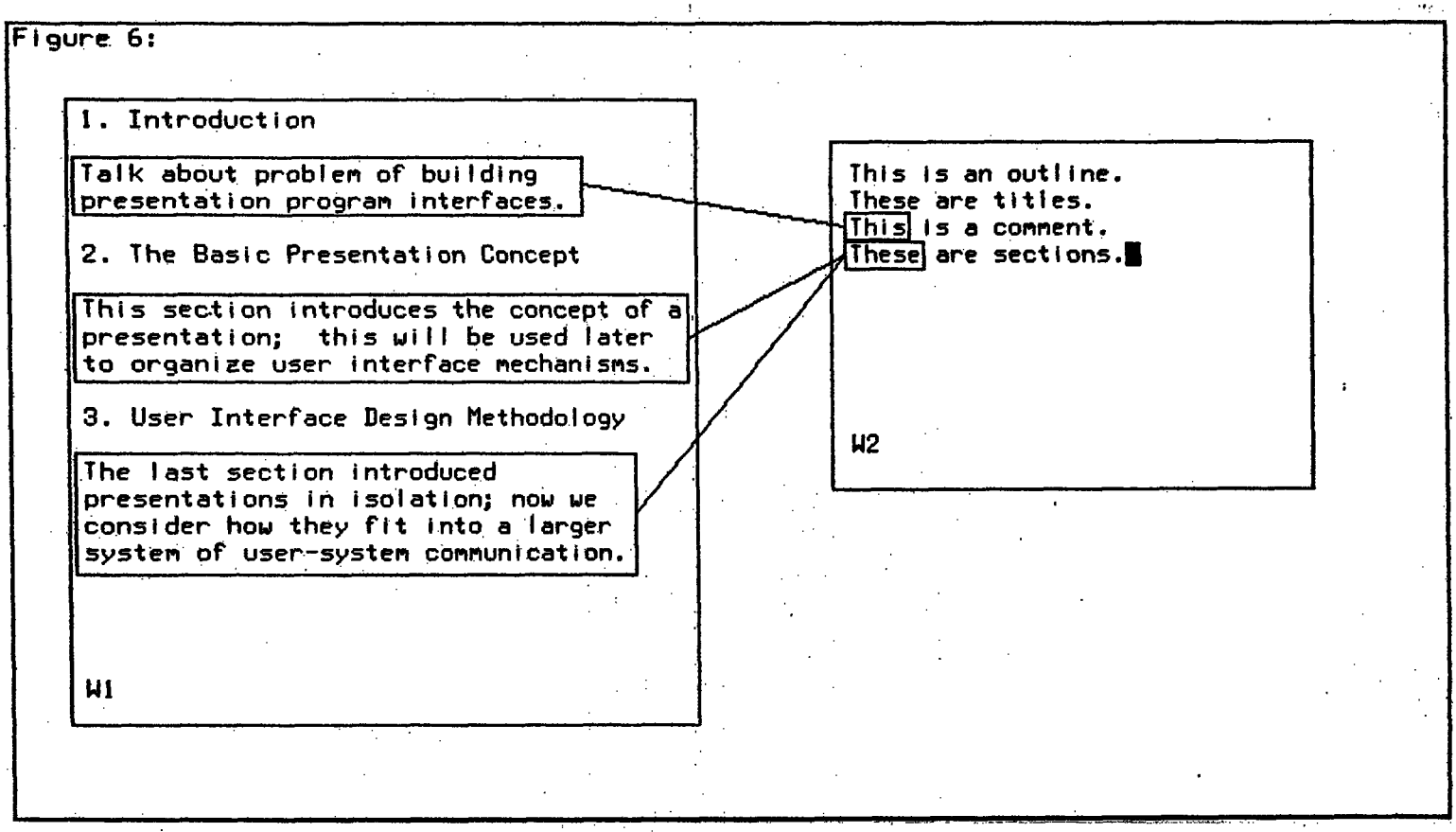


Figure 7:

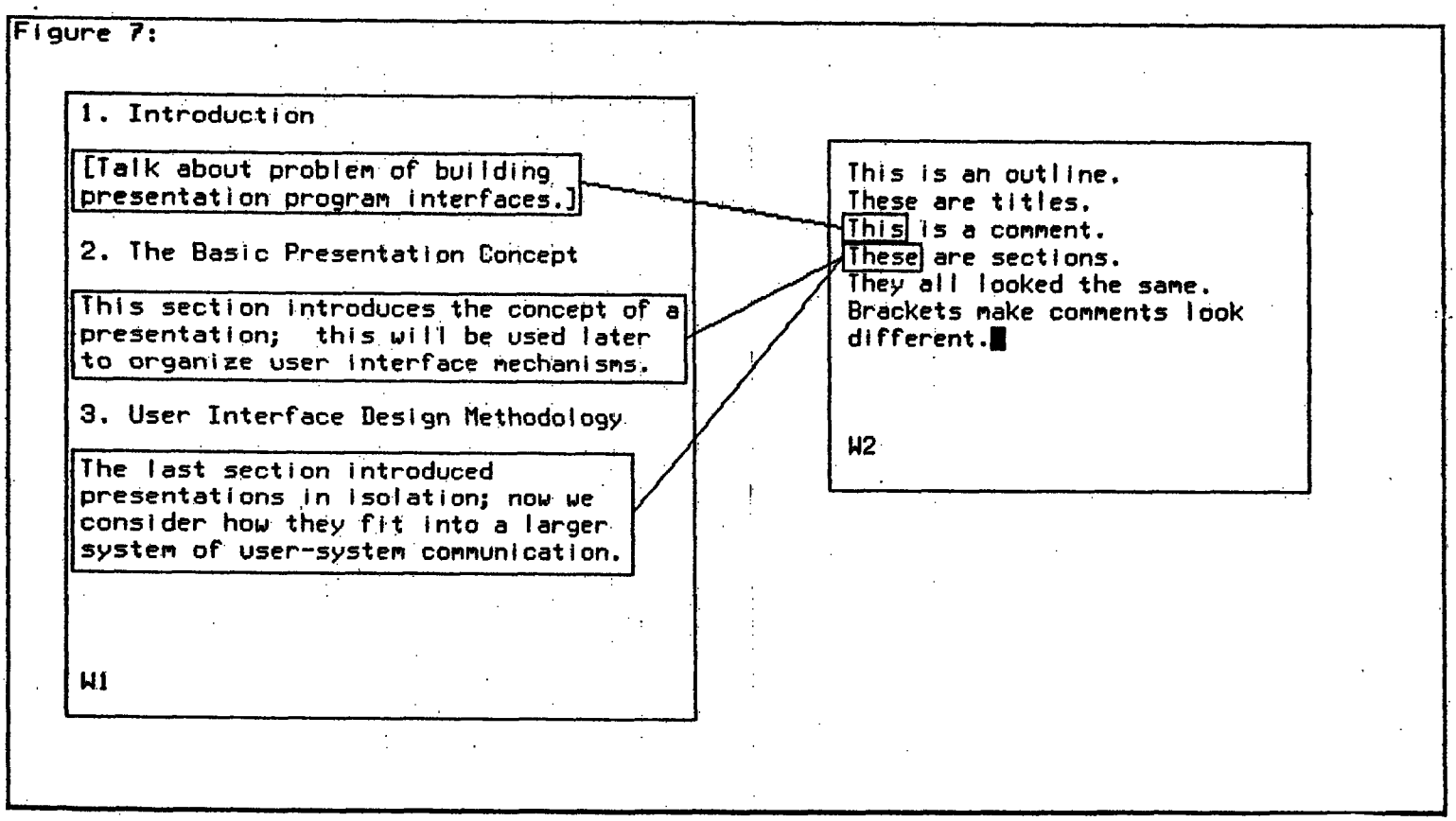


Figure 6: The system is wrong -- those are not comments, but the start of the paper's text for two sections. The user declares this by editing the system's sentence "These are also comments" to "These are sections".

This is the second example in this scenario of the user manipulating a presentation created by the system in order to present something back to the system. (The first example was connecting the unattached referent knobs.) This example shows that this ability can be used to undo system recognition in a simple way. System recognition can be a powerful aid to reducing the amount of work the user has to do, removing the distraction (and effort) of making declarations. But recognition can be a terrible distraction if the user does not know exactly what happened and does not completely trust the recognizers' decisions; even a person watching can get some things wrong -- the user needs feedback. Furthermore, since recognition will not always be correct -- it must take chances if it is to be helpful -- the user needs some easy way to undo the mistakes. (This is similar to EMACS's philosophy that gives the user very powerful single-character commands coupled with visual feedback (redisplay) and the commands to undo the effects.)

Figure 7: Since it has been corrected, the W1 recognizer presents the reasoning behind its incorrect conclusion: It assumed strong consistency and differentiation in the style of the text attached to the outline, implying that they all present the same kind of domain object. This assumption is presented by "they all looked the same".

Since only a small, probably not distracting change can increase the low differentiation, the W1 presenter modifies the user's presentation to include bracket highlighting of the comment presentation. (Perhaps the user would in fact find that change unacceptable. Deleting the brackets just added would be an obvious signal to the presenter that its decision was in error. Or perhaps the user could say "NO" in the W2 abstract command presentation.) The user and system can cooperate (albeit just a little in this case) in the creation of a presentation, if the system has some understanding of what the user is trying to create.

Figure 8: Show how much text has changed, from viewpoint of new version.

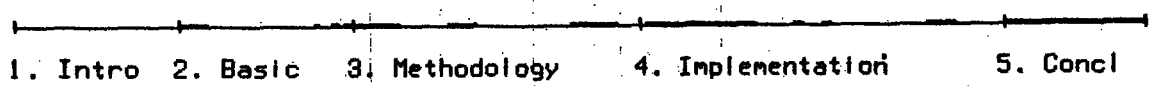


Figure 9: Show documentation on changes.

Changes have:
- author
- text affected
- position (is the position of the text affected)
- reason →○

Kinds of changes:
- addition
- deletion
- reordering
- movement
- edit

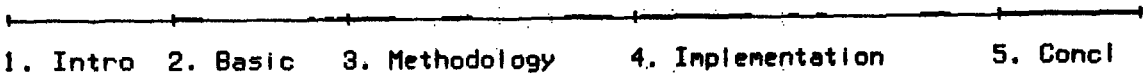


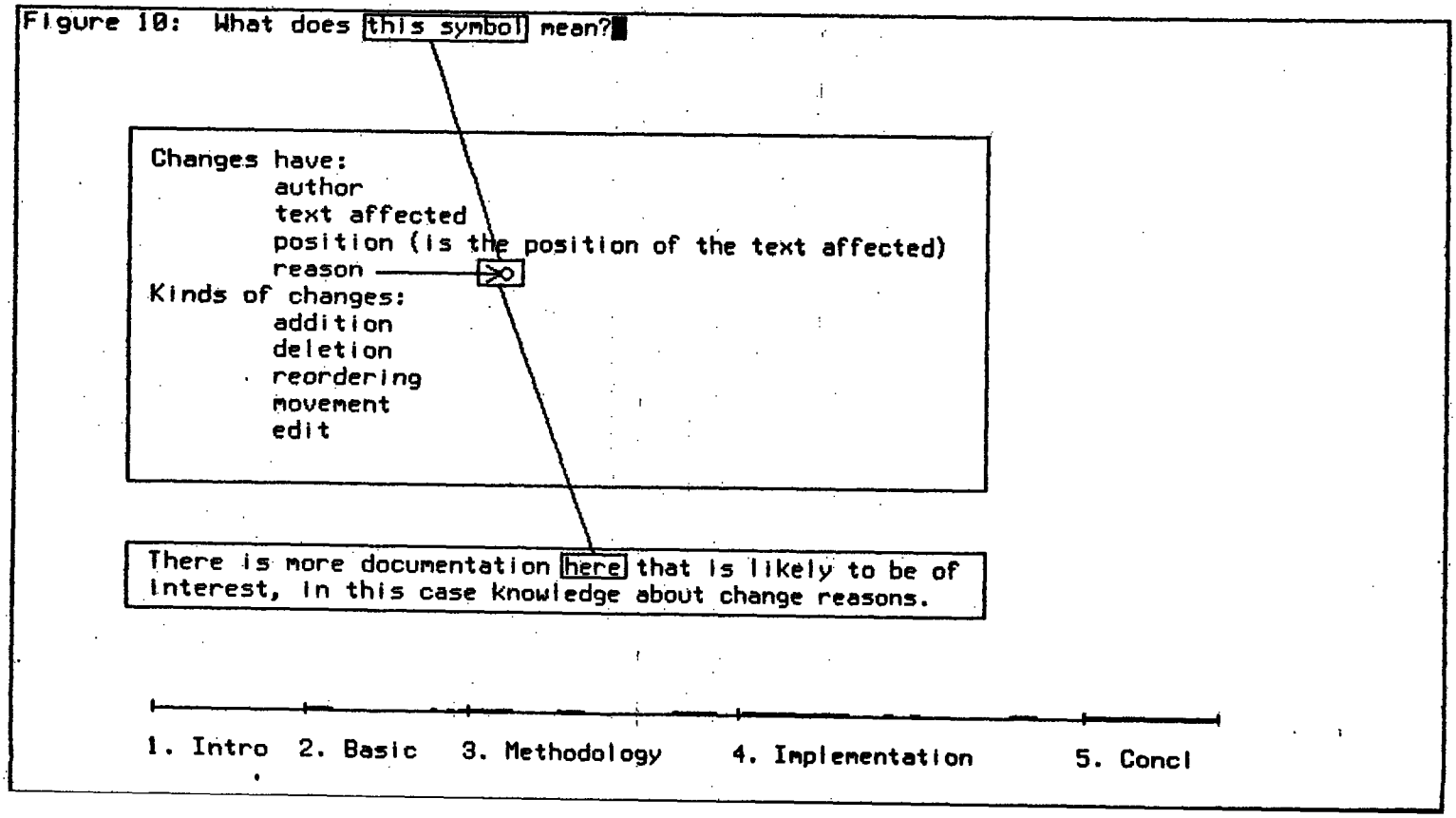
Figure 8: It is now a much later stage in the writing of the paper; the user has not been working on the paper lately and wants to review the changes that were last being made. The presenter chooses as presentation framework a graph showing the proportionate sizes of the major sections of the text, and adds as attachment presentations some change bars (heavy lines) presenting the sizes and relative positions of the changed areas.

Figure 9: Needing some help viewing the extensive changes, the user looks for ways to categorize them. For the presentation framework, the presenter chooses a tabular-text presentation of its knowledge about changes.

This is the first time in this scenario that a presentation framework is not presented in its entirety. The presentation in tabular form of the whole semantic network is not only too large for the screen -- most of it is irrelevant: the user wants to see only the knowledge about changes. Furthermore, some of semantic network will be "common knowledge", and even if it concerned changes, presenting it would be a distraction.

The presenter has two means of controlling the scope of this presentation. First, the domain builder provides *meta knowledge* to identify the common knowledge, and the presenter avoids showing it. Second, the presenter lets the user make some of the decisions about relevance, by showing a certain region of the semantic network, letting the user specify which of the closely related regions should be shown next: the user explores the network, and the presenter shows only the local context at any one time. The presenter suggests likely directions by attaching a special arrow-and-circle presentation.

Figure 10: The user asks for a presentation of the arrow-and-circle presentation style, and the presenter responds by explaining the style's general use and adding information about what this instance of the presentation means.



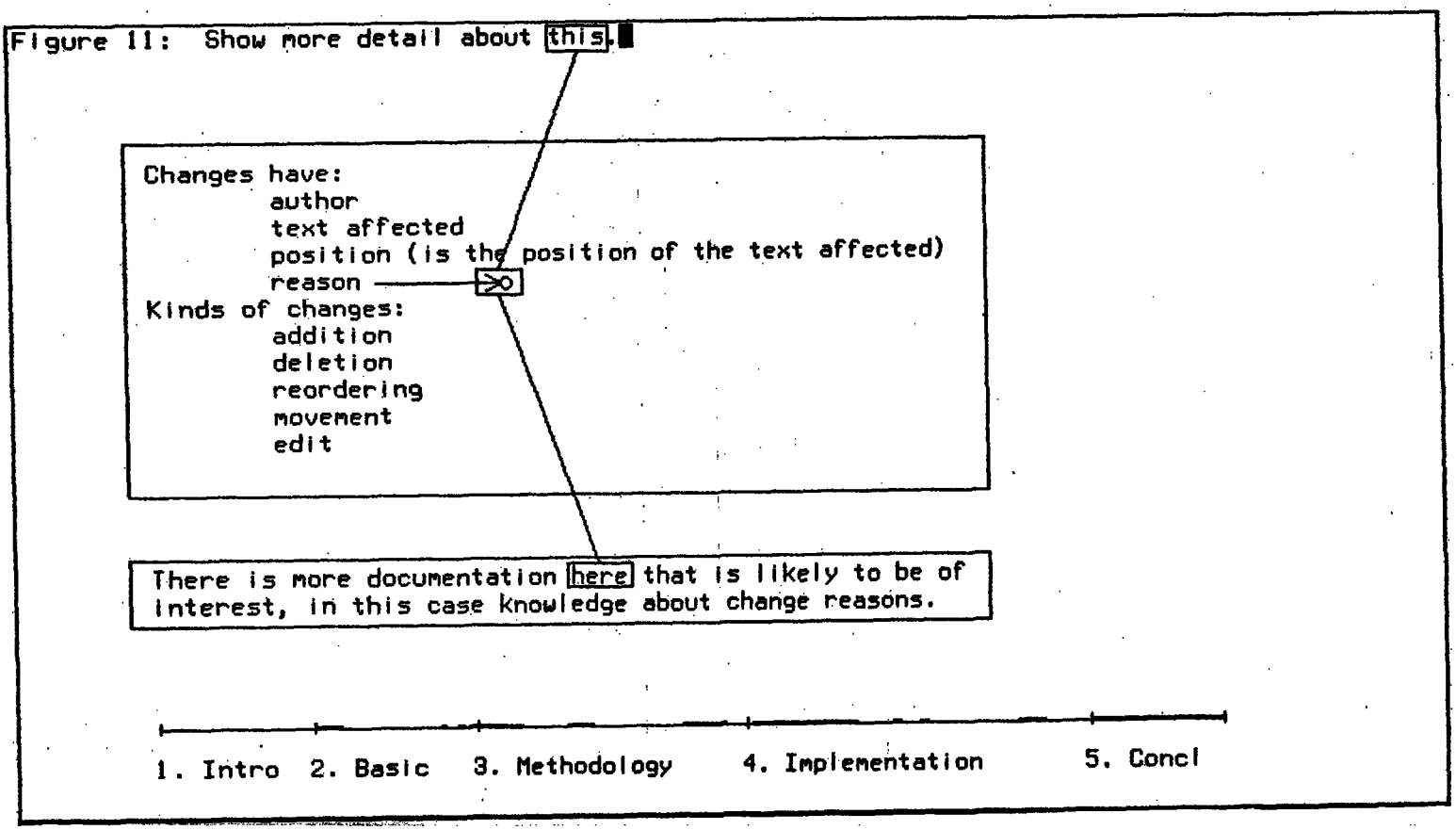


Figure 11: Show more detail about this.
Changes have:
author
text affected
position (is the position of the text affected)
reason
Kinds of changes:
addition
deletion
reordering
movement
edit
There is more documentation here that is likely to be of interest, in this case knowledge about change reasons.
1. Intro 2. Basic 3. Methodology 4. Implementation 5. Concl

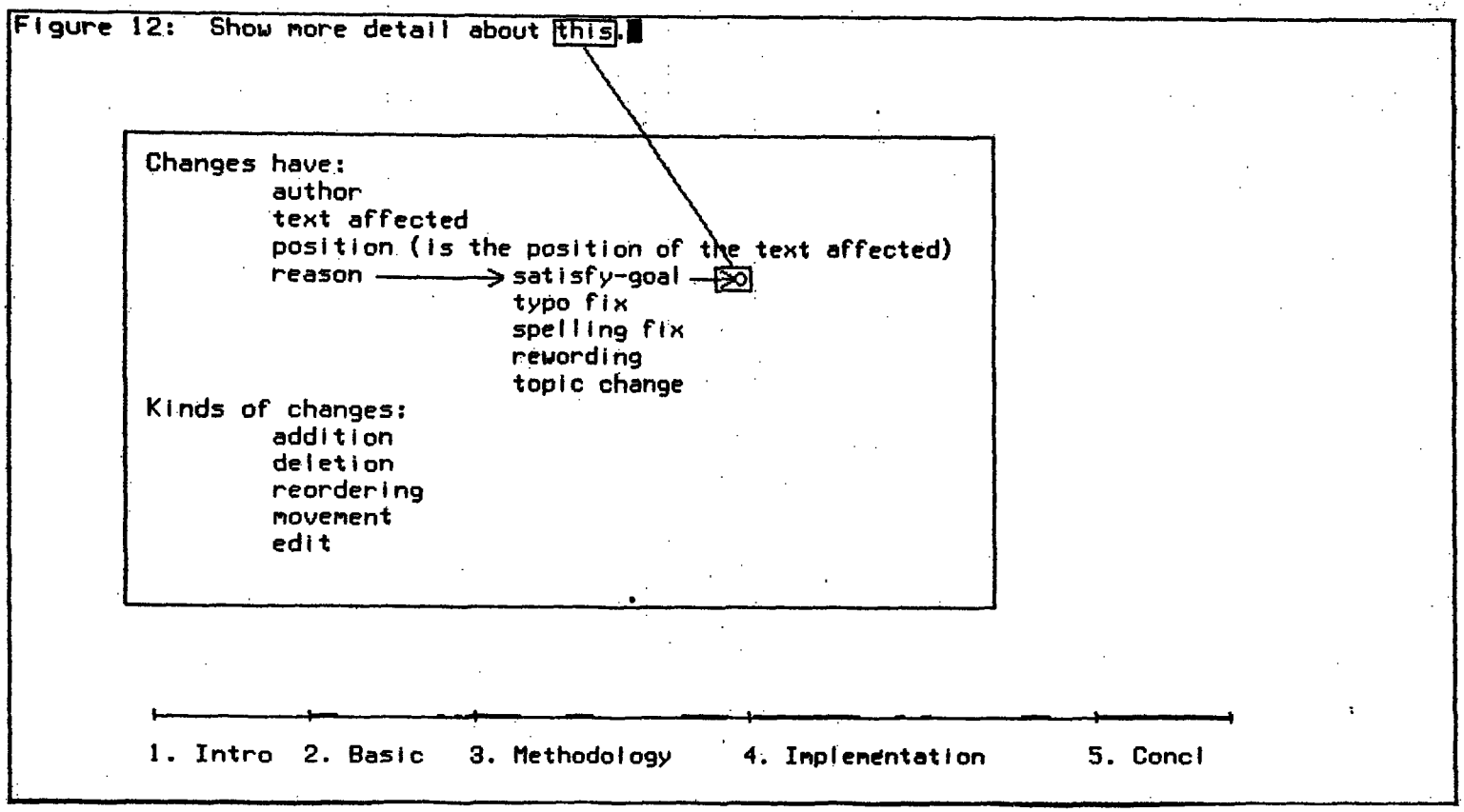


Figure 12: Show more detail about this.
Changes have:
author
text affected
position (is the position of the text affected)
reason
satisfy-goal
typo fix
spelling fix
rewording
topic change
Kinds of changes:
addition
deletion
reordering
movement
edit
1. Intro 2. Basic 3. Methodology 4. Implementation 5. Concl

Figure 11: The user takes the system's suggestion and asks to see documentation about change reasons. (I.e. the user wants to "walk in that direction".)

Figure 12: The presenter shows the kinds of reasons for changes. Since there has been a change made to satisfy some goal, the presenter suggests that the user find out what the satisfied goal is.

Figure 13: The semantic network has not structure for the satisfied goal other than a string supplied by the user; that string is presented in quotes to indicate that the system does not understand what it means.

Unfortunately, the user does not understand what it means either and cannot recall ever using the cryptic phrase "redder x". The user therefore checks whether someone else made that change.

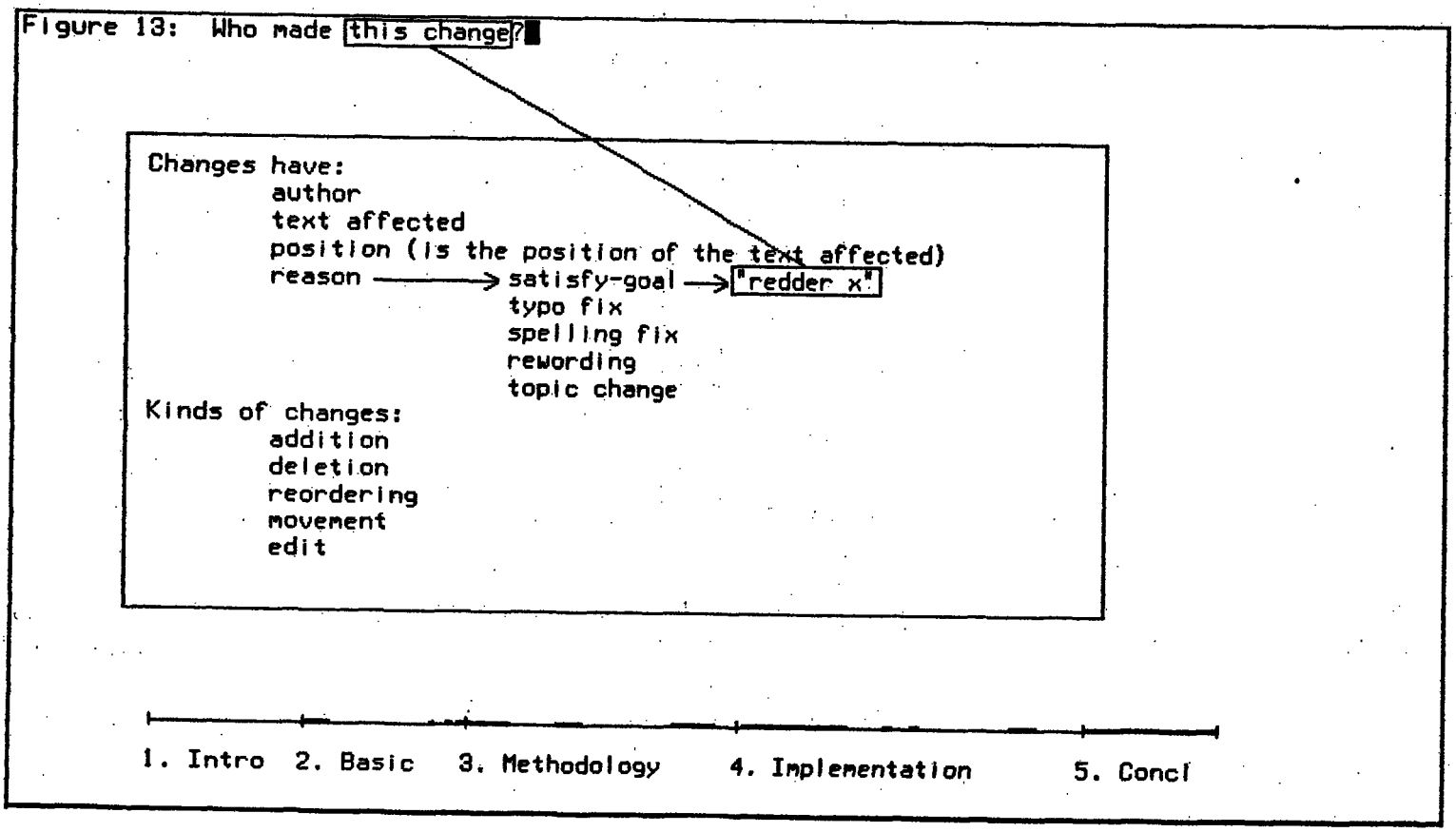


Figure 14:

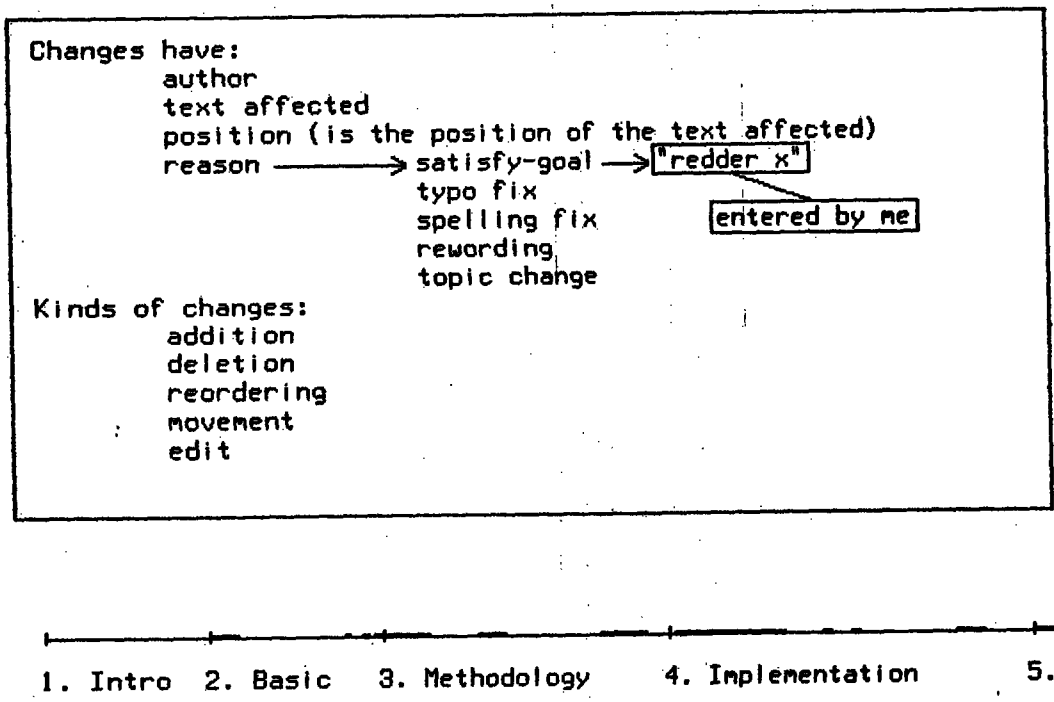


Figure 15: Show where these changes are.

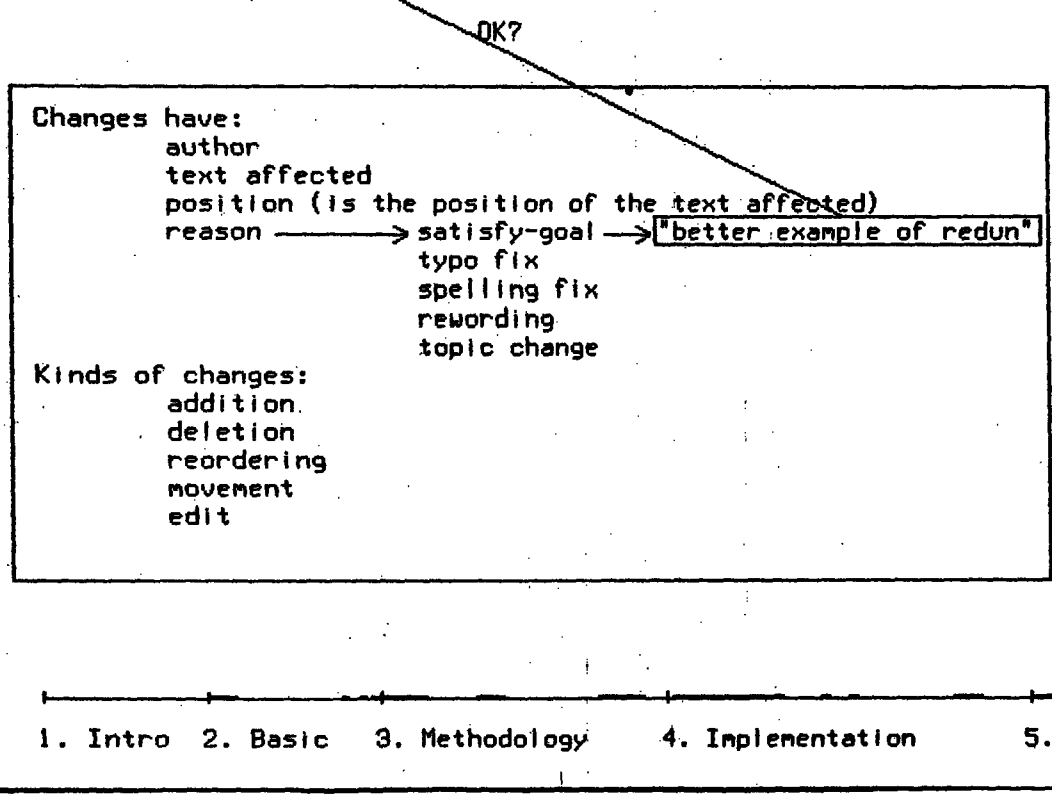


Figure 14: The presenter adds an attachment presentation to the tabular text framework of the semantic network: the general boxes-and-connecting-line style used here to present a related property, namely the user who added the "redder x" goal string.

Upon reflection, the user does recall this goal, which at the time seemed a clever abbreviation, but now just seems to be obscure. Therefore, the user decides to change the string to be less compact.

The user changes the goal string by editing its presentation. The user changes "redder x" to "better example of redun". The presenter meanwhile decides that since the user is changing the goal string, the presentation of who last entered it should be removed.

Figure 15: The user's command ("show where these changes are") needs disambiguation of "these changes", but the recognizer has defaulted it to the latest change of interest to the user. It thus presents the referent relation, but decides to get a confirmation before acting upon it; the fact that the recognizer wants a confirmation is presented by an "OK?" attached to the referent relation presentation. (The "OK?" is thus a very simple form of *peek presentation.*)

Figure 16: The graph's presenter is already presenting those changes' location; the system can take advantage of that by having the presenter show which of those changes are to satisfy the goal. The presenter uses the documentation as a "legend", and shows the correspondence between the "legend" and the graph by connecting lines.

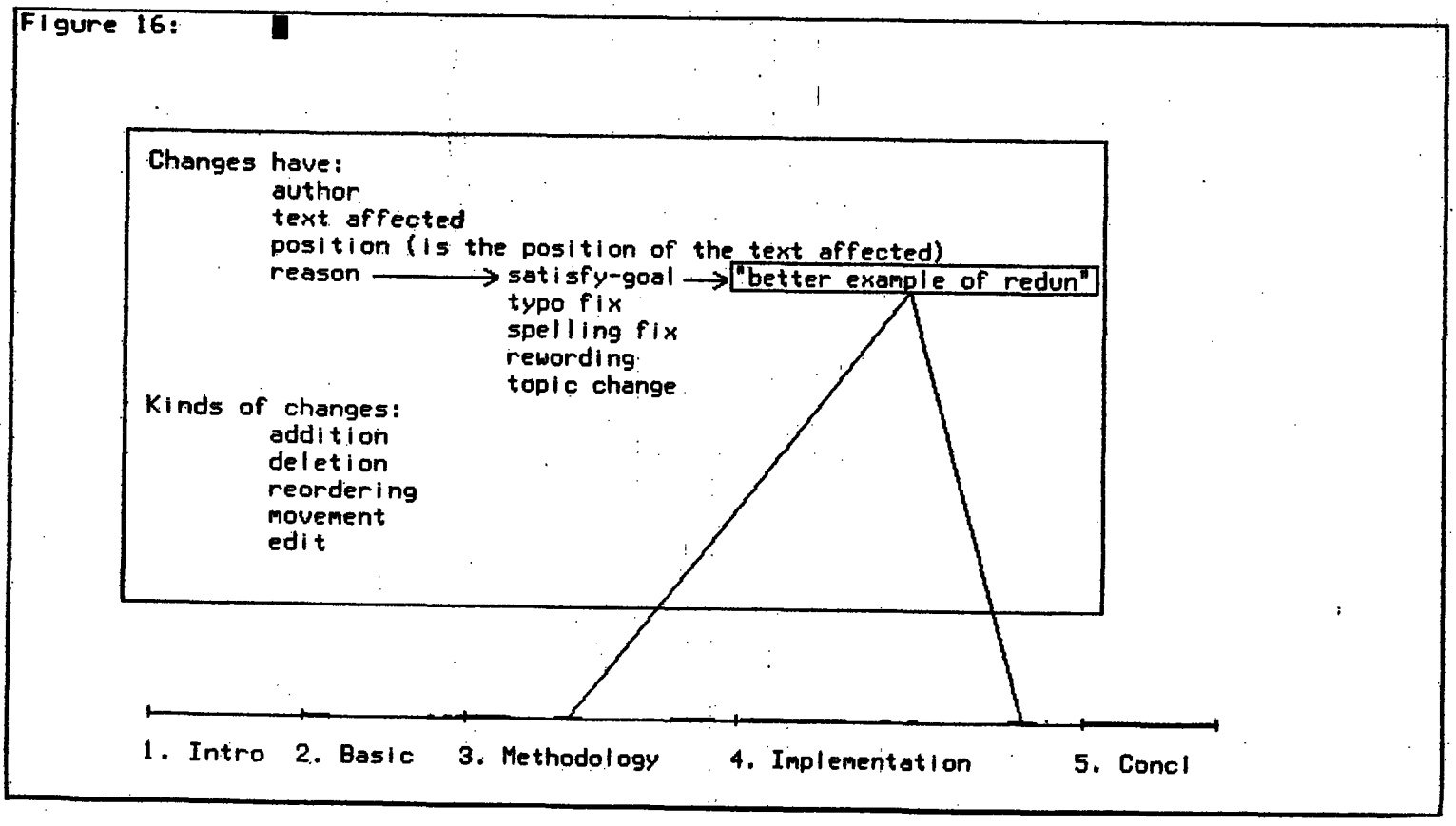


Figure 17: Don't show typo or spelling fixes.

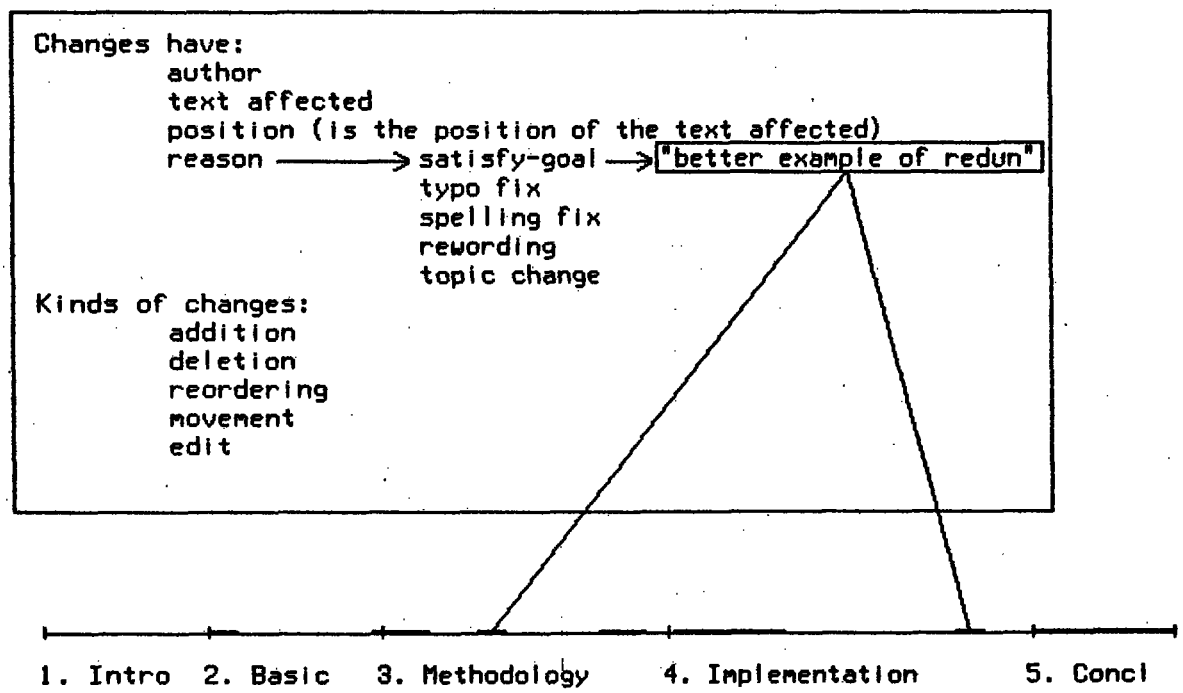


Figure 18: Show which are additions and edits.

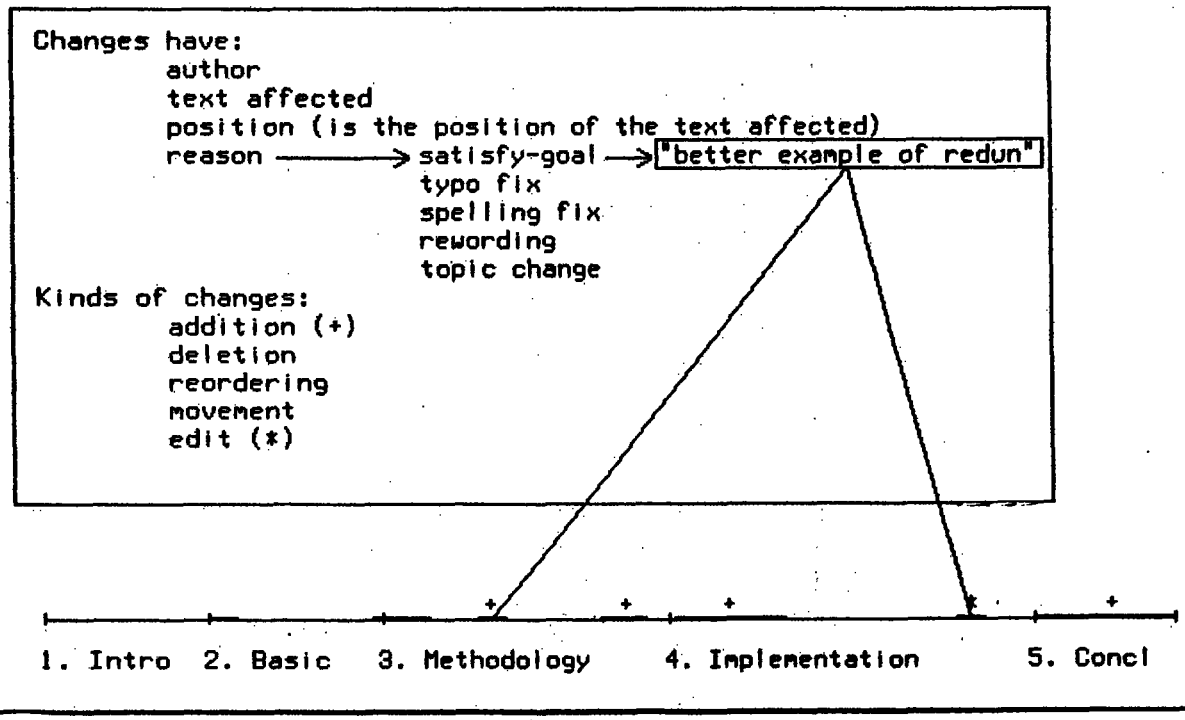


Figure 17: The documentation indicates that certain inconsequential changes can be recognized; the user takes advantage of this and tells the presenter to stop presenting those changes. The graph presenter responds by removing the change bars presenting typographical errors and spelling fixes.

Figure 18: Again the user issues a categorizing command, and again the presenter uses the documentation presentation as the "legend" framework. But now it does not use connecting lines as there would be too many on the screen, resulting in clutter. It shows the correspondence another way: by attaching labels.

The presenter finds no suggestions provided for labels for addition and edit, so it arbitrarily chooses two, "A" and "B", and displays them in the documentation and graph. The user has a better choice and edits the "A" and "B" in the documentation window to be "+" and "*". The recognizer changes the label presentation style which causes the labels on the graph to change accordingly; figure 18 shows the result. The user has commanded the presenter to change presentation style by *editing an instance of the use of that style.*

Editing a system-created presentation may thus not only be interpreted as a command affecting the presentation's domain object -- the command may affect the activity of the presenter that created the edited presentation. Another example would be deleting a presentation's display form; this can be interpreted as a command to stop presenting that object, rather than a command to delete that domain object. *This sort of interface ambiguity is inherent in any presentation program.* Some application programs may restrict the interface to eliminate this ambiguity; others may deal with the ambiguity in other ways, for instance by establishing safe, defaults that can be undone or asking for confirmation, thus giving the user more convenience and flexibility in how commands are issued. Recall that this same sort of ambiguity occurs in the EMACS DIRED and Edit Options subsystems discussed in section 2.2.

Figure 19: Show changes from old version viewpoint.

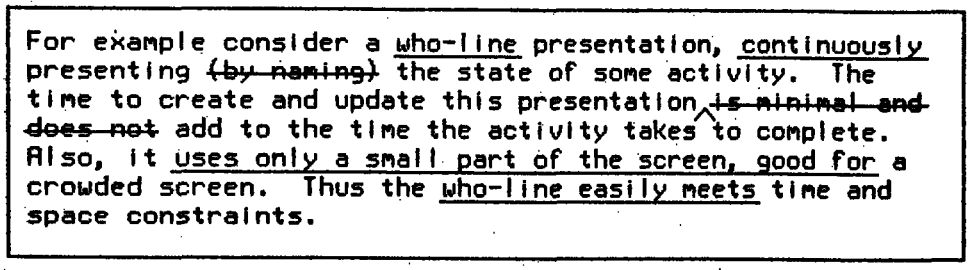


Figure 20: Show more detail about this.

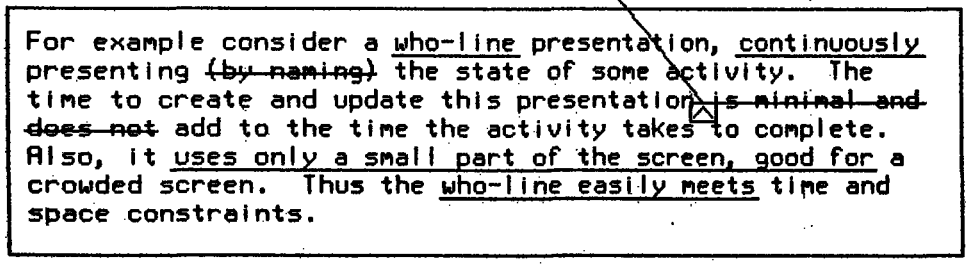


Figure 19: The user now decides to take a detailed look at one of the changes, identifying which by pointing to one of the change bars on the graph. The previously displayed presentations are erased (or at least not shown in this figure, for simplicity) and the new detailed change presentation takes their place.

The previous version of the text is the presentation framework. This is an example of a framework with "canned presentation algorithm": a standard text-filling algorithm that makes the text fit into the window. The attachment presentations are proofreader-style marks presenting the kind and position of the changes: Underlined text indicates edited text (e.g. one phrase replaced by another), crossed out text indicates deleted text, and carets indicate where new text has been added. But for the moment these marks do not present all of the detail: they do not show what edited text was changed to, or what new text was added.

Figure 20: The user asks for some of that missing detail.

Figure 21: And the presenter provides it, using the general boxes-and-connecting-line attachment presentation style. The content of the larger box, "(assume a one processor system)", is the text that was added at the point indicated by the caret.

This style of old-version framework plus proofreader attachment presentations, used in this scenario to *view changes*, could also be used to *make changes* to a paper: the user could delete text (or *mark* it for deletion) by crossing it out, using display form editor commands to draw lines through the text display forms; or make insertions by creating a caret, connecting line, and box with the text to be inserted.

Furthermore, the two could be combined, the system presenting changes already made, and the user making more -- or undoing some -- changes. This is another example of the idea of a presentation being a *shared communication medium* between the user and system.

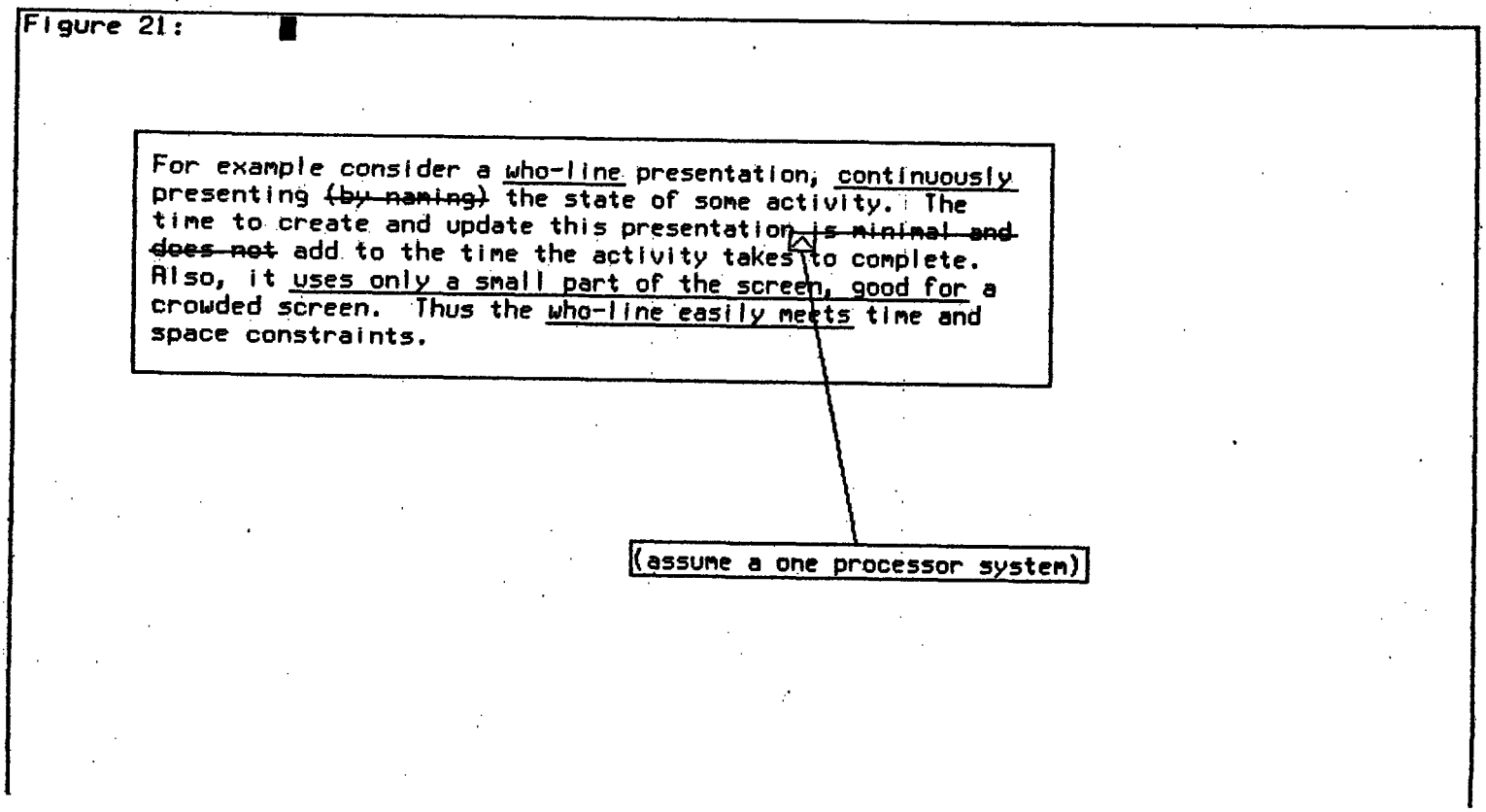


Figure 22: Show a comparison of the changes.

For example consider a who-line presentation, continuously presenting ~~(by naming)~~ the state of some activity. The time to create and update this presentation ~~is minimal and does not~~ add to the time the activity takes to complete. Also, it uses only a small part of the screen, good for a crowded screen. Thus the who-line easily meets time and space constraints.

For example, consider a peek presentation, continually presenting the state of some activity. The time to create and update this presentation (assume a one processor system) adds to the time the activity takes to complete. Also, it is often undesirable for the peek to use much of a crowded screen. Thus the peek is both time and space constrained; it should either be small and simple, or use as much of existing presentations as possible.

Figure 23: Show more detail about this

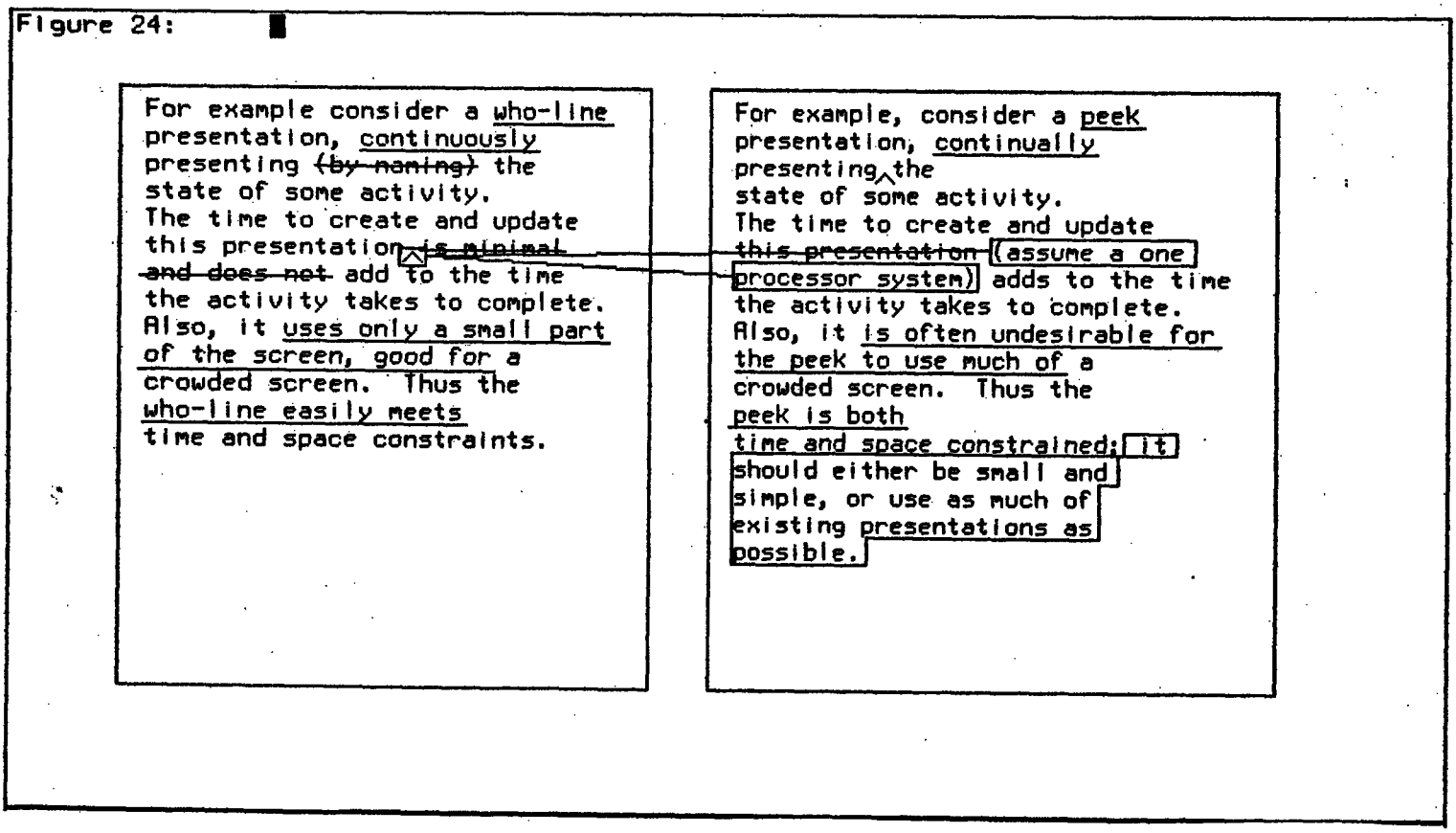
For example consider a who-line presentation, continuously presenting ~~(by naming)~~ the state of some activity. The time to create and update this presentation ~~is minimal and does not~~ add to the time the activity takes to complete. Also, it uses only a small part of the screen, good for a crowded screen. Thus the who-line easily meets time and space constraints.

For example, consider a peek presentation, continually presenting the state of some activity. The time to create and update this presentation (assume a one processor system) adds to the time the activity takes to complete. Also, it is often undesirable for the peek to use much of a crowded screen. Thus the peek is both time and space constrained; it should either be small and simple, or use as much of existing presentations as possible.

Figure 22: As the previous figure illustrated, one way to see the details of the changes is to show attached boxes with the additional text detail inside. This is especially good for quick, temporary presentations when the surrounding text and changes are already presented and will happen as a result of the presenter's tendency to augment existing presentation frameworks (thereby minimizing redundancy, decreasing the space used, and increasing the presenter's response). But the user can also ask for a general comparison of the changed area and see old and new versions side by side.

This figure illustrates some further processing by the presentation framework algorithm: paragraphs are not only filled to the window size, but synchronized between the two frameworks so that their lines start and break together whenever possible.

Figures 23 and 24: *The same command given by the user in figure 20 could be treated differently if both viewpoint frameworks were being used*: Instead of adding a box attachment presentation with the inserted text inside, which would be a *redundant presentation* (the inserted text is already being presented in the new-version framework), the presenter shows the correspondence between the old-version's caret and the new-version's inserted text.



Bibliography

[BOXER81]
Progress report on the BOXER system for non-expert users, part of the progress report for the Educational Computing Group, M.I.T., 1981. H. Abelson, group leader.

[Bolt79]
Bolt, Richard A.
"Spatial Data-Management". Architecture Machine Group, M.I.T., 1979.

[CCA79]
"Program Visualization: Concept Paper". Computer Corporation of America, October 1979.

[Moriconi80]
Moriconi, Mark.
"A Graphics-Oriented Environment for Understanding, Manipulating, and Debugging Systems". Research Proposal No. ECU-80-069R, SRI International, August 1980.

[Newman&Sproull73]
Newman, William N.; Robert F. Sproull.
Principles of Interactive Computer Graphics, McGraw-Hill, 1973.

[XeroxStar-Seybold81]
Seybold, Jonathan W.
"The Xerox Star". *The Seybold Report on Word Processing*, May 1981, Vol. 4, No. 5. Seybold Publications, Inc.

[Stallman81]
Stallman, Richard M.
EMACS Manual for ITS Users, Massachusetts Institute of Technology Artificial Intelligence Laboratory, AI Memo 554, 1981; *EMACS Manual for TWENEX Users*, AI Memo 555, 1981.

[Weinreb&Moon81]
LISP Machine Manual, third edition, Massachusetts Institute of Technology Artificial Intelligence Laboratory, 1981.

[Zdybel, Greenfeld, Yonke, Gibbons 81]
Zdybel, Frank; Norton R. Greenfeld; Martin D. Yonke; Jeff Gibbons.
"An Advanced Information Presentation System". To appear in *Proceedings of the 7th International Joint Conference on Artificial Intelligence*, 1981.