Capturing Graphic Design Knowledge from Interactive User Demonstrations

by Alan Greg Turransky
Bachelor of Fine Arts
University of Massachusetts at Amherst, 1990

Submitted to the Program in Media Arts and Sciences,
School of Architecture and Planning,
in Partial Fulfillment of the Requirements for the Degree of
Master of Science in Media Arts and Sciences
at the Massachusetts Institute of Technology, September 1993

© Massachusetts Institute of Technology 1993, All rights reserved.
Capturing Graphic Design Knowledge from Interactive User Demonstrations

by Alan Greg Turransky

Submitted to the Program in Media Arts and Sciences, School of Architecture and Planning on July 15, 1993 in partial fulfillment of the requirements for the degree of Master of Science in Media Arts and Sciences at the Massachusetts Institute of Technology.

Abstract

The abundance of layout problems commonly associated with the presentation of visual information on computer displays necessitates that computer systems be incorporated with graphic design knowledge to effectively and intuitively aid users in presenting, customizing, and organizing this form of data. Current methods of encoding such knowledge requires that human designers verbally translate their expertise into a set of programmable rules, frames, cases, or constraints. Computer systems which can be trained to learn the techniques designers use to effectively present visual information, by having a designer demonstrate their application on a working example may provide a more natural means of translating this type of knowledge from its original visual form into the electronic environment, without the necessity to first translate it into a textual representation.

This thesis describes a system which uses a machine learning technique called Programming by Demonstration to overcome this translation problem and enable the transformation of visual ideas into usable symbolic forms. It offers a working model, called the Abatan system, for capturing re-usable, graphic design knowledge from interactive user demonstrations.

Thesis Supervisor: Ronald MacNeil, M.F.A.
Title: Principal Research Associate

This work was sponsored in part by the Alenia Corporation, the Joint National Intelligence Development Staff, and the Kansa Corporation. The views and conclusions in this document are those of the author and should not be interpreted as necessarily representing the official policies, expressed or implied, of the Alenia Corporation or the Joint National Intelligence Development Staff.
Capturing Graphic Design Knowledge from Interactive User Demonstrations

by Alan Greg Turransky

The following people have served as readers for this thesis.

Supervisor
Ronald L. MacNeil, M.F.A.
Principle Research Associate
Visible Language Workshop, Massachusetts Institute of Technology

Reader
Russell A. Kirsch, S.M.
Director of Research
Sturvil Corporation

Reader
William L. Porter, Ph.D.
Muriel and Norman Leventhal Professor of Architecture and Planning
Department of Architecture, Massachusetts Institute of Technology
# Acknowledgments

Several people have contributed to this work in many ways; writing computer code, giving advice and suggestions, and providing general overall support. I wish to thank:

<table>
<thead>
<tr>
<th>Name</th>
<th>Acknowledgment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ron MacNeil</td>
<td>For being my supervisor and pushing me to describe the subject matter in this thesis using illustrations more than words.</td>
</tr>
<tr>
<td>Russell Kirsch</td>
<td>For providing challenging questions and detailed responses, not to mention being such a wonderful source of inspiration.</td>
</tr>
<tr>
<td>Bill Porter</td>
<td>For truly being interested in the design process and the nature of design.</td>
</tr>
<tr>
<td>Muriel Cooper</td>
<td>For hard-pressed questions and providing a stimulating environment with lots of fun toys to play with.</td>
</tr>
<tr>
<td>Louie Weitzman</td>
<td>For suggestions, advice, and always keeping in perspective what research should be about; not to mention helping with the Output Mechanism.</td>
</tr>
<tr>
<td>B.C. Krishna</td>
<td>For being honest, playing devil's advocate, and taking lots of breaks; not to mention writing the text library.</td>
</tr>
<tr>
<td>Maia Engeli</td>
<td>For always seeing and putting emphasis on the design rather than the computer, regular shipments, and help on the example scenario chapter.</td>
</tr>
<tr>
<td>John Grabowski</td>
<td>For providing me with and teaching me about all of the fun and wonderful UNIX toys.</td>
</tr>
<tr>
<td>Bob Sabiston</td>
<td>For answering all of my BadWindows questions, lots of free movie passes, having such a great music selection, and for being such a wonderful late night working partner.</td>
</tr>
</tbody>
</table>
Alice Lei
For being the other black sheep of the family with me.

Laura Scholl
For saying those three little words to me early on which truly said it all ... “It's not you.”

Amy Freeman
For a wonderful ear, a warm heart, and for putting up with everything (you deserve better).

Linda Peterson
For everything.

Dave Small
For being Dave Small.

Fellow Graduate Students
Suguru Ishizaki (for wonderful advice and suggestions on the Graphical History, object movement, and ideas in general), Steve “Stone” Librande (for explaining C in real world, easily understandable terms), Grace Colby (for emphasizing the design before the computer), Didier Bardon, Michelle Fineblum, Craigie, “K,” and Jeff.

Henry Lieberman
For his interest in Programming by Demonstration.

David Kurlander
For advice and suggestions on the Graphical History.

David Maulsby
For advice and suggestions in general.

Louis Silverstein
For advice on how we go about designing.

Beverly Woolf
For first suggesting that I come to the Media Lab.

Ray Lauzzana
For being my true advisor and friend over the years.

Leo Degen
For knowing all along what it was all truly about. Hou je Taai!

Deb and Sean
For eventually giving me a niece or nephew whom I can quickly hook up to a computer.

Mom and Dad
For never questioning and always supporting my educational decisions, and for never making me truly explain to them what it is I actually do.

Jay
For putting up with my long absence and for (believe it or not) being a wonderful source of inspiration.
# Table of Contents

## chapter 1. Introduction

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Motivation</td>
<td>11</td>
</tr>
<tr>
<td>Approach</td>
<td>15</td>
</tr>
<tr>
<td>The Title Example</td>
<td>17</td>
</tr>
<tr>
<td>Scenario</td>
<td>19</td>
</tr>
<tr>
<td>Document Structure</td>
<td>22</td>
</tr>
<tr>
<td>Definitions</td>
<td>22</td>
</tr>
</tbody>
</table>

## chapter 2. Related Research

<table>
<thead>
<tr>
<th>Name</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Designer: A Knowledge Based, Graphic Design Assistant</td>
<td>25</td>
</tr>
<tr>
<td>Liga: Intelligent Layout for Information Display</td>
<td>26</td>
</tr>
<tr>
<td>Tyro: The Case Based, Graphic Designer's Apprentice</td>
<td>27</td>
</tr>
<tr>
<td>Metamouse: Specifying Graphical Procedures by Example</td>
<td>28</td>
</tr>
<tr>
<td>Mondrian: A Teachable Graphical Editor</td>
<td>29</td>
</tr>
<tr>
<td>Chimera: Graphical Editing by Example</td>
<td>30</td>
</tr>
</tbody>
</table>

## chapter 3. Overview of the Project

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>The Name “Abatan”</td>
<td>31</td>
</tr>
<tr>
<td>System Capabilities and Components</td>
<td>32</td>
</tr>
<tr>
<td>Selecting an Example</td>
<td>33</td>
</tr>
<tr>
<td>Building and Describing the Example Layout</td>
<td>34</td>
</tr>
<tr>
<td>Encoding the Knowledge</td>
<td>37</td>
</tr>
<tr>
<td>Testing the Knowledge</td>
<td>39</td>
</tr>
<tr>
<td>The Table of Contents Example</td>
<td>40</td>
</tr>
<tr>
<td>Scenario</td>
<td>41</td>
</tr>
</tbody>
</table>
## List of Figures

### chapter 1. Introduction

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>The direct translation of visual rules into symbolic form.</td>
</tr>
<tr>
<td>1.2</td>
<td>Applied design principles.</td>
</tr>
<tr>
<td>1.3</td>
<td>Weitzman's Design Support system, the Logic of Layout.</td>
</tr>
<tr>
<td>1.4</td>
<td>Techniques used to emphasize a “title.”</td>
</tr>
<tr>
<td>1.5</td>
<td>A “title” object formatted with simple design rules.</td>
</tr>
<tr>
<td>1.6</td>
<td>Hierarchical structure of a layout.</td>
</tr>
<tr>
<td>1.7</td>
<td>Anchor objects of a layout.</td>
</tr>
<tr>
<td>1.8</td>
<td>Re-application of “Title” rules to new arguments.</td>
</tr>
</tbody>
</table>

“Title” example action sequences. 19 - 21

### chapter 2. Related Research

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Weitzman’s Designer.</td>
</tr>
<tr>
<td>2.2</td>
<td>Colby’s Liga.</td>
</tr>
<tr>
<td>2.3</td>
<td>MacNeil’s Tyro.</td>
</tr>
<tr>
<td>2.4</td>
<td>Maulsby’s Metamouse.</td>
</tr>
<tr>
<td>2.5</td>
<td>Lieberman’s Mondrian.</td>
</tr>
<tr>
<td>2.6</td>
<td>Kurlander’s Chimera.</td>
</tr>
</tbody>
</table>

### chapter 3. Overview of the Project

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>The Abatan Design Environment.</td>
</tr>
<tr>
<td>3.2</td>
<td>Schematic diagram of the Abatan system.</td>
</tr>
<tr>
<td>3.3</td>
<td>The color and font palettes.</td>
</tr>
<tr>
<td>3.4</td>
<td>The Lexicon palette.</td>
</tr>
<tr>
<td>3.5</td>
<td>The Exemplar Layout.</td>
</tr>
<tr>
<td>3.6</td>
<td>The Graphical History.</td>
</tr>
<tr>
<td>3.7</td>
<td>The Generalization Editor.</td>
</tr>
<tr>
<td>3.8</td>
<td>A table of contents formatted with seven design rules.</td>
</tr>
<tr>
<td>3.9</td>
<td>The “Bottom,” “Left,” and “Right” aligning actions.</td>
</tr>
<tr>
<td>3.10</td>
<td>The “Equal Color” and “Equal Font” actions.</td>
</tr>
<tr>
<td>3.11</td>
<td>Graphically altering elements to reflect system interpretation.</td>
</tr>
<tr>
<td>3.12</td>
<td>The Generalization Editor.</td>
</tr>
<tr>
<td>3.13</td>
<td>The “Table of Contents” domino icon.</td>
</tr>
<tr>
<td>3.14</td>
<td>“Table of Contents” layout before rules have been applied.</td>
</tr>
<tr>
<td>3.15</td>
<td>Re-applying the “Table of Contents” rules to new arguments.</td>
</tr>
</tbody>
</table>

Capturing Graphic Design Knowledge from Interactive User Demonstrations 8
chapter 4. How It Works

4.1 The “encoding” pipeline.  
4.2 Symbol class hierarchy.  
4.3 Symbols in Abatan are a three-part structure.  
4.4 The nine touch sensitive “hot spots” of a graphical object.  
4.5 Three different states of the Lexicon palette.  
4.6 The grammatical knowledge base default-settings look-up table.  
4.7 Three different states of the Exemplar Layout.  
4.8 The ACTION structure.  
4.9 Menu selection and direct manipulation editing of a panel.  
4.10 The Generalization Editor.  
4.11 Visual manipulation editing of a recorded action.  
4.12 “Sending” an action panel to the Editor.  
4.13 “Before” and “After” generalization states of an action.  
4.14 A domino icon.  
4.15 An Output Mechanism lexical action look up table.  
4.16 The LOL_action_name_lookup function.

chapter 5. Scenario

5.1 A current Scientific American table of contents.  
5.2 The Scientific American “Table of Contents” domino.  
5.3 The “before” state of a new Scientific American entry.  
5.4 The new entry after applying “Table of Contents” rules.  
5.5 Re-application of the rules after changes have been made.  
5.6 Further re-application of the rules to adapt to a new situation.  
5.7 Partial re-application “before” state.  
5.8 Partial re-application “after” state.  
5.9 A newly formatted Scientific American table of contents.  
5.10 Outputted rules from the Scientific American example.  
5.11 Automatic generation of a new table of contents.

Storyboard sequence.

Appendices

Layout 1, Version 1  
Layout 1, Version 2  
Layout 1, Version 3  
Layout 1, Version 4  
Layout 1, Version 5  
Layout 2, Version 1  
Layout 2, Version 2  
Layout 2, Version 3  
Layout 2, Version 4  
Layout 1, Rule and Constrain Function  
Layout 2, Rule and Constrain Function

Capturing Graphic Design Knowledge from Interactive User Demonstrations
chapter 1. Introduction

*Abatan* is an interactive design environment which allows designers to graphically create, manipulate, and describe page-layouts. During the design process, the system monitors how the designer assembles the layout. As the designer alters and conforms graphical elements in the layout, the system records the actions performed and directly translates these into generalized computational descriptions that can be re-used to format new elements in a similar style. The goal of the system is to directly translate visual knowledge into a symbolic form (computer code) so that it may be utilized by other existing design programs to aid users in creating new layouts.

This prototype system has been developed to provide visually oriented users with a tool to graphically encode their design knowledge into a symbolic representation, without the necessity to first translate it into a verbal or textual form. *Abatan* differs from existing *macro* systems and programs which allow users to graphically define *templates*, in that the knowledge stored is *generalized*, allowing for re-application in a manner which maintains the logical structure of the information, but is capable of adapting to new configurations. The system provides two major advantages over existing programs. First, the user is able to encode, edit, and re-use the knowledge they wish to convey, graphically, and second, the encoded knowledge can be output in a symbolic form which other existing design systems can utilize.

This thesis describes the design and implementation of the *Abatan* program and illustrates several examples of the system at work capturing graphic design knowledge from interactive user demonstrations.
In the past decade, technical advances have made the process of creating, manipulating, and transferring data extremely easy and quick. News information can be sent around the globe via satellite and televised onto a personal computer's display in a matter of seconds. Legal documents can be drafted, edited, and faxed in half the time it took only a few years ago, and multimedia presentations and hypertext documents can be customized to meet individual user needs and requirements. In order to enhance computer-human communications for a wide range of users, computer systems must be able to present information to the user in an easily understandable and comprehensible manner, which usually results in interacting with the data graphically.

Figure 1.1. The goal: The direct translation of visual rules into symbolic form.

In static media such as newspapers and magazines, it has traditionally been the job of a graphic designer to take the data and present it in a form which satisfies necessary communications goals, such as ensuring that the information maintains legibility, formatting certain aspects to emphasize importance, and so on (figure 1.2). Over the years the field of graphic design has developed a rich set of solutions for solving a wide range of visual communication problems, but unfortunately the amount of information used today is too overwhelming for a human designer to customize each and every document [Colby 92].
```c
#include <stdio.h>

#define PNMAX 10 /* max digits */

int digits; /* number of digits */
int pn[PNMAX]; /* phone number */
char *label_ptr[PNMAX]; /* current position */

main(argc, argv)
    int argc;
    char *argv[];
{
    register int i;
    bool foundvowel = FALSE;

    /* For each phone argument ... */
    while (*++argv != NULL)
        if (!getpn(*argv))
            fprintf(stderr, "PhoneName: %s isn't a phone number\n", *argv);

    Figure 1.2. Design principles can be applied to reinforce and emphasize portions of a layout. Even simple techniques such as changing typefaces, point sizes, and styles can make a difficult design easier to read (Reprinted from [Marcus 92]).
```
Currently, research in the fields of *Automatic Layout* and *Design Support* attempts to create computer programs, encoded with graphic design knowledge of how to format information, to aid novice users in easily creating high quality, professional looking visual layouts ([Colby 92] [Feiner 88] [MacKinlay 87] [MacNeil 90] [Marks 90] [Weitzman 88]). Weitzman's Design Support system, the *Logic of Layout* [Weitzman 92] contains a knowledge base which describes stylistic relationships among elements commonly found in the layout of *Scientific American* articles. Users of the system construct new *Scientific American* layouts by roughly sketching in objects, such as *headlines*, *images*, and *captions* without concern for fine detail. The system performs inferencing on the partially inputted data to determine relationships between the objects. After identifying such, it automatically applies constraints to format the design, changing typefaces, point sizes, and positioning to produce a layout which adheres to the magazine's uniform style.

![Figure 1.3](image)

*Observation of how a critical language relates to the design world.*

*Figure 1.3.* Weitzman's Design Support system, the *Logic of Layout*, uses inferencing to determine relationships between graphical objects, such as the image and a caption shown above, and automatically applies constraints to alter their attributes to conform to a specific style.

As can be seen in figure 1.3, when creating a layout which contains an image and a caption, the system automatically combines the two to create a composite object that functionally acts as one. In the process, the caption's width is constrained to equal that of the image's and its position altered to reflect that found in the magazine, not requiring that the user format these individual attributes.
While the importance and growing need for computer programs to be incorporated with graphic design knowledge (to aid users in effectively presenting visual information) is apparent, until recently, little emphasis and support have been placed upon the problem of actually encoding the knowledge into the computer. The primary method of encoding involves several non-trivial steps, most of which are non-intuitive to a visual designer. To incorporate the knowledge used in these systems a human designer must identify the relationships and problem solving techniques exhibited in existing layouts, verbally translate these into a textual description, and then hand this off to a programmer who encodes it into a symbolic form, such as a set of rules, cases, or constraints. Unfortunately these steps contain implicit variables which make the process extremely time consuming, error prone, and shallow in re-application scope, only enabling knowledge from context specific domains to be conveyed. For example, without further programming Weitzman's Logic of Layout is only able to support the creation of *Scientific American* articles, even though the individual rules which govern the layout, if generalized, could be used to format new layouts in different domains.

Computer systems that can be trained to learn design techniques through demonstration of their application on a working example may provide a more natural means of translating this knowledge from its original visual form into the electronic environment. This thesis describes an alternative model to the current method of verbally encoding design knowledge, that uses a machine learning technique called *Programming by Demonstration* [Cypher 93]. Programming by Demonstration allows users to program a system in the same manner a teacher might instruct a student; by presenting a problem task and illustrating the sequence of operations needed to solve the task on a working example. When Programming by Demonstration is used to teach graphic design, problem tasks and their solutions are communicated through the creation of example layouts. The actions used to format the layout and the results obtained are generalized to produce a working computer program that can be applied to solve similar layout problems. In the case of Weitzman's Logic of Layout, the rules used to constrain the image and caption in the previous figure could be illustrated simply by graphically manipulating the objects' attributes until they conformed into the final state which satisfied the rules.
Even as far back as the cave drawings at Lascaux, pictures have been one of the most effective, universal forms of communication [Jansen 77]. We use pictures to convey messages, present ideas, and visualize abstract data. When giving complex directions we typically draw a map rather than write out the steps, to provide a greater understanding. In short, a picture truly is worth a thousand words.

While the ability to interpret drawings is not difficult, the ability to effectively convey or teach this form of “knowledge” is not so easy. Artists and designers often find that it is extremely difficult to verbalize what they are conveying or teaching this form of “knowledge” is not so easy. Artists and designers often find that it is extremely difficult to verbalize what they are conveying or teaching. While the ability to interpret drawings is not difficult, the ability to effectively write out the steps, to provide a greater understanding. In short, a picture truly is worth a thousand words.

Below we see several examples of techniques used to emphasize a “title.” Note that no verbal or written explanation need accompany these layouts to describe them; the visual representation is fully capable of providing all of the necessary “visual rules” needed to express the idea [Librande 92].

Figure 1.4. Visual examples of techniques used to emphasize a “title.”

Capturing Graphic Design Knowledge from Interactive User Demonstrations
In order to test the “by example” idea, the Abatan system was built. The system uses the Programming by Demonstration technique to monitor the designer’s activities throughout the design process and automatically translates the graphical actions performed into a usable symbolic form. The interaction dialog consists of the designer demonstrating design rules by performing the sequence of steps needed to solve a specific design problem on a concrete example. The application focus of the system centers around designing page-layouts.

The type of design knowledge encoded deals with the lower-order rules which specify constraint relationships among graphical elements found in the layout. These relationships are the building blocks which are necessary in order for high-level concepts to operate. For example, if the idea is to have a “dominant structural order” in the layout, then the rules which constrain spatial and dimensional attributes of the graphical elements must be described before the concept can work. The Abatan system focuses on encoding such rules.

The goal of the system is three part. First, the designer should be able to encode knowledge into the system simply by creating a layout which exhibits the necessary rules. Second, the designer should be able to create and access the knowledge encoded to aid them in their work, while they are working and in a manner which minimally intrudes on the design process itself. Finally, the system should be able to output the knowledge in a symbolic form for re-use in other existing design programs.
Since this thesis focuses on learning by example, the best way to discuss the system is with a demonstration. The following section presents a short example scenario of capturing the simple design knowledge used to format the objects found in figure 1.5. In this example, the theme to be conveyed is that of emphasizing a piece of text by placing a rulebar of the same width, height, and color directly underneath it, drawing more attention to it than if it had been placed in the layout by itself. The rules which describe the various relationships between the individual elements in the layout are taught to the system by physically manipulating their attributes until they conform to a final state which satisfies the rules. For this example the targeted rules to be defined are as follows.

1. The rulebar's top edge should be aligned to the text's baseline.
2. The rulebar's left edge should be aligned to the text's left edge.
3. The rulebar's width should equal the text's bounding box.
4. The rulebar's height should equal the text's body-height.
5. The rulebar's color should be the same as the text's.

In addition to figure 1.5, the above rules (with the exception of number 4) describe the various constraints that were used to layout the chapter titles of this thesis, as well as the captions present in the appendices. When this document was created it was constructed in a program that allowed this author to graphically define these rules, but unfortunately did so in a "macro-like" manner. This required that two sets of rules be defined, one for the chapter titles and another for the appendices captions, even though if generalized, these rules could be used to format both. In Abatan, users can teach the system the basic, general idea defined by these rules using a single example and in return encode the necessary information to be able to layout both the titles and the captions in the same style.

Figure 1.5. A "title" object formatted with simple design rules.
Before illustrating the following scenario several details about the structure of the knowledge to be encoded must be explicitly stated. Traditionally, the individual visual elements which comprise a layout are often structured hierarchically. In the layout of the *New York Times* newspaper (figure 1.6) the top "node" of the hierarchy tree is the title "The New York Times," from which other "child nodes," such as headlines, base their attributes off of (figure 1.7). As will be seen later in this thesis the top node serves as an anchor point in which to base relationships. Appropriately enough, when conveying design knowledge in the Abatan system anchor objects are initially selected as the starting point. In the following example the text will serve as the anchor object, from which the rulebar's attributes will be dependent.

![Hierarchical structure of a layout.](image)

Figure 1.6. Hierarchical structure of a layout.

![Anchor objects of a layout.](image)

Figure 1.7. Anchor objects of a layout.

---

*Capturing Graphic Design Knowledge from Interactive User Demonstrations*
SCENARIO

The interaction sequence begins by creating the text and rulebar objects and roughly placing them into the layout. Since the text will serve as the anchor object, it is selected first. The first relationship which needs to be illustrated is that the rulebar's top edge should be aligned with the text's baseline, which in this case just happens to be its bottom edge. As will be described later in more detail (in chapter four), graphical objects within Abatan have nine touch sensitive hot spots; their four edges, four corners, and center. When an object's hot spot is clicked, it informs the system that there is something important about that point and it should look for a possible, future relationship involving it. For this relationship, the targeted hot spot is the text's bottom edge, since the rulebar's position will be dependent on it. After selecting the text's bottom edge the rulebar's top edge is clicked on (to point out its hot spot) and the object is dragged to a position directly below the text to illustrate the first of the four relationships stated above.

Next the designer must illustrate the relationship of left alignment between the objects. Again the text is selected first as the anchor and its left edge is touched as the hot spot. The designer then clicks on the rulebar's left edge and drags it over to be roughly aligned with the text's left edge. The designer need not precisely align the two objects, since Abatan incorporates the notion of semantic gravity points [Lieberman 93a] which allows for tolerance and provides a more fluid type of interaction.
Now that the objects are in their appropriate positions, the designer proceeds to illustrate the width relationship by clicking on the rulebar's lower right corner and resizing the object to the same width as the text's bounding box, during which time the text is still selected as the anchor object. Since we are describing relationships in a computational environment, certain attributes can be determined based on the information provided. For example, in traditional design the above relationship could never exist since the notion of a bounding box is local to computers. Instead this would be replaced by stating that the rulebar's width should equal the literal string's width or is based in a grid's column width. In Abatan, the designer is not restricted to using only those conventions found in traditional design. For the above relationship, any of these means of specification can be used.

The final dimensional relationship to be shown is that the rulebar's height should equal the text's body-height. This relationship is specified in almost the exact manner as the width relationship; by resizing the rulebar's height to equal that of the text's.
Finally, the two objects are colored the same color by making a new selection in the color palette to state the last relationship.

Technical Work

Rather than arbitrarily record each individual action performed, Abatan uses a grammar to determine whether an action is legitimate enough to record based on the rules which describe the distinct emergent properties associated with each object. After the demonstration is completed the system generalizes the information stored in these recordings and produces symbolic rules which both describe the relationships stated above and are capable of formatting new "titles" in a similar style. As can be seen in figure 1.8, when the rules are applied to new arguments, only those attributes which the relationships affect are altered, but in a manner which adapts to the new situation, the rulebar's position and dimensions are now relative to the new title's, as is its color, but the text's typeface and point size remain as is.

Figure 1.8. Re-application of "Title" rules to a new set of arguments.
This thesis is arranged into several different sections. Chapter two describes related research, highlighting work from the fields of Programming by Demonstration, Case Based Learning, Design Support, and Automatic Layout. Chapter three briefly gives an overview of this project and presents a second example scenario of capturing graphic design knowledge from a user presented example. Chapter four is the heart of this document. It describes, in detail, the different components of the Abatan system, their functionality, and purpose. Chapter five presents a final detailed scenario example and chapter six concludes this document with a discussion of future directions of capturing design knowledge and a brief summary of this thesis. Finally, appendices illustrating visual and symbolic examples of design knowledge are presented.

Since the subject matter contained within this thesis covers aspects from several different fields, the following section is presented to define and clarify research specific terms that will be used throughout this document. While all terms described below are formally defined within the body of this thesis, they are initially presented here for clarity and quick reference.

**Anchor:** Reference object or point in which another object's attributes are dependent on.

**Automatic Layout:** Area of graphic design and artificial intelligence research which focuses on the automatic generation of layouts containing text and image information for computer displays based on constraint rules which define specific styles of layouts.

**Bounding Box:** The smallest rectangular region which encompasses an object or group of objects in a layout.

**Case Base Learning:** An artificial intelligence problem solving methodology which makes decisions by adapting previously defined example problems, called cases, and applying these to solve new problems which have similar features.

**Class:** A group of similar objects that share common behaviors and general characteristics.

**Constraints:** A method of specifying values as relative relationships rather than as absolute numbers. These relationships are declared once and then maintained automatically. Constraints are often characterized by their ability to establish relationships (spatial, dimensional, typographic) among objects and their quality of suggesting relative connections between the objects.
Demonstrational Techniques: Techniques with which to construct abstract computer programs by performing actions on concrete example objects [Cypher 93].

Design Support: Area of design and artificial intelligence research which deals with supporting the user during the design process.

Example Based Programming: When computer programs are written through the use of user presented example data rather than conventional testing and debugging of textual programming languages [Ellman 89].

Felicity Conditions: Standards necessary for a human teacher to successfully instruct a student [Maulsby 92].

Generalization: The process of stripping away of example specific detail to produce, convert, or replace absolute items, such as numerical values, with abstract representations.

Graphical Histories: Connected, linear, sequential, thumbnail illustrations which represent a history of events, usually pertaining to user actions in a computer program.

Instructible Systems: Computer programs which generalize user presented information and automatically create symbolic routines, rules, or constraints.

Lower-Order Rules: Spatial, dimensional, and typographical rules which state specific constraint relationships among graphical objects found in a layout.

Macro: A small, user defined, program used to automate an exact series of actions or processes.

Message: A request sent to an object to change its state or return a value.

Method: A function which automatically implements a desired response when a message is sent to an object.

Page-Layout: The static presentation of textual and pictorial information to effectively communicate a message or idea. Good page layout is often characterized by its ability to link the visual structure of the information to reflect or enhance the content structure, in addition to making it perceptually legible and visually pleasing [Colby 92].

Programming by Demonstration: A subclass of Example Based Programming. A methodology for programming in which programs are written via the use of example data. Commonly, Programming by Demonstration systems learn how to perform a task by watching a user demonstrate the sequence of actions needed to execute the task on a working example. The system traces the user's input/output actions, infers information from these traces, and generalizes this into a high level function which can be used to solve new tasks [Cypher 93].
The problem of encoding and using various types of design knowledge to aid users in their work is one which has been investigated from many different angles. Most of this related research focuses on creating systems which use the knowledge to support the user while designing and not so much on the process of actually capturing it. While these systems all deal with “graphical” knowledge, their primary target domain focuses on general graphical editing tasks and the presentation of technical information such as charts and diagrams (as opposed to focusing on typographic and pictorial information found in page-layout). One of the reasons for this is that the conventions used in page-layout are not as well defined and structured as they are in other types of layouts, such as technical layouts. Technical layouts often follow a format which falls within a strict formal boundary, with distinct well defined rules making it an attractive problem set to work with. While within one style of page-layout there can and often does exist strict conventions which can be captured, the formal structure between page-layouts commonly differs from one design to the next.

There are three main areas of research that have close ties with the problem of encoding graphic design knowledge; these are: Design Support, Automatic Layout, and Programming by Demonstration. Research in Design Support and Automatic Layout does not focus so much on encoding, but rather on how the knowledge is re-used to aid in the design process, with specific emphasis on the quality and authenticity of the knowledge. Programming by Demonstration on the other hand deals more with the encoding aspect, but unfortunately has often fallen short on quality and detail, producing only general knowledge which is not satisfactory or robust enough to offer any serious support in designing page-layouts.
Weitzman's Designer [Weitzman 88] is one of the first Design Support systems to deal with problems commonly found in two dimensional layout design. The system critiques the design of graphical interfaces used as front-ends to instructional computer systems and offers designers of such interfaces alternative layout configurations to their designs (figure 2.1). The alternative suggestions are based on design contexts and styles defined in the system's knowledge base and attempt to improve the design by making it more visually effective and consistent with other related interfaces. The design knowledge used in Designer is based on visual communication principles [Cheatham 83] which link visual relationships to content relationships. For example, if two elements of the design are significantly different in function or meaning, Designer attempts to produce a visual representation which is also significantly different.

![Figure 2.1. Weitzman's Designer system critiques the layout of instructional interfaces and offers the user alternative solutions to their design.](image)

The system's design knowledge is stored using a frame based [Minsky 85] representation facility and is applied to and maintained in the design using constraints. While these constraints are used in the design of technically oriented layouts, their definition and representation are based on general design principles found in the visual arts [Wong 72], graphic design [Hurlburt 77] [Bertin 83], and architecture [Ching 79] [Sherwood 81].
Colby's research in Automatic Layout places heavy emphasis on the type of design knowledge needed to effectively present visual information on a computer display. Her system, *Liga* [Colby 92], automatically generates presentations of text and image information using a *case-library* [Riesbeck 89] to make design decisions based on the layout content structure of previously defined layouts. It dynamically adapts and re-configures a layout as environmental variables, such as the display's dimensions, change to ensure that the information maintains its legibility and logical semantic relationships (figure 2.2).

![Figure 2.2](image)

*Figure 2.2. Colby's *Liga* uses knowledge of the problem solving techniques exhibited in existing page-layouts to dynamically adapt a computer display's layout when environmental variables change, rather than simply reducing or enlarging the entire layout.*

The graphic design knowledge stored in Liga's case-library is defined by human analysis of the problems solving techniques used in existing page-layouts (i.e., the current method of encoding). The analyzer identifies these techniques and translates them into a set of programmable constraints. Sets of constraints which describe the stylistic relationships that govern the look of an individual layout are grouped together to form a case. Each case in the library is stored as a textual description of visual objects and their relationships to one another, and states how far these relationships can be bent before legibility of the information is lost.
Similar to Colby's Liga, MacNeil also represents and stores design knowledge as cases in his program Tyro [MacNeil 90]. MacNeil's philosophy for using cases is modeled after the fact that people solve a task by adapting an existing case (learned from previous experience) and applying it to the problem at hand. Tyro, a case based reasoning system, uses this methodology to aid a designer in the presentation and revision of technical designs. MacNeil defines a model of the design process and incorporates this into the system to control the design decisions made. The design process is classified as: 1. Making a decision; 2. Testing the decision by creating an example layout; 3. Critiquing and revising both the layout and the reasoning processes that went into creating the layout; and 4. Evaluating and fine tuning the generalization process devised for the given task. Each case in the system's library is composed of a sequence of steps illustrating how to resolve a single, specific type of layout problem (figure 2.3) and contains a condition-action rule which describes the circumstances surrounding the use of the case.

Figure 2.3. MacNeil's Tyro uses a case based library to aid in the design, presentation, and revision of technical layouts.

Capturing Graphic Design Knowledge from Interactive User Demonstrations
Maulsby's *Metamouse* [Maulsby 92], a Programming by Demonstration system, is one of the first successful attempts made at capturing graphical knowledge from a user presented example. With the system, users define and perform graphical editing tasks by instructing a turtle named *Basil*, “coaching” it through example executions of the task. The turtle observes the user's actions over time and performs localized analysis of changes in spatial relationships between objects in order to isolate constraints. A constraint, such as having one object's width always equal that of another, is encoded by performing multiple resizing operations to the second object each time the first object's dimensions are changed. Once such a constraint is found, the system attempts to predict and perform future actions (figure 2.4) by creating a customized, iterative procedure to ensure that the constraint is maintained.

The innovative importance of Maulsby's Metamouse was to allow users to define new procedures to aid them in performing tedious graphical editing tasks while they were actually designing and in a manner which minimally interfered with their interaction. As will be described in the next chapter, *Abatan* uses the same learning approach as Maulsby's Metamouse. This approach is based upon the notions of *felicity conditions* [van Lehn 83] which describe rules of interaction for human teachers to effectively instruct pupils. The four conditions are defined as: 1. Show all steps of the procedure; 2. Do all steps correctly; 3. Make all invisible objects and relationships visible; and 4. Introduce, at most, one new branch per lesson. The system incorporates an internal model of these conditions and uses it as a basis for learning.

*Figure 2.4. Maulsby's Metamouse performs localized analysis of changes in the spatial relationships between objects in order to determine constraints, and attempts to predict and perform future actions involving these constraints (Reprinted from [Cypher 93]).*
Lieberman's *Mondrian* [Lieberman 93a], like Maulsby's Metamouse, is an *instructible system* which learns graphical editing commands by watching the user illustrate sequences of actions on a working example. As the user performs a task, the system records the steps executed and creates a generalized, parameterized function which can be applied to solve future analogous problems. The system incorporates the new function into its tool palette, allowing users to re-access it as though it were a pre-defined command, and provides the user with a static visual representation of the steps in the procedure it has generalized (figure 2.5). The system requires that the user explicitly inform it when a demonstration begins and ends, and parameterizes the demonstration based on arguments supplied as variables at the beginning of the example. Mondrian uses domain knowledge of significant graphical relationships to determine constraints among objects in a design. Spatial relationships such as “left-of,” “half-of,” and “center” are defined internally and used as a basis for interpreting user actions. The choice of defining these relationships is based on the application scope of the system.

*Figure 2.5. Lieberman’s Mondrian allows users to extend its interface by demonstrationally teaching it new commands.*
Of the related research highlighted in this chapter, the most extensive and influential to this thesis is Kurlander's Chimera [Kurlander 93] editing system. Originally created for automating repetitive tasks commonly dealt with when creating user interfaces, the system has been expanded to deal with general graphical editing problems. It contains functionality for performing several demonstrational techniques such as Graphical Search and Replace [Kurlander 88a], Editable Graphical Histories [Kurlander 88b], Constraints from Multiple Snapshots [Kurlander 91], Constraint Based Search and Replace [Kurlander 92], and Macros by Example [Kurlander 93]. The system uses these techniques to create high level semantic operations which can be transformed into user-customizable, graphical editing functions.

As with the Abatan system, Chimera transparently monitors and records the user’s actions during the design process and generalizes them to define new formatting operations. Users are able to select, group, and edit individual panels of the system’s graphical history (figure 2.6) at any time to create specialized formatting commands which may or may not have been explicitly demonstrated in an example, but whose individual parts are present. Chimera differs from most Programming by Demonstration systems in that its recording device is always on. Only when the user is ready to or wants to define a new function does the system’s presence become truly apparent.

Figure 2.6. Kurlander’s Chimera records actions over time and allows users to access this information in an editable graphical history (Reprinted from [Cypher 93]).
chapter 3. **Overview of the Project**

Figure 3.1. The Abatan Design Environment.

**THE NAME “ABATAN”**

Appropriately enough, the name “Abatan” comes from a young Yuroba potter who learned her craft not by formal teaching and explicit training, but by observing, over time, how her mother and maternal grandmother applied their skills to perform their craft [Anderson 79]. *Abatan’s* method of learning, which has grown into and is now notably referred to as the *apprenticeship* system, is the most common method of teaching in the fine arts and design fields today. In the spirit of Abatan and the apprenticeship system, the prototype program bearing the same name as the Yuroba potter was developed by this author to demonstrate the utilitarian functionality and advantages of using this model of learning to encode design knowledge.

Capturing Graphic Design Knowledge from Interactive User Demonstrations 31
The Abatan system is composed of several different components, most of which are interconnected and directly accessible to the designer (figure 3.1, 3.2). These components and their sub-components include:

- **A Design Environment**: in which to construct the example layouts. The environment contains a variety of tools such as a color and font palette.

- **A Knowledge Base**: containing pre-defined grammars which describe rules used to format existing categories of page-layouts, a lexicon describing the vocabulary of graphical objects available to the designer (and which the grammar affects) to construct the example layouts with, and an internal listing which describes the hierarchical structure of the graphical objects.

- **A Recording Mechanism**: which translates graphical actions, verbatim, into computer code. The Recorder contains a Generator Test module that tests which actions are legitimate enough to record.

- **A Graphical History**: which visually represents user actions that the system has recorded.

- **An Editor**: to edit the data contained in the recordings and the history.

- **A Learning Module**: which generalizes the recorded actions.

- **An Output Mechanism**: which both incorporates the generalized knowledge back into the system and also outputs symbolic computer code.

*Figure 3.2. Schematic diagram of the Abatan system. Dark gray arrows suggest future extensions.*
In order for design knowledge to be encoded into the system there are essentially four steps the designer must go through. First, a subject matter to be demonstrated must be decided upon. Second, an example layout which embodies the lower order rules that describe the idea must be built. While creating the example layout, all objects which serve as pre-conditions for the rules to operate on must be specified and all post-condition relationships necessary for the rules to hold must be made visible. Any ambiguities which arise as a result of under specification of the rules and any aspects which are not explicit must also be clarified. In the third step, called the “learning phase,” the system takes over. At this point the actions performed and recorded during the design process are analyzed and then generalized to produce a high-level, adaptable formatting procedure which describes the rules. The final stage involves having the designer test the validity of the knowledge by reapplying it to new data.

The above steps are defined and loosely based on the notion of felicity conditions which state the interaction rules a human teacher must use to effectively illustrate an idea to a student. These rules are: 1. Show all steps of the procedure; 2. Do all steps correctly; 3. Make all invisible objects and relationships visible; and 4. Introduce, at most, one new branch per lesson. In the Abatan system, when the designer (who play the role of the teacher) is instructing the computer (the student), it is not necessary to follow these steps exactly.

SELECTING AN EXAMPLE

Initially, the designer must decide upon a subject matter to be encoded. The content of the design can range from newspaper and article layouts to technical document design. Whatever type of layout it is, the example which is to be used must fully exemplify the rules which describe the knowledge to be encoded. In the “Title” example in chapter one, the idea was to emphasize a text string by placing a rulebar of the same color and width underneath it, drawing more attention to the text than if it had been placed on the page by itself. If the designer wishes to encode information about how to emphasize a piece of text, by altering its typeface, point size, and color, creating a layout which only contains images and other graphic elements such as rulebars would prove useless.
In this second phase, the designer must build and describe a layout which exemplifies the idea and the rules. In order to do so, the vocabulary of graphical objects which the rules operate on (the lexicon) and the various relationships among them (the grammar), must be made visually apparent, the logical structure of the information firmly stated, and all ambiguous relationships clarified.

Figure 3.3. The color and font palettes.

The Design Environment (i.e., the graphical user interface) is where the designer creates the layout. Tools which accompany the environment include a color and font palette (figure 3.3). Color and transparency, and typeface and point size attributes of an object are altered using the color and font palettes respectively. As was seen in the “Title” example, spatial and dimensional attributes are manipulated directly by moving and resizing an object.

Figure 3.4. The Lexicon palette.
Objects are placed into the layout area, called the *Workspace*, with the help of the *Lexicon* palette and the *Exemplar Layout*. The Lexicon palette, shown in figure 3.4, lists the vocabulary of lexical objects available to the designer to build the layout with. Rather than supplying generic “text” and “graphical” objects, such as those found in traditional paint programs like MacDraw™ and Canvas™, vocabularies of objects with distinct functional purposes and attributes are defined. These include various types of headlines, titles, captions, etc. Each object’s function and attributes are dependent on and inherit from one of several pre-defined grammars which describe different types of page-layouts.

The Exemplar Layout (figure 3.5) illustrates examples of such grammars for laying out a page of a technical document, a *Scientific American* article, and a newspaper respectively. Its primary use is to visually display the affects of applying the grammar to a layout, and to show the designer what the different attributes associated with each type of object are. The Exemplar Layout is composed of objects found in the Lexicon palette (i.e., headlines, titles, rulebars), but each individual piece has been formatted with specific rules and has distinct properties and behaviors associated with it. When the designer clicks on the “Headline” button in the Lexicon palette, three corresponding headline objects are highlighted in the Exemplar Layout. One headline has the properties that its typeface is Helvetica and its point...
size is 24; the other two have the typeface Times-Roman and the point size 18. The designer uses the Lexicon palette to select a general category of objects, from which the system highlights all of the available, specific instances in the Exemplar Layout. When the designer selects one of the specific headlines, its attributes are sent back into the selected Lexicon palette object to show the designer a visual example of how such an object will look when placed into the layout. To add an object to the current layout, the designer clicks on an object in the Exemplar Layout and drags it over into the Workspace. The new object's appearance and functionality reflects that found in the Layout. For example, when the text object used in the “Title” example was created, it was done so by clicking on the “Title” button in the Lexicon palette. The system then highlighted all available title objects in the Exemplar Layout offering the designer several different choices, of which one was chosen. When the title object was placed into the design, its color, typeface, point size, and dimensions reflected that found in the grammar object from which it was spawned.

The initial purpose for the Exemplar Layout and Lexicon palette is to provide the designer with an existing set of tools to create the layout with. As will be seen later, the Exemplar Layout's grammar not only sets an object's default values when it is placed into the Workspace, but uses the properties associated with the object to distinguish it from others. In addition, the system uses the grammar to perform special tests to see if an action performed on an object is legitimate given the rules which describe it.

As was also seen in the “Title” example, relationships between objects in a layout are described by physically manipulating their attributes until they conform into a final state which satisfies the post-condition rules for the design knowledge to hold. In order to fully describe a layout, all superior and subordinate relationships must be specified. If a caption's position is dependent on an image's which is dependent on a headline's, the designer must show the system each of these individual relationships before a full description of the layout can be obtained.
The actual process of encoding the design knowledge is not a one step activity, but rather a three tier development. First, the actions performed while building the layout must be recorded. Second, the designer must be able to view and edit the recordings. Finally, the system must generalize the information. It is the job of the Recording Mechanism to capture, verbatim, the actions performed. The Graphical History visually presents the recordings to provide feedback to the designer, allowing them to browse over what the system has recorded. The Generalization Editor enables the designer to access and edit the recordings, and the Learning Module generalizes the finalized data to produce a high level procedural description of the entire design process.

The Recording Mechanism works in conjunction with the grammar. Each time an action is performed on an object, the grammar rules associated with that object tell the Recording Mechanism whether or not it was legitimate enough to record. For example, a grammar rule describing a headline object may state that the headline's point size can not be changed to a value smaller than a subheadline's. If the designer tries to do so, the system would deduce that the action was invalid and therefore would not record it. Given the rules which describe the text and rulebar objects from the previous example, only four actions were recorded, all of which were valid. The purpose for these tests, called Generator Tests, is two part. First, to filter which actions are recorded by the system, and second, to limit initially the type of knowledge encoded. It was the choice of this author to restrict early on what could be taught to the system in hope that it would learn within a small scope. Later on as more examples were presented, the range would broaden building off of what had already been learned. The idea behind these tests came from [Kirsch 72].

Figure 3.6. The Graphical History.
As each valid action is recorded a miniature visual panel is created in the system's Graphical History window (figure 3.6) reflecting the state of the design at that point. As mentioned a moment ago the Graphical History allows the designer to preview what the system has already recorded. In addition, the designer can access the actual operation that was performed to create the panel in the first place, translate the current state of the layout back to that point in time, or edit the panel by directly manipulating the objects contained in it. While it is visually noticeable each time the system records an action (or does not in which case a new panel is not created), the Graphical History is used to provide feedback that minimally intrudes on the designer's activities, but offers a solid sense of confirmation.

Figure 3.7. The Generalization Editor.

Once an action or series of actions has been performed, the designer can edit the recorded data describing it by “sending” the panel of the Graphical History which represents the action to the Generalization Editor (figure 3.7). The Generalization Editor, an interactive dialog box, provides both visual and textual access to the underlying, internal representation. It lists the type of action (i.e., alignment, color change, resize), the objects that are directly affected, and the resulting values. If we take a look at the action which resized the rulebar's width to equal the text's in the “Title” example, the action type is “Equal Width,” the arguments are a piece of text (as the anchor object) and a rulebar, and the value is 200 (the text's width in pixels). Similar to Pavlidis and Van Wyk's Automatic Beautifier [Pavlidis 85], during the design phase the designer need not align or size the objects precisely. Unlike other Programming by Demonstration systems, such as Peridot [Myers 86] which interrupts the design process by asking the user to confirm each action, Abatan allows the designer to quickly and freely
create the layout. The system automatically takes into account the
imprecision during generalization and adjusts it so that the objects are
aligned perfectly. If the designer wishes to break this precision (i.e., adding
an offset) or any other information about the action, it can be done,
through menu selections and register input, using the Editor. While a
textual transcription of the action is presented in the Editor, visual re-
enforcement of it can be view in the design itself. Techniques such as
overlaying guide lines and placing reference tags near objects are used to
emphasize relevant information.

Once the designer has finished creating and clarifying the layout, the
system takes over and generalizes the data. Unless specified otherwise,
absolute values are transformed into high-level relational descriptions which
state a desired goal and provide a method for obtaining it. For the resize
operation just mentioned, the action would be generalized into a form which
stated that the goal was "equal width of a rulebar to a piece of text" and the
method for achieving it would be a "width resize" operation, where the
relative value of "a text object's width" would be supplied as an argument
to the operation. When the knowledge is finally encoded, the system
creates a new icon in its interface, providing the designer access to this
chunk of encoded design knowledge.

TESTING THE
KNOWLEDGE
In order to test the validity of the knowledge encoded, the designer must
re-apply it to new data. This testing can be achieved in one of two
manners; either within Abatan or using a Design Support system. To test
the knowledge within Abatan, the designer creates new objects which
correspond to those used as pre-conditions to the demonstration, selects
them, and clicks on the new icon. The results of which are shown in the
Workspace by applying the generalized information stored in the icon to the
newly selected objects. The second means of testing is to output the
knowledge, in symbolic form, and plug it into a Design Support system for
use as a grammar. As will be seen later, both methods produce identical
results.
When the layout of this document was created, there were several repetitive tasks which needed to be performed. As was seen in the “Title” example, several of the rules involved in the design process overlapped (i.e., the chapter titles and the appendices captions). Looking at the table of contents in this document alone (figure 3.8), repetition is inherent throughout the entire layout. If we were to identify all the individual rules and define these using a macro facility such as Microsoft Word's Styles [Young 89], we would have to create individual rules for the chapter titles, their page numbers, and the section headlines and their numbers, even though there are only seven different rules which describe the entire layout (excluding the actual title “Table of Contents”).

<table>
<thead>
<tr>
<th>Introduction</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>Motivation</td>
<td>11</td>
</tr>
<tr>
<td>Approach</td>
<td>15</td>
</tr>
<tr>
<td>The Title Example</td>
<td>17</td>
</tr>
<tr>
<td>Scenario</td>
<td>19</td>
</tr>
<tr>
<td>Document Structure</td>
<td>22</td>
</tr>
<tr>
<td>Definitions</td>
<td>22</td>
</tr>
</tbody>
</table>

Figure 3.8. A table of contents portion formatted with seven design rules.

The general rules which describe the above layout are defined as:

1. A piece of text and a page number are bottom aligned.
2. Text is left aligned to the left margin of the page at a fixed offset.
3. Page numbers are aligned flush right with the page's right margin.
4. Page numbers have the same color, typeface, and point size as the text they are bottom aligned with.
5. Only one piece of text and page number can appear on a line.

The more specific rules are:

1. Section headlines have a smaller point size than the chapter titles.
2. Section headlines are left indented by a fixed amount, but chapter titles are flush left to the margin.
It took nearly a half hour and several attempts (before they were correct) to visually identify and manually enter these rules using the conventional method of encoding. The finalized version of these rules, which is described above, was determined and defined by Abatan in only five minutes.

The following example illustrates how these rules were encoded, generalized, and used by the system to format the entire table of contents layout. Since the rules described in the above “general” list work on a piece of text and a page number, to illustrate the entire list only one example which contains these two items need be demonstrated.

**SCENARIO**

The designer begins by creating a chapter title object and a page number object. In this example the chapter title serves as the anchor object, so the designer selects it first by clicking on its bottom edge to begin illustrating the first rule of bottom alignment. Next the page number’s bottom edge is clicked on and the object is dragged downwards so that it is roughly bottom aligned with the title. After performing this action, Abatan produces the leftmost panel in figure 3.9, which states the type of operation that was just performed (i.e., Align Bottom) and reflects the state of the design at that point. Abatan performs localized analysis [Cypher 93] between selected objects to determine if there are any relationships. Since the action dealt with spatial properties, Abatan tried to infer a positional relationship. Because both the title and page number’s bottom edges were clicked on initially and their finalized positions were almost equal, it was determined that the two were bottom aligned and the action was recorded.

![Graphical History](image)

*Figure 3.9. The “Bottom Aligning,” “Left Aligning,” and “Right Aligning” actions.*
Next the designer illustrates the left alignment rule. Since the layout area (i.e., the background) serves as the anchor object in this case, it is selected first. In Abatan, the layout area is also a design element used to build a layout and has the same functionality as other graphical objects; it may be clicked on, resized, has hot spots, etc. After selecting the layout area's left margin, the title's left edge is clicked on and the object is dragged to a position on top of the corresponding edge (figure 3.9 center). In a similar fashion, the same respective operation is performed with the page number and the right margin of the layout area (figure 3.9 right).

Figure 3.10. The “Equal Color” and “Equal Font” actions.

Finally, both the title and the page number are selected, and their color and font (i.e., both the typeface and point size) are changed so that they equal one another. This is accomplished by selecting appropriate attributes from the color and font palettes. The two panels illustrated in figure 3.10 display these actions and confirm that they are legitimate based on the grammar rules which describe the objects.

Similar to the bottom alignment action, the left and right alignment actions were also tested to determine if there was a possible relationship. Since there was, these actions were recorded and their panels created. Note that primal actions of moving, selecting, or simply resizing an object, by themselves, are not enough for the system to invoke recording the operations. Only those actions which are deemed valid by the grammar rules associated with the objects are recorded.
Now that the demonstration is over the designer must account for and disambiguate any ambiguities which have resulted. In this case there are only two. Since the above actions are needed to format section headlines and their page numbers as well as the chapter titles, the designer must illustrate to the system that the actions should work on any type of text object, not just chapter titles, and that when formatting a section headline, its horizontal position should be aligned to the left margin of the layout area, but is offset by a fixed amount to accommodate for the indent. To do so, the designer selects the panel of the graphical history which represents the left alignment action and sends it to the Generalization Editor. As mentioned briefly, the Editor allows the designer to edit data recorded for a particular action. When actions are initially recorded, the system's default behavior is to interpret them in relative, rather than absolute terms (i.e., the chapter title will be left aligned to the layout area's left margin regardless of where it lies, rather than at the absolute point of 100 pixels). When a panel is sent to the Editor, Abatan visually presents the recorded action by graphically altering elements in the Workspace to reflect how the system is currently interpreting them (figure 3.11). A textual transcription of the action is presented and can be accessed in the Editor. For the left alignment action, a vertical line is placed into the workspace directly on top of the layout area and the title's left edge to indicate to the designer this relationship. In the Editor, the action has been given the name "Align Left," it operates on two arguments, a layout area and a chapter title, and that the alignment position value equals the layout area's left edge plus a zero offset (figure 3.12).

![Figure 3.11. Abatan visually presents a recorded action by graphically altering elements in the Workspace to reflect how the system is currently interpreting them.](image)
The designer begins to disambiguate the action by first stating that the left alignment operation may have an offset value. To do so, the number 50 (which will be the indent value in pixels) is entered into the "Value" register of the Generalization Editor. This indicates to the system that it should override its default behavior of precisely aligning the objects and incorporate this offset when the action is both generalized and re-used. Next the designer shows the system that the entire action may be performed on any text object and not just titles by selecting the "Any Piece of Text" item from the Editor's "Arguments" menu (figure 3.12). After this selection is made and upon viewing the workspace, we now notice that the title has become a "generic text" object to reflect the action change (figure 3.11).

![Generalization Editor](image)

**Figure 3.12.** Editing a recorded action using the editor.

Once the ambiguities have been resolved, the information is encoded by selecting each appropriate panel in the Graphical History and clicking on the encode button in the Editor, from which the system produces a new domino icon in its interface that represents the concatenation of all actions selected in the history, and provides access to this chunk of encoded design knowledge (figure 3.13).

![Format Table of Content Items](image)

**Figure 3.13.** A procedure domino icon displaying "Table of Contents" design knowledge.
Now that the knowledge has been encoded the designer must test to see if it was indeed conveyed correctly. Three more page numbers, a chapter title, and two new section headline are created and placed into the Workspace. For variation sake to show that the knowledge encoded is generalized and only affects those relationships described during the demonstration, the new objects are given different typefaces and colors. The designer selects the new chapter title and one of the page numbers and clicks on the new icon just produced (figure 3.13), the results of which can be seen in the first line of figure 3.15. Here we see that the chapter title “Overview of the Project” has been positioned flush left to the margin, its page number is flush right, the two objects are bottom aligned, and the page number has the same typeface, point size, and color as the chapter title. Now the section headline “The Name Abatan” and another page number are selected and the same knowledge is applied, to which the desired response is produced; the section headline is indented 50 pixels, the page number is positioned flush right to the margin, and its typeface, point size and color are altered to equal the section headline’s. The same knowledge is applied to the remaining objects to format them appropriately. If this knowledge were defined using Microsoft Word’s Styles, the designer would be unable to reapply it to both the chapter titles and the section headlines. On the other hand, since Abatan generalizes the action, this is not a problem.
Now that we have seen Abatan in action, it is time to take a look at the underlying mechanics which drive the system. As stated in chapter two, there are essentially seven different components to the system; the Lexical and Grammatical Knowledge Bases (the Lexicon palette and the Exemplar Layout), the Design Environment (the graphical objects and other design tools), the Recording Mechanism (the recorder and the Generator Tests), the Graphical History, the Generalization Editor, the Learning Module (search and generalization), and the Output Mechanism (graphical procedures and symbolic rules). As was seen in the previous examples, the designer encodes knowledge by creating a layout which demonstrates the lower-order rules necessary for the knowledge to hold. The layout is constructed by selecting and assembling the graphical objects, arranging and manipulating their attributes, specifying relationships, and then finally recording, generalizing, and outputting the layout and the knowledge (figure 4.1).

In this chapter, the implementation and structure of the system are discussed in detail by analyzing its components.

Figure 4.1. The “encoding” pipeline.
The first stage in building the system was to identify the different groups of graphical objects that were to serve as the primary elements for creating the layout. In Abatan, three categories or classes of graphical objects are defined, TEXT, IMAGE, and GRAPHIC. Each class contains items which share common behaviors and general characteristics. All TEXT objects, such as the section headline in the “Table of Contents” example, contain information describing a literal string, a typeface, and a point size. GRAPHIC objects, such as the rulebar in the “Title” example, define a color and a fill pattern, and IMAGE objects state a default bitmap figure and a transparency value. Each class is defined hierarchically and inherits basic attributes from a single primitive called a SYMBOL. For each class of symbol there are several sub-classes which contain related types of graphical objects that are commonly used together in a layout (figure 4.2).

Figure 4.2. Symbol class hierarchy.
Figure 4.3. Symbols in Abatan are a three-part structure which describe a Behavior, a list of Attributes, and a Geometry.

As can be seen in figure 4.3, the super-class SYMBOL is a three-part structure defined by an Information Block, an Attribute List, and a Geometry. A symbol's Geometry is the rectangular bounding box which describes its spatial position in the layout and the total geometric area it covers. The list of Attributes describes the set of basic visual properties associated with the symbol, such as color or transparency. Lastly, each symbol contains an Information Block which states how it should "behave" and what role it plays in the layout. The text object used in the "Title" example was a subclass of the TEXT category. Its internal structure was as follows.

<table>
<thead>
<tr>
<th>Geometry</th>
<th>Attributes</th>
<th>Information</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vertical: 100</td>
<td>Helvetica</td>
<td>Type: Title</td>
</tr>
<tr>
<td>Horizontal: 300</td>
<td>black</td>
<td>Status: Selected</td>
</tr>
<tr>
<td>Width: 200</td>
<td>Color: black</td>
<td>Hot Spot: Left Edge</td>
</tr>
<tr>
<td>Height: 25</td>
<td>Point Size: 24</td>
<td>Behavior: Grow Taller</td>
</tr>
<tr>
<td></td>
<td>Justification: Left</td>
<td>Function: Title_Generator_Test</td>
</tr>
<tr>
<td></td>
<td>Opaqueness: 0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>String: Technical Work</td>
<td></td>
</tr>
</tbody>
</table>

Here we see that the text was positioned at Cartesian coordinates (100, 300) with the dimensions of 200 pixels (wide) by 25 pixels (tall). Its color was black and completely opaque. The font was typeface Helvetica at a point size of 24, and the text left justified. Its type was a sub-category of the TEXT class (i.e., a Title) and at the time it was selected and touched by the designer on its left edge. Finally, its behavior was that its bounding box should grow taller when the literal string contained inside it exceeded the space when edited, and was assigned the function Title_Generator_Test for use by the system with the grammar.
All three Information-Units (i.e., Geometry, Attribute List, and Information Block) are created as structures in the C programming language. The various slots contained in these structures are used internally by the system to physically display the objects on the screen and locally by an object itself to set and remember its own values. The actual structures for the three are given below. A symbol automatically inherits the BASIC_ATTRIBUTES structure. Depending on which class a symbol is instanced as, either the TEXT_ATTRIBUTES, IMAGE_ATTRIBUTES, or GRAPHIC_ATTRIBUTES are attached onto the BASIC_ATTRIBUTES to give the symbol its form.

```c
typedef struct{
    int relative_x;
    int relative_y;
    int width;
    int height;
}GEOMETRY;

typedef struct{
    int type;
    int status;
    int where_touched;
    int (*func)();
}INFORMATION;

typedef struct{
    int red;
    int green;
    int blue;
    int transparency;
}BASIC_ATTRIBUTES;

typedef struct{
    int red;
    int green;
    int blue;
    int transparency;
    char *color_string;
}TEXT_ATTRIBUTES;

typedef struct{
    int mask_color;
    char *default_bitmap_path;
}IMAGE_ATTRIBUTES;

typedef struct{
    int fill_pattern;
    int graphic_type;
}GRAPHIC_ATTRIBUTES;
```

/* symbol's bounding box */
/* box's relative x coordinate */
/* box's relative y coordinate */
/* box's width */
/* box's height */

/* symbol's information block */
/* what type is it? */
/* is it selected? */
/* where was the symbol touched */
/* generator test function */

/* basic set of attributes */
/* red color value */
/* green color value */
/* blue color value */
/* transparency value */

/* text symbol's attributes */
/* string's red color value */
/* string's green color value */
/* string's blue color value */
/* string's transparency value */
/* justification of the text */
/* font point size value */
/* font typeface */
/* the object's literal string */

/* image symbol's attributes */
/* background masking color */
/* the object's literal string */

/* graphic symbol's attributes */
/* fill pattern type */
/* rectangle, triangle, etc */

Capturing Graphic Design Knowledge from Interactive User Demonstrations 49
Symbols themselves are “self-aware.” They can alter their own Information-Unit values or can be told by the system to conform themselves, in which case they “know” what to do. For example, an object can either have its color explicitly changed, by having the designer select a new color from the palette, or it can be instructed by the system to color itself, in which case the object knows how to follow the system’s instructions and do so. In the “Title” example, when the text and rulebar were having their colors altered, each symbol individually told itself “a color selection has been made; I am currently selected; I better update the red, green, and blue color slots in my attribute list to reflect the new changes.” Symbol self-awareness is achieved by attaching commands or methods to the symbol. A method defines the behavior of a symbol’s slot and is able to update, retrieve, or set its values [Khoshafian 90]. The method used to update the text's color slots is shown below.

```c
int update_current_color_method(SYMBOL *text_symbol)
{
    SendMessage(text_symbol, SET_COLOR, GetCurrentColor());
}
```

The above method tells the text symbol to send itself the message “SET_COLOR” to update its color based on the values returned from the function `GetCurrentColor`, which retrieves the currently selected color in the system's color palette. Depending on the category of symbol, different methods are attached to the object to perform operations such as these.

![Figure 4.4. The nine touch sensitive “hot spots” of a graphical object.](image)

Figure 4.4. The nine touch sensitive “hot spots” of a graphical object.
In addition to being self-aware, and as mentioned earlier, symbols are also touch sensitive. Each time the designer clicks on a symbol that object knows where it has been touched and what it should do. For instance, if an object's edge is touched, it will tell itself that it needs to perform a move operation; if the object's corner is clicked on, it knows that it has to perform a resizing operation. Figure 4.4 illustrates the nine touch sensitive areas of an object. Touch sensitivity and hot spots of objects used in Abatan are similar to the snap-dragging [Beir 86] technique of triggers, where vertices and line segments of an object are used to inform the system of specific alignment constructions. The following illustrates how touch sensitivity is implemented. First a high level method is attached to the object to determine where it has been touched (i.e., its hot spot). This method highlights the appropriate edge or corner, and depending on the type of hot spot (edge, corner, or center) it automatically calls the appropriate move or resizing method.

```c
int SelectHitPoint(SYMBOL *S, int x, int y)
{
    if(x < S->x1 + 10){
        if(y < S->y1 + 10)
            SetHitPoint(S, TOP_LEFT);
        else if(y > S->y2 - 10)
            SetHitPoint(S, BOTTOM_LEFT);
        else
            SetHitPoint(S, LEFT_EDGE);
    }
    else if(x > S->x2 - 10){
        if(y < S->y1 + 10)
            SetHitPoint(S, TOP_RIGHT);
        else if(y > S->y2 - 10)
            SetHitPoint(S, RIGHT_EDGE);
        else
            SetHitPoint(S, RIGHT_EDGE);
    }
    else if(y < S->y1 + 10)
    else if(y > S->y2 - 10)
        SetHitPoint(S, CENTER);
}

int touch_method(SYMBOL *S, PICKSTRUCT *mouse)
{
    int hit_point = NULL;
    if(mouse->button == JUSTDOWN){
        if(hit_point = SelectHitPoint(S, mouse->x, mouse->y))
            SendMessage(S, HIGHLIGHT, hit_point);
        if(event(hit_point))
            SendMessage(S, MOVE, hit_point);
        else if(odd(hit_point))
            SendMessage(S, RESIZE, hit_point);
    }
    return(0);
}
```
In addition to the touch sensitive move and resize method just mentioned, symbols, depending on their class, have several other methods attached to themselves to accomplish other behaviors, such as changing their typeface, transparency, or point size. The following list describes the different categories of methods associated with each symbol class. These methods are used by both the objects and the system to incorporate attribute changes.

<table>
<thead>
<tr>
<th>Category</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Typeface</td>
<td>The “Typeface” method is attached to all TEXT symbols and allows these objects to get and set the typeface slot of their attribute list.</td>
</tr>
<tr>
<td>Point Size</td>
<td>The “Point Size” method is attached to all TEXT symbols and allows these objects to get and set the point size slot of their attribute list.</td>
</tr>
<tr>
<td>Color</td>
<td>The “Color” method is attached to all TEXT and GRAPHIC symbols and allows these objects to get and the red, green, and blue slots of their attribute list.</td>
</tr>
<tr>
<td>Transparency</td>
<td>The “Transparency” method is attached to all symbols, TEXT, GRAPHIC, and IMAGE, and allows these objects to get and set their transparency value.</td>
</tr>
<tr>
<td>Move</td>
<td>The “Move” method is attached to all symbols and allows these objects to get and set the positional slot values located in their Geometry. Furthermore, this method is capable of manipulating a symbol's position to one of several specific locations, such as “above” or to the “left-of” another object.</td>
</tr>
<tr>
<td>Resize</td>
<td>The “Resize” method is attached to all symbols and allows these objects to get and set the dimensional slot values located in their Geometry. This method is also able to manipulate the symbol's dimensions, making it the “same-width-as” or “same-height-as” another object.</td>
</tr>
</tbody>
</table>
KNOWLEDGE BASE

Built into Abatan is a knowledge base which defines different lexical categories of related graphical objects and states grammatical rules describing how these objects should be arranged in the layout. The knowledge base is part of the larger Learning Module and is used by the system to both aid and restrict the designer when creating a layout. At present, three different lexicons and associated grammars are defined, one for technical document design, a second for newspaper layout, and a third for creating Scientific American articles.

Currently, both the lexicons and the grammars are hard-coded into the system. While this is a limitation to the designer (being unable to add new items or edit existing ones) it was done so for two distinct reasons. First, interactive editing of the lexicons and grammars were beyond the scope of this thesis given both the focus and the time restriction. As will be mentioned in the Future Directions section of chapter six, research to tackle this problem, discussed in [Lieberman 93b], is currently underway.

As a design decision, since Abatan is still in its infancy, the second reason for the limitation was so that the system would learn within a limited scope. As it grew, it would hopefully build off of what it had already learned, using it as a solid base from which to work off of (as opposed to randomly learning a vast range of rules which may or may not have relevance to one another).

LEXICAL KNOWLEDGE

The lexical knowledge base used within Abatan defines related groups of graphical objects commonly found in specific types of layout. The technical document lexicon is made up of various titles, headlines, subheadlines, diagrams, rulebars, and body texts, each with different features. The one for Scientific American articles has titles, authors, bullets, figures, diagrams, and descriptions defined as its lexicon. The final one, for newspaper layout, contains a mixture of the previous two.
The lexical knowledge base is simply a pre-defined, pre-encoded "lookup" table. The primary use of the lexicon is by the system as a means of distinguishing and defining groups of objects, the actual members of which are illustrated in the bottom tier of figure 4.2. To distinguish objects in these groups, each symbol is given a distinct id number and placed into a list which states what other objects are related to it. This number states the object's class (TEXT, IMAGE, or GRAPHIC), the lexical family to which it belongs (technical document, newspaper, or Scientific American article), the object's subclass (title, headline, rulebar, etc), and a precedence ordering.

In addition to grouping the objects, the id number also serves to organize the graphical objects, present within a particular group, hierarchically, giving them a precedence ordering or a "level of importance." Graphical objects can take on different levels of importance depending on the type of layout they are in. The id number is used to state how important the object is to the overall layout. This id number is primarily for use with the grammars and is discussed in more detail in the upcoming Grammatical Knowledge section.

![Lexicon](image)

Figure 4.5. Three different states of the Lexicon palette (the default and two different instancing states).
The visual counterpart to the lexical knowledge base is the Lexicon palette. The designer loads one of the lexicons by selecting the appropriate item from the palette’s menu (figure 4.5 left). The purpose of the palette is to display the types of objects contained within a particular group and to illustrate what the objects' visual attributes are. As can be seen in figure 4.5, for *Scientific American* articles the Lexicon palette displays seven graphical objects, each of which can be given distinct attributes (figure 4.5 center, right).

The Lexicon Palette is simply a visual index and place holder for instancing graphical objects into the Workspace. The index aspect allows the designer to request a general category of typed symbols, from which the system highlights specific instances of them in the Exemplar Layout. As will be seen in the next section, a layout may have several different instances of the same type. For example, in a newspaper layout there may be two headlines present; one whose typeface is Helvetica and whose function is that of a main headline used to summarize the whole page, and another whose typeface is Times-Roman and whose purpose is to describe a specific story. The place holder aspect is one in which the specific instance's properties and attributes are transferred over and stored into the general item (the object in the Lexicon palette) showing to the designer what the appearance of such an object will look like before it is placed into the layout (figure 4.5 center, right). The palette is physically comprised of a single lexicon of graphical objects. When an object is created in the Workspace, its values inherit those found in the three Information-Units of the currently selected object in the Lexicon palette. Similar to how the a lexicon object has its attributes updated from a corresponding object found in the Grammar palette, a newly instanced object in the layout has its default values set to equal that of the lexicon object from which it is based.
Grammatical Knowledge

The grammatical knowledge base used in Abatan is essentially a series of rules and functions that specify post-condition relationships under which elements in a layout should be arranged. As mentioned in chapter two, the grammars are primarily for use by the system and serve as a "safety net," to restrict what the system can and cannot learn. In the previous section describing the graphical objects, contained within each symbol's Information Block is a function. This function is part of a larger network called Generator Tests. Generator Tests are used as a means of filtering what actions are recorded by the system, based on the grammar rules which describe an object. Depending on the type of object, different generator functions are assigned to tell the object what its rules are, how it should behave, and how far it can be manipulated. In the "Title" example, the text's generator function was called Title_Generator_Test. This function uses the rules which describe a title's properties and makes sure that they are always maintained when the designer is manipulating the object. For the headline object used in technical document designs, its grammar rules are defined as:

1. The headline has the second highest precedence order.
2. The headline width cannot exceed that of a title's.
3. The headline point size cannot exceed that of a title's.
4. The headline is placed below a title unless there is a rulebar directly underneath the title, in which case the headline is put below the rulebar.

If the designer performs an action which breaks these rules, the system will ignore it and not record the action. If the rule's specifications hold after the action has been performed, then the system records the action and provides the feedback of creating a new panel in the Graphical History window; otherwise no visual response is given.

The grammar rules are the driving force behind the Generator Tests. Their representation in the system is not as a global list of rules (where only a single instance of each rule is present) that is applied, by the system, to the current state of the design, (which is commonly how grammars are used). Instead, the grammar rules are placed into the local methods attached to the individual objects. When an object is manipulated, that object invokes its generator function which then tests to see if the grammar rules associated with that object can be fired.
For example, in the grammar which describes the layout style of technical documents, a rulebar is always placed below a piece of text. The rule definition for positioning a rulebar in a technical document is as follows.

```
int position_rulebar(SYMBOL *rulebar, SYMBOL *layout)
{
    SYMBOL *temp;
    if(temp = is there a symbol of type(TITLE | HEADLINE, layout))
        put_below(rulebar, temp, 0);
}
```

The above rule reads as: “If there is a title object or a headline object present in the layout, then move the rulebar to a position directly under it at an offset of zero.” If the test for finding a title or headline is true (i.e., there is one already present in the layout), then the rulebar is told to send a message to move itself under the object.

Rules used in the grammar take the form of a condition-action rule. A rule's condition may be one of three types of tests; either a binary test such as “Are X and Y aligned?”; a range query, “What is X above?”; or a domain query, “What is above X?” The following list states all of the different tests presently used by the grammars. Note that binary tests take two symbols as arguments and return an answer of either true or false. Domain and range queries take a single symbol as an argument and return a list of all symbols which meet the criteria.

**Binary Tests:**
- `is_left_align`
- `is_bottom_align`
- `is_below`
- `is_left`
- `is_width_equal`
- `is_same_color`
- `is_same_size`

**Domain Queries:**
- `what_is_below`
- `what_is_left`
- `what_has_same_width`
- `what_has_same_color`
- `what_has_same_size`

**Range Queries:**
- `above_what`
- `left_of_what`
- `same_width_as_what`
- `same_color_as`
- `same_point_size_as`
- `below_what`
- `right_of_what`
- `same_height_as_what`
- `same_typeface_as`
- `same_transparency_as`
The post-condition actions of a rule are obtained by sending the target object a specific message stating how it should conform itself. In the position_rulebar rule stated above, the post-condition put_below(rulebar, temp, 0), is a function which retrieves "temp's" (a symbol) current location in the layout and then tells the rulebar to move itself to a position below it at a zero offset. The actual code for the put_below function, listed below, reads as: “Ask the anchor symbol what its current vertical Cartesian coordinate is and store this information in the variable y_position. Then tell the target symbol to set its vertical position to a location below the value stored in y_position and offset it by the amount stored in the parameter offset.”

```
int put_below(SYMBOL *target_symbol, SYMBOL *anchor_symbol, int offset)
{
    int *y_position;
    SendMessage(anchor_symbol, GET_CURRENT_POSITION, y_position);
    SendMessage(anchor_symbol, PUT_BELOW, y_position);
    SendMessage(anchor_symbol, OFFSET_BELOW, offset);
}
```

Generator Tests are defined as methods and are automatically invoke each time an object is manipulated. The Generator Test method for the technical document headline is as follows. Note, that if all tests within the method fail and an object's values have not been set, the default settings are used.

```
int Technical_Headline_Generator_Test(SYMBOL *headline, SYMBOL *layout)
{
    SYMBOL *title, *rulebar;
    if(is_precedence_order_set(headline) != TRUE)
        set_precedence_order(headline, 8);
    if(are_all_variables_set(headline) != TRUE)
        set_all_variables(headline);
    if(title = is_there_a_symbol_of_type(TITLE, layout))
        if((rulebar = what_is_closest_below(title)) == RULEBAR)
            put_below(headline, rulebar, 0);
        else
            put_below(headline, title, 0);
    if(is_greater(headline, title, WIDTH))
        make_less_than(headline, title, WIDTH);
    if(is_greater(headline, title, POINT_SIZE))
        make_less_than(headline, title, POINT_SIZE);
}
```
The above Generator Test method is read as: “If the headline's precedence order is not set, set it to be the second highest with a value of eight. If all of the symbol’s variables are not set, set the symbol's variables based on the default settings. If there is a title in the layout, first test to see if there is a rulebar directly below it. If so, place the headline directly below that at a zero offset, else place the headline directly below the title at a zero offset. If the headline's width is greater than that of the title's, reduce its width to a value less than it. If the headline's point size is greater than that of the title's, also reduce its point size to a value less than it.”

The secondary use of the grammars within Abatan are as “safety checks.” For the designer these checks make sure that all objects, when placed into the Workspace, have all of the necessary variables set. When instanced, if a graphical object has not had all of its Information-Unit values set, the system automatically uses the object's id number to check in the grammatical knowledge base to determine its default settings. Figure 4.6 illustrates examples of functions used for this purpose. Depending on the symbol’s class, different default settings are defined.

```c
int KB_point_size(int symbol_id)
{
    switch(symbol_id)
    {
        case TITLE: return(24); break;
        case HEADLINE: return(18); break;
        case SUBHEAD: return(12); break;
        case AUTHOR: return(10); break;
        case BODY: return(6); break;
        default: return(3); break;
    }
}

char *KB_color(int symbol_id)
{
    switch(symbol_id)
    {
        case TEXT: return("Black"); break;
        case GRAPHIC: return("Gray"); break;
        default: return("White"); break;
    }
}
```

*Figure 4.6. The grammatical knowledge base default-settings look-up table.*
The Exemplar Layout displayed in figure 4.7 illustrates an example of one grammar, contained in the knowledge base, applied to format a page of a technical document. As mentioned earlier, this Layout is interconnected to the Lexicon palette. The interaction between the two is achieved by sending messages back and forth between corresponding objects. For instance, when the designer selects a general category of objects in the Lexicon palette, the system sends a message to all objects in the Exemplar Layout to see if they are of the same class. If so, those objects are told to highlight themselves, revealing to the designer all of the different choices available (figure 4.7 center, right). The primary use of the Exemplar Layout is to visually illustrate to the designer the effects of applying the grammar to a layout and to specify different states and attributes associated with a specific type of object. The physical makeup of the Exemplar Layout is the same as the Lexicon palette, simply a group of graphical objects identical to those used when constructing the example layouts. When the Exemplar Layout is created by the system upon startup, the symbols contained within it do not have their Information-Units explicitly set. Instead, one of the grammars is applied to the palette to relatively set constraints between the objects. Each object’s values are set by individually telling each object to apply its generator function to itself, which in turn applies the grammar rules. Since a specific type of action has not been performed on the object the generator function only applies the default settings, in turn, formatting each object appropriately.

Figure 4.7. Three different states of the Exemplar Layout; completely unselected, with all headline objects highlighted, and with all body text objects highlighted.
The entire designer process used to create the example layouts can be considered, in its simplest form, just a sequence of events over time. From this sequence, much information about the finalized layout, the object's used, and their relationships to one another can be gathered. The Recording Mechanism is what obtains and stores, in a textual form, this information. The Recording Mechanism is an agent-like [Cypher 93] device which translates the designer's graphical actions into low level procedural calls and arguments. Recording an action in Abatan is an interconnected process; a joint effort between the Recording Mechanism, the grammar, the graphical objects, and the designer. First the designer performs some type of manipulation on an object, after which that object checks the grammar rules which describe it (using its generator function) to see if indeed the action was within bounds. If so, that object sends a message to the Recording Mechanism stating what had happened and the results.

Recorded events are stored in a C structure called an ACTION (figure 4.8). The action describes the type of relationship which resulted, the arguments which the action was performed on, the absolute values initially determined, how the action should be performed when re-applying it, and finally a system given, default name in which to reference the action.

To demonstrate its use, recall the action of bottom aligning the chapter title and the page number in the “Table of Contents” example. Initially, the chapter title was selected as the anchor object and its bottom edge clicked upon. Then the page number's bottom edge was selected and that object was dragged downward to a position roughly equal to that of the chapter title. From this information, the following was recorded.

<table>
<thead>
<tr>
<th>Action</th>
<th>Name:</th>
<th>“Align Bottom”</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Id:</td>
<td>BOTTOM_ALIGN</td>
</tr>
<tr>
<td></td>
<td>Values:</td>
<td>200</td>
</tr>
<tr>
<td></td>
<td>How:</td>
<td>Relative</td>
</tr>
<tr>
<td></td>
<td>Arguments:</td>
<td>chapter title, page number</td>
</tr>
</tbody>
</table>
Rather than keeping the Recording Mechanism on all the time, recording each action, Abatan's approach to recording the design process is "symbol-oriented." If we look at the bottom alignment action stated above in detail, we can see how the symbols are the ones "directing traffic."

**Action:** The designer clicks on the chapter title's bottom edge.
**Result:** The chapter title sends a message to itself stating where it has been clicked, that it was the first object selected (remembering that it is the anchor object), and that its bottom edge was touched. This information is stored in the `where_touched` slot of the symbol's Information Block.

**Action:** The designer clicks on the page number's bottom edge.
**Result:** The page number sends itself a message stating where it has been touched and stores this in its `where_touched` slot.

**Action:** The designer drags the page number's bottom edge downward until it is approximately horizontally aligned with the section headline's bottom edge.
**Result:** Since the manipulation action is directed towards the page number, it calls its generator function to see if the action is legitimate enough to record. In its generator function, the rules which describe page number alignment (in the *Scientific American* grammar) state that it can be aligned to any object of greater precedence ordering and that it can not be an anchor object for an alignment action since it has the lowest precedence ordering. Since the action is valid, the page number and the section headline both send messages to the Recording Mechanism stating what had happened. When the Recording Mechanism receives this data, it first checks to see if a possible relationship is present. It uses each object's hot spots (i.e., where it was clicked) to determine what type of operation was performed (i.e., corner hot spots indicate resizing, edge hot spots indicate move, etc) and what type of relationship to look for. In this case, since both bottom edges were selected and their finalized positions after performing the move operation were equal, the Recording Mechanism determined the bottom alignment relationship and recorded the action.

```
typedef struct {
  SYMBOL *panel;
  SYMBOL *values;
  LIST *arguments;
  char *name;
  int how;
} ACTION;
/* recorded action */
/* the action's graphical history panel */
/* action data */
/* list of objects which action affects */
/* reference name */
/* interpret relative or absolute */
```
GRAPHICAL HISTORY

Once an action has been recorded, Abatan creates a miniature action panel to provide the designer with feedback, illustrating that the system has recorded something, and providing a means of access to the recorded data. In the Graphical History, action panels serve as visual counterparts to what the Recording Mechanism's has recorded, depict one user action, and display the current state of the design at that point. The action panels are not merely static bitmap snapshots of the screen, but actual collections of miniature graphical objects modeled after those found in the Workspace.

To physically create a panel, Abatan simply asks all objects currently located in the Workspace for their three Information-Units, from which, new identical objects in the panel are created. The only difference is that the new object's dimensions are reduced to provide thumbnail views of each action. The reason for reproducing the entire layout, rather than only those objects involved with the action, is two part. First, to set up a proper viewing context, and second, to enable the system to re-configure the present state of the design to reflect that found in the panel, allowing the designer to perform multiple "undos." Transforming back to a previous state is achieved in a similar manner to the way the panels are created in the first place. All objects in a panel are asked for their three Information-Units, then the system finds the corresponding objects in the Workspace and tells each object to update its values based on the appropriate Information-Unit, by sending the objects a series of messages.

The recorded action is attached to the panel as a method, similar to those which work on an object's attribute slots. The panel can be asked to set, retrieve, or update its action, in addition to being asked to apply it to the current state of the layout. For example, once the bottom alignment action has been recorded, if the designer moves the chapter title to a new position, the action can be re-applied to automatically move the page number so that it is bottom aligned with the new position by sending the "Align Bottom" panel in the Graphical History window a re-application message.
For each panel, the designer can instruct the system to either "remember" that action absolutely (always align symbol A and symbol B at point C, where point C is an absolute coordinate on the page) or relatively (wherever symbol B is, align symbol A to it) by selecting the appropriate item from the panel's menu (figure 4.9 left). This means that the system essentially offers the designer two different types of interpretations for their actions. Further editing of an action, such as aligning symbol A to symbol B at an offset of fifty pixels, can be accomplished using the Generalization Editor, which is described in the next section.

The designer is not restricted to having all panels interpreted in the same way; they can mix and match as they please. Finally, the designer is able to edit the individual panels as they would objects found in the current state of the design. Objects can be resized, moved, and have their attributes changed all through direct manipulation (figure 4.9). The only difference is that the system does not record these actions. In this case, the changes are incorporated back into the actual action method attached to the panel.
The heart of the system is the Learning Module. It is here that the system analyzes the recorded data and then generalizes this information to produce high level procedural calls and arguments. The front-end to the Learning Module is the Generalization Editor and the meat of this component is a series of conversion functions and search algorithms. The Learning Module uses information from the knowledge bases to guide it in sorting out the data contained in the recordings. The Editor is used before sorting takes place to further state specifications if needed. Once sorted, the system uses the generalization functions to transform each slot of an action into an abstract representation which enables the system to re-use what has been encoded. By the time the “encoding” pipeline reaches the Learning Module, all primary actions, not including those performed while editing, are assumed to be complete and ready for generalization.

As stated in chapter two, the Generalization Editor is an interactive dialog box that allows the designer to edit the data stored in a recorded action and clarify any ambiguities which may have arisen due to under specification. For each action that is recorded, initially Abatan offers the designer two different interpretations, relative and absolute. With the help of the Editor, the designer can view and alter these interpretations further, stating explicit specifications and conditionals. In simplistic terms, the Editor is no more than a “language interpreter” which allows the designer to “speak” with the computer code, directly manipulating the data stored in the system.

Actions are edited by loading their panel of the Graphical History into the Editor. This process is achieved by sending the panel a message asking it for its recorded action, from which the panel supplies the data. Once received, the Editor uses the action's id number to update its visual appearance accordingly. For spatial and dimensional actions, numerical input registers are displayed, allowing access to the stored values. For typographic actions such as transparency or point size changes, menus are used.
Figure 4.10 illustrates the left alignment action, from the "Table of Contents" example in chapter two, as it appeared in the Editor. The arguments to the action are listed sequentially in the "Arguments" menu based on their precedence ordering, where the anchor object is placed at the top of the list. For each argument listed, when clicked on, a sub-menu appears stating several methods of how that object can be view in terms of generalization.

For the chapter title in the left alignment action, three menu items are offered; "Only This Chapter Title," "Any Chapter Title," or "Any Piece of Text." Above the "Arguments" menu is another displaying the spatial parameters of the action. At present, the "Values" register displays the value of 50 (the vertical Cartesian coordinate of the layout area's left margin in pixels). If the designer clicks in this, three choices are displayed; "Interpret Value to be 50" (i.e., the current default choice), "Interpret Value to be Anchor Object's Left Edge," or "Alter Value." If the designer selects the first choice, the graphical objects affected by the action will always be aligned at 50. If the second choice is selected, the value supplied will be an anchor object's left edge, where ever that may be at the time. The last selection allows the designer to edit the "50" value and supply the system with a new absolute position by directly typing into the register located next to it.

Figure 4.10. The Generalization Editor.

If a color change action is sent to the editor, its visual display would offer different choices. The "Arguments" menu would still remain the same, but rather than presenting a register or a menu, the designer is able to directly select a new color from the color palette, from which the "Values" section would be updated to reflect the new selection. The same is true for typeface changes also.
In addition to this dialog-like interaction, once an action has been loaded into the Editor, the system visually alters objects in the Workspace to illustrate how it is currently interpreting them and to provide further feedback to the designer, as well as a means of editing. For spatial and dimensional actions, guide lines are overlaid onto the workspace for further emphasis. The designer is able to edit an action by re-positioning the guide lines (figure 4.11). For example, the designer may wish to alter the “50” value to a position half way across the page, but not know what that exact value is. If the guide line, which is also a graphical object, is re-positioned to this new location, Abatan will automatically calculate the value by asking it for its new position and display this information in the Editor.

![Figure 4.11. Visual manipulation editing of a recorded action.](image)

When the designer edits a value, either directly in the Editor or via the visual reinforcements in the Workspace (i.e., guidelines, etc.), these changes are directly incorporated into the action and may be used instantly.
The actual calls used to send an action to the Editor are invoked by the
designer by physically dragging the panel to a position on top of the Editor
(figure 4.12). Once this drag-and-drop action is performed, the following
method is called to "get" the data.

```c
ACTION *LoadPanelActionIntoEditor(SYMBOL *action_panel)
{
    ACTION **temp;
    SendMessage(action_panel, GET_ACTION, (temp);
    SendMessage(Editor, LOAD_ACTION, temp);
}
```

where the "LOAD_ACTION" message invokes the following messages:

```c
SendMessage(Editor_Id_Menu, LOAD_ID, get_action_id(temp));
SendMessage(Editor_How_Menu, LOAD_HOW, get_action_how(temp));
SendMessage(Editor_Argument_Menu, LOAD_ARGS, get_action_args(temp));
SendMessage(Editor_Arguments_Menu, LOAD_VALS, get_action_vals(temp));
```

It is not a requirement for the designer to edit each and every action.
Those which are not are generalized based on their present data. Actions
which have been edited are generalized using the new (edited) data.

Figure 4.12. "Sending" an action panel to the Editor using the drag-and-drop
technique.
GENERALIZATION

If the knowledge is to take on any type of robust form, it must first be generalized. Simple recording of a demonstration produces a heavily detailed, frozen representation of the exact sequences of actions performed, which in turn only enables re-application in a macro-like manner. This restricts the knowledge to only work in identical contexts, from which the same results are always produced each time.

The goal of generalization in Abatan is to strip away the example specific detail to produce an abstract representation that preserves the underlying idea, but will allow the knowledge to adapt and conform itself when being reapplied to new situations. This involves converting the recorded actions' slots, by substituting absolute values with relative relationships, into high-level descriptions which are modeled after, but not identical to, the exact situation from which they were defined (i.e., the demonstration) [Abelson 85] (figure 4.13). For example, if the knowledge encoded for the “Table of Contents” example was recorded in a macro-like fashion, it would only allow the designer to format the objects one way, where the chapter title would always be positioned at coordinates (100, 200), as opposed to being relative to the page’s left margin; the page number would always be positioned at (700, 200), instead of relative to the right margin; the two objects would always be colored black, instead of always “equaling one another,” and so on. In addition, the knowledge, when reapplied, would only work on a single chapter title and page number explicitly. It would be unable to handle a variable number of entries and would require that the designer explicitly reapply the knowledge to each and every object. Furthermore, it would only be able to work on either a chapter title or a section headline, not both.

<table>
<thead>
<tr>
<th>Action</th>
<th>Generalized Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name: “Align Left”</td>
<td>Name: N/A</td>
</tr>
<tr>
<td>How: Relative</td>
<td>How: Relative</td>
</tr>
<tr>
<td>Id: LEFT_ALIGN</td>
<td>Id: LEFT_ALIGN</td>
</tr>
<tr>
<td>Values: 200</td>
<td>Values: search_for_anchor(LEFT_EDGE)</td>
</tr>
<tr>
<td>Arguments: rulebar, headline</td>
<td>Arguments: search_for(RULEBAR) search_for(HEADLINE)</td>
</tr>
</tbody>
</table>

Figure 4.13. “Before” and “after” generalization states of an action.
Instead of viewing an action literally, such as “a piece of text with the string “Introduction” was moved from point (100, 100) to point (200, 100),” Abatan can generalize this information into the form that “a title object was moved 100 pixels horizontally to the right,” which will enable any title, not just those with the string “Introduction,” to be moved 100 pixels to the right of its present position, wherever that may be.

Generalization, as commonly used in Programming by Demonstration, is applied to recorded data to produce a function, such as that shown below, which is much more robust than a macro recording.

```
(defun color-object (object)
  (if (> (send object :value) 70)
    (if (> (send object :value) 80)
      (send object :set-color Red)
      (send object :set-color Orange))
    (send object :set-color Yellow))
```

The above function, adapted from [Ishizaki 89], reads as: “Test to see if the object's value is greater than 70. If so, test to see if it is greater than 80. If it is, send the object a message to color itself red, otherwise tell it to color itself orange. If the object's value is not greater than 70, color the object yellow.” In this case, the argument “object” is an object of variable type.

This function allows the knowledge to work in several different situations and offers several executional behaviors given the appropriate context. If the function were produced from a macro recording, it would only be able to illustrate a single method of performing the actions (i.e., it could only handle one of the conditions and would not be able to work on objects of variable type), in which case the reapplication situation would have to be identical in order for it to perform properly. On the other hand, a generalized description can adapt to analogous, but not necessarily identical situations, and offer variations that will work.
In Abatan, generalization is performed each time the knowledge is reapplied, by running the list of recorded action data through a function, called *Generalize_Action_Data*, which converts the information stored in an action structure into a series of search functions [Rich 91] and messages. The function, illustrated below, takes as its sole argument a list of actions. It sequentially looks through the list and applies the recorded actions, as specified and disambiguated by the designer with the help of the Generalization Editor, to the objects contained in the action's arguments list.

```c
int Generalize_Action_Data(ACTION_LIST *actions)
{
    SYMBOL *anchor, *temp;
    SearchList(actions->arguments)
    if (anchor = get_anchor_object(actions->arguments))
        if (temp = search_for_entries_in_list(actions->arguments))
            SendMessage(temp, actions->id, actions->values, anchor);
}
```

This function reads as: “While traversing down the list, for each action get the anchor object from the list of arguments. Then go through each action and send the arguments of that action a message based in the action's id type, the anchor object, and the action values.” The difference between this method and other methods of generalization commonly used in Programming by Demonstration systems, is that generalization can be performed using search functions that search for objects based on the parameters supplied as arguments to the function, rather than using only attribute slots or absolute values. For example, in the earlier *Example Based* graphical programming system described in [Ishizaki 89], a generalized function call and parameters would take the form of:

```
(send argument-1 :set-color (color (argument-2)))
```
“Send argument-1 a set color message with the present color value of argument-2,” where both argument-1 and argument-2, which are graphical objects, would be supplied as parameters to the function beforehand. In Abatan, this call would be executed as:

\[
\text{SendMessage(target\_symbol, \text{SET\_COLOR, color\_argument});}
\]

where "target\_symbol" could be either an explicit object or a description of an object which the system should look for in the current layout, such as "search for a title objects whose color is red." "Color\_argument" can also be either a graphical object, in which case it would do exactly the same as the previous statement, an absolute value, a function which returns a value, or a search function and a parameter. For instance, all four of the following calls are valid.

\[
\begin{align*}
\text{SendMessage(target\_symbol, \text{SET\_COLOR, GetCurrentColor();});} \\
\text{SendMessage(target\_symbol, \text{SET\_COLOR, Black});} \\
\text{SendMessage(target\_symbol, \text{SET\_COLOR, headline});} \\
\text{SendMessage(target\_symbol, \text{SET\_COLOR, search\_for\_entries\_in\_list(TITLE));}}
\end{align*}
\]

In the first case, the values returned from the function \textit{GetCurrentColor}, which returns the currently selected color at the time, is used as the argument. The second case simply uses an explicit color value and the third uses the present color of the symbol headline. The fourth allows for a search function, which returns values based on the arguments supplied as parameters, to be used. The function \textit{search\_for\_entries\_in\_list} searches in the current layout for objects whose attributes meet the criteria of the parameter and defines the search space based on the arguments supplied as parameters. For example, one search may look for only title objects whose color is red. Another may look for any object whose color is red, and a third variation might look for only title objects, regardless of their color. Search functions in Abatan use the binary tests, range queries, and domain queries stated earlier in this chapter to guide them in their work.
The important difference of this method of generalization is that search functions and parameters are determined at the time of re-application and not directly after the demonstration is completed. The parameters used in the search functions are determined by converting the graphical objects' (the arguments to the action) attribute slots into high-level, distinguishable descriptions. Recall the bottom alignment operation whose resulting action was:

**Action**
- Name: "Align Bottom"
- Id: BOTTOM_ALIGNMENT
- Arguments: chapter title, page number
- Values: 200
- How: Relative

As is, this action's data is generalized into the following parameters:

**Action**
- Name: N/A
- Id: BOTTOM_ALIGNMENT
- Arguments: search_for(TEXT), search_for(PAGENUMBER)
- Values: search_for_anchor(BOTTOM_EDGE)
- How: Relative

For the **Values** slot, the number 200 has been replaced with the function `search_for_anchor(BOTTOM_EDGE)`, which searches for the current anchor object, whatever that may be at the time, and figures out where its bottom edge lies. In the **Arguments** list, the chapter title has been replaced with a search function that takes as its parameter a generic "TEXT" object since the designer had edited the action to do so earlier. The last argument, the page number, has been replaced by another search function which takes as its parameter a PAGENUMBER, taking into account the precedence-order value also.

The designer invokes the generalization process by selecting all of the appropriate (wanted) action panels in the Graphical History window and then clicking on the encode button in the Generalization Editor. The results of which are produced by sequentially generalizing each action panel (in the order they were selected) and concatenating them together into a list which is used by the `Generalize_Action_Data` function when re-application takes place.
As mentioned earlier, there are two different means of testing the encoded knowledge. The first is from within Abatan, where the designer formats a new layout by applying the encoded knowledge to the graphical objects, and the second method is to output the computer code generated from the demonstration and incorporate it into existing Design Support system for use as a grammar.

Once a series of actions has been encoded, the system produces a new domino icon (figure 4.14) in its interface which allows the designer to re-apply this information to new arguments from within Abatan. The domino, which is a two part icon (left and right sides), is categorized in [Lieberman 93a] as “example-oriented” icons, where the left side of the domino states the types of graphical objects which the knowledge affects (i.e., the pre-conditions necessary for the rules to operate on) and the right hand side illustrates the result of applying the operation to the objects (i.e., the post-condition relationships). The right hand side of the domino is created in the same manner as the action panels in the Graphical History window; by gathering each graphical object's Information-Units and creating new objects based on them. The left hand side is determined by looking through each action's argument list to find out what the objects' classes are, from which new graphical objects are manufactured. The entire list of recorded/edited actions is attached to the icon as a method, similar to those attached to the action panels, and is invoked when the designer clicks on the icon. After doing so, the list of actions are sent to the Generalize_Action_Data function and applied.

Figure 4.14. A domino icon.
Let us assume that the designer encoded the bottom alignment action just mentioned and another which made sure that the page number's color was dependent on that of the chapter title's, and wanted to test this out. First, two new objects are created, a chapter title and a page number. The chapter title has its color changed and is moved to the bottom of the page. To re-apply the encoded knowledge, the designer selects both objects and then clicks on the new domino. The following is what happens internally.

First the action list attached to new icon is sent to the Generalize_Action_Data function and the first action in the list (the bottom alignment action) is extracted. Next the anchor object of that action is sent the message:

\[
\text{SendMessage(original\_anchor, action->id, action->values, values):}
\]

The system uses this first message to determine what the new anchor object is based on the original anchor object used in the demonstration and data supplied in action->values. Action->values in this case is a search function with the parameter of "a TEXT object." Since only two objects are selected (i.e., the chapter title and the page number), the search space is minuscule and the first object is determined to be the anchor based on the fact that it is a TEXT object and it was selected as the anchor object in this present context. This object is stored in the variable values. At present, Abatan is still quite a novice in that it selects the first object which matches the parameter case. A more robust criteria for selecting the object might be to have it match multiple parameter cases before it is chosen.

Next, an almost identical message is sent to the rest of the action argument list (which only contains one other member, the page number).

\[
\text{SendMessage(action->arguments, action->id, values):}
\]
In this case, `action->arguments` is the search function `search_for(PAGE_NUMBER)` which was produced when the action was generalized. This will search for the object in the current workspace whose attributes match the parameter "PAGE_NUMBER," (i.e., any object whose class is that of a page number). Since there is only one likely choice, the new page number is sent the "BOTTOM_ALIGN" message stored in `action->id` with the argument `values` as its parameter. The "BOTTOM_ALIGN" message, when sent, invokes the move method `put_below` which calculates the present position of the new anchor object stored in `values` and moves the page number to a position below it.

Now that the bottom alignment action is out of the way, the `Generalize_Action_Data` function loops forward to the color change action. For this action, the page number's color should equal the present color of the chapter title (which is the anchor object). To do so, the page number is sent the following message:

```
SendMessage(action->arguments, action->id, action->values);
```

In this case, `action->id` is the message "SET_COLOR," and `action->values` is the chapter title. When the "SET_COLOR" message is invoke with another graphical object supplied as the data (as opposed to an explicit color value), the page number's `update_color_method` is invoked. This method sends the object stored in `action->values` a "GET_COLOR" message which explicitly returns the current red, green, and blues values of the object. These values are incorporate to change the color attributes of the page number so that they explicitly equal the chapter title's.
The second method of testing is to output the knowledge and plug it into an existing design system for use as a grammar. Weitzman's Logic of Layout Design Support system [Weitzman 92] was selected as the targeted testing environment.

In order for the Logic of Layout to utilize the encoded knowledge it must first be translated into a format which is acceptable by that system. In this case it means that the data should be translated into *relational grammar* rules [Weitzman 93] which are written in the Common Lisp programming language [Steele 90]. Relational grammars, as defined in [Weitzman 93], are extensions of traditional string languages which “include user-supplied domain relations” between graphical objects provided as input. A relational grammar rule defines a list of primitives (graphical objects) as its preconditions and manipulates these objects to produce constrained, composite objects. The grammar rule used to combine the image object and the caption, described in the first chapter, is as follows.

```lisp
(defvar (make-figure the-grammar)
  (0 figure (self (image 0) 1
                  (caption 0) 2))
  (1 image)
  (2 caption (caption-of 2 1))
  :out
  (constrain-figure 1 2))
```

This rule produces the composite object “figure” by combining an image and a caption. The caption's width is equaled to that of the image's and its position is moved to a location such that the image is directly above it. These manipulation operations are accomplished by applying a *constrain* function to the arguments. The following is the constrain function of the previous rule.

```lisp
(defun constrain-figure (image caption)
  (constrain-spaced-above image caption)
  (constrain-eq-width image caption))
```
Abatan manufactures custom relational grammar rules and corresponding constraining functions by running the list of actions stored in a domino through the Output Mechanism. For each action performed in Abatan, a single grammar rule and constrain function are generated for the Logic of Layout. The Output Mechanism is essentially a series of look up functions which have lexical mappings of action types used in Abatan to corresponding ones in the Logic of Layout. By design, the Abatan system was built such that for every type of action that can be recorded or performed, there is a corresponding function already present in the Logic of Layout. For example, the action of positioning the rulebar under the text, from the “Title” example, is directly translated into the function call:

```
(constrain-spaced-below rulebar text)
```

The following illustrates the entire constrain function produced from this example.

```
(defun constrain-title (text rulebar)
  (self (foreground-color rulebar) (foreground-color text))
  (constrain-spaced-below rulebar text)
  (constrain-aligned-left text rulebar)
  (constrain-eq-width text rulebar))

(constrain-eq-height text rulebar))
```

The rule which invokes this function, which is also written by Abatan, is made up of two parts. The first (up until the “:out”) lists the arguments necessary before the rule can be applied. The second part of the rule is an explicit call to the constrain function, mentioned above, with the rule's preconditions used as arguments.

```
(defrule (make-title the-grammar)
  (0 title (self (text 0) 1)
   (rulebar 0) 2))

(1 text)
(2 rulebar-cat (rulebar-of 2 1))
:out
(constrain-title 1 2))
```
The Output Mechanism uses the series of look up tables to explicitly write the rules and the constrain functions to a file. Figure 4.15 illustrates one of the Output Mechanism's lexical action look up functions. Depending on the action's id number, the appropriate Logic of Layout function call is used to write the rule.

```
char *LOL_action_type_lookup(int action_id)
{
    switch(action_id){
    case ALIGNLEFT:  return("aligned-left");       break;
    case ALIGNRIGHT: return("aligned-right");      break;
    case ALIGNTOP:   return("aligned-top");        break;
    case ALIGNBOTTOM: return("aligned-bottom");    break;
    case ALIGNOF:    return("spaced-right-of");     break;
    case ALIGNLOF:   return("spaced-left-of");      break;
    case ALIGNTOP:   return("spaced-above");       break;
    case ALIGNBOP:   return("spaced-below");       break;
    case SETCOLOR:   return(NULL);                   break;
    case SETSIZE:    return(NULL);                   break;
    case EQUALWIDTH: return(NULL);                   break;
    case EQUALHEIGHT: return(NULL);                  break;
    default:         LOL_convert(action_id);          break;
    }
}
```

Figure 4.15. An Output Mechanism lexical action look up table.

In a few cases there is not a one to one matching between Abatan and the Logic of Layout. For example, rather than using a function or a method to set an object's color, the Logic of Layout makes the explicit assignment call shown above to set the values. For these cases, Abatan uses another function, LOL_convert, which breaks the function call down into fragments that are sent through other lookup functions to write the appropriate sequence of commands. For the color constraint in the "Title" example, where the rulebar's color equals the text’s, the LOL_convert function performs the following sequence to produce the finalized assignment call shown above.
First, the action is converted into separate opening and a closing statements (figure 4.16). Next, the system retrieves the anchor object and performs the following three calls to write the assignment. The first writes the opening statement, (setf, the second writes the argument's values, (foreground-color argument), and the third writes the closer, (foreground-color anchor)).

```c
char *LOL_action_name_lookup(ACTION *action, char *string, int which)
{
    if (which == OPENER)
        CopyString("(set", string);
    else
        switch(action->id){
            case SETCOLOR: CopyString("(foreground-color %s)", string, action->values);
                break;
            case SETSIZE: CopyString("(string-size %s)", string, action->values);
                break;
        }
}
```

Figure 4.16. The LOL_action_name_lookup function.

Functions such as LOL_action_name_lookup (figure 4.16) and LOL_convert are accessed from the main output function, OutputActionList. As soon as a series of actions are encoded, the system automatically sends a message to the new domino asking it for its action list. The OutputActionList function loops through the action list and sequentially writes each individual action to a file. Upon startup, the Logic of Layout reads in this file, evaluates its contents, and incorporates this knowledge into its grammar. As users of that system create new layouts, the grammar rules are used to automatically format the design, in turn, aiding them during the design process.
The *Abatan* system was developed at the Visible Language Workshop in the MIT Media Laboratory. It was designed concurrently on a Digital DECstation 5000/200™, Silicon Graphics Iris Indigo™, and IBM RS/6000™ workstation, using the *BadWindows 2.0* [Alavi 91] graphical windowing environment which runs on top of the X Window™ system (Version X11R4) running ULTRIX™ 4.3. Both the BadWindows 2.0 windowing environment and the prototype software were written in the C programming language (ANSI) [Kochan 83]. The entire amount of source code used to write *Abatan* was roughly 400kb.

The *Text Management Library* used to display, edit, and encapsulate the TEXT class of graphical objects and the font palette were written by B.C. Krishna at the Visible Language Workshop and developed on a Digital DECstation 5000/200 using the C++ programming language [Lippman 89].

The Logic of Layout program used to test the output, encoded knowledge was written by Louis Weitzman at the Visible Language Workshop and developed on an Apple Macintosh IIFX™ using the Common Lisp programming language [Steele 90] and the Common Lisp Object System [Keene 89].
Figure 5.1 illustrates an example of how the magazine *Scientific American* currently formats its table of contents section. In this chapter the designer is going to teach the system how to layout a new table of contents configuration. For this example there are four different graphical objects which make up a single table of content's "article listing" (i.e., one entry). These are an article-image, an article-name, article-author, and article-description.
What the designer is actually going to do is to re-order the graphical objects so that:

1. The article-image is placed directly to the right of the article-name.
2. The article-image is top aligned with the article-name.
3. The article-author is placed directly to the right of the article-image.
4. The article-author is top aligned to the article-image.
5. The article-author's width is equal to the article-name's.
6. The article-description is placed directly below the article-author.
7. The article-description is left aligned with the article-author.
8. The article-description's width is equal to the article-author's.
9. The article-author and the article-description are colored the same color as the article-name.

The figures illustrated below show the step by step sequence of the actions performed. Only those panels highlighted in dark gray are actually recorded.

The demonstration begins by creating the anchor object, which in this case is an article-name, placing it into the layout and resizing it to an appropriate width. The first two rules to be illustrated are that the article-image should be placed directly to the right of the article-name and that it should be top aligned with it. First the anchor object's right edge is selected. Then the article-image's left edge is selected and moved over so that it is directly to the right of the article-name, from which the action “Right-Of” is recorded by the system and a new action panel is created in the Graphical History window.
Internally, when the designer selected the anchor's right edge, that object sent a message to itself stating that this was its hot spot. This information was noted and stored in the object's *where_touched* attribute slot. Then the article-image's left edge was selected, from which that object noted its hot spot, and was moved over to a position just to the right of the article-name. Once the move action had been performed, that object sent the system a message stating that it had just been manipulated. The type of operation was determined by which hot spot on the object had been selected. Taking this into account, the system inferred the "Right-Of" relationship using the heuristics that the action performed was a spatial operation, one object's left edge and another's right edge were selected, and their present position after the move in absolute terms of pixels was close enough to assume that the relationship of "Right-Of" was true.

Now the anchor object's top edge and the image's top edge are selected, and the article-image is dragged downward to be roughly top aligned with the article-name. This action produces the same respective internal response as the previous move operation. The system used the fact the same edges on both objects had been selected, the type of operation was a move, and the resulting values were close enough to warrant recording the action. As mentioned earlier, the objects do not have to be exactly aligned since *Abatan* allows for tolerance so that the designer can quickly create the layout.
In a similar fashion the next relationships are illustrated, showing the system that the article-author should be positioned to the right of the article-image and the two objects should also be top aligned. Internally, both the “Right-Of” and top alignment actions are recorded identically to the ones just mentioned, with the exception that the article-image is selected as the anchor object. After performing these actions both are recorded.
The next relationship to be shown is between the article-author and the article-name, such that the author's width should equal that of the name's. To do so, the designer clicks on the lower right hand corner of the object and resizes its width to the desired size, during which time the article-name, which is the anchor object, is still selected. The system records this action a little differently than the previous move operations. After the designer resized the object, that object sent a message to itself to find out how (in what direction) it was resized. Since its height remained the same and its width shrunk, it told the system that a "width resize" operation had just been performed. The system then checked to see if that object was the anchor. Since it was not the system compared the object's new width to that of the anchor's and because they were equal it recorded the action.
Now the article-author becomes the anchor object, since the article-description's spatial and dimensional attributes are to be dependent on it; its vertical position should be placed directly below it, its width equal, and its left edge aligned to it. In this series of relationships, a corresponding sequence of actions used to top align the article-name and article-author are now applied to left align the author and the description respectively. In addition, a second sequence, used to position the description under the author, corresponding to that performed to place the author to the right of the image, is executed, as is also a final one to resize the description's width.
The last relationship to be stated is that all objects in the layout, with the exception of the article-image, should have their color changed to equal that of the article-name's. To do so, each object is selected, with the article-name being the first to state that it is the anchor. Then a color selection is made by choosing a new color from the color palette, directly changing the anchor's red, green, and blue slot values. Internally, once the color selection is made the anchor object updates its color and tells the system about this change. The system then sends out messages to all currently selected objects telling them to change their color based on the anchor's. Each object sends a "GET_COLOR" message to the anchor to receive its current values and calls its \texttt{update\_color\_method} to incorporate these changes. After doing so the system records the action of coloring all objects the same color, with the anchor object being the deciding factor.
To add a little flavor to this example, two new rules with some potential ambiguity are added. The designer will state that all pieces of text, with the exception of the article-name, should have their point size set to eighteen and that these objects should be fifty percent transparent. In terms of this example, this means that the article-author and the article-description should have their attributes altered.

First the two objects are selected, the order of which does not matter. Once selected, their point size value is altered by setting the point size slider, located in the font palette, to eighteen. This invokes the same sequence of events that recorded the color change, with the exception that the objects send themselves messages to update their point size slots rather than their color slots. Finally, the transparency selection of 128, which is the halfway value between 0 and 256 (the current transparency range) is made using the transparency slider in the color palette and the action is recorded.
Now that the primary demonstration is over, the designer must clarify the point size and transparency actions. For the point size change, the designer must illustrate to the system that this action should be viewed as “change both objects' point size equal to eighteen,” and not just to “make sure that both objects have the same point size value.” Since the interpretation in simply one of absolute versus relative recognition, the designer selects the “Interpret Absolutely” item from the “Equal Point Size” action panel's menu, from which the system notes this specification by setting the action's How slot value equal to “ABSOLUTE.” When this action is re-applied the system first checks this slot to see how the action should be performed. Since the slot's value equals “ABSOLUTE,” it will then retrieve and use the value eighteen, stored in the action->values slot as the parameter to the change point size method.

Disambiguating the transparency action is performed a little differently. If the designer selects the “Interpret Absolutely” item from the panel's menu, the system would remember this action as “change all objects' transparency values equal to 128.” While within Abatan this would produce the correct response, since 128 is fifty percent of the transparency range, in other programs this may cause problems, since their ranges may vary. The designer must explicitly state that this value is “fifty percent.” To do so, the “Change Transparency” action panel is sent to the Generalization Editor. Once loaded, the designer selects the “Alter Value” menu item from the Values menu and then types in the amount “50%.” This informs the system that the designer's intent was to view the value in terms of a relative amount to the overall range.

Now that the ambiguities are out of the way, the panels in the Graphical History window that are to be included in this description are selected. For this example, all of the actions that were performed during the demonstration need to be included, so they are all selected and in the order that they were created. As mentioned earlier, panels can be selected in different orders to produce different effects and it is not a requirement to select all of the panels.
Now that all of the necessary panels are selected in the Graphical History, the encode button in the Generalization Editor is clicked on and the system produces a new domino icon in its interface (figure 5.2). The left hand side of the domino lists the type of graphical objects which this bit of design knowledge operates on and the right hand side shows an example of the results obtained when the knowledge is applied to these arguments.

![Format Scientific American Table of Content Items](image)

Figure 5.2. *The Scientific American* “Table of Contents” domino.

It is now time to test the knowledge by re-applying it to new objects. First, new article objects are created and placed into the layout. The new article-name has its color changed to be white and its width increased. To add another twist, its typeface is also changed to be ZapfChancery. This is to show that the designer can still make other extraneous changes after the demonstration is over and since the system was not explicitly told to “make sure that all objects’ typeface are the same,” it will not incorporate this change. *Abatan* is not like a macro system in the respect that it does not restrict re-application to be a hard-coded event where all of an object’s attributes are pre-determined at the time of the demonstration and would therefore be changed to equal that of what had been recorded.
Once all the objects have been selected, the designer clicks on the new domino icon and the system automatically formats these new arguments. As can be seen in figure 5.4, the knowledge has properly positioned and resized all objects relatively to the new situation; the text objects have their color changed to equal the new anchor object's color, and only the new article-author and article-description have had their point size and transparency altered.
To further test the knowledge from within *Abatan*, the article-name's width and color are altered again and the knowledge re-applied to reformat the layout relative to the new changes (figure 5.5).

![Image](image_url)

**Figure 5.5.** Re-application of the Scientific American "Table of Contents" rules after changes have been made.

If this knowledge was to be used in a program such as Colby's LIGA, it would enable the system to automatically reformat the layout to accommodate for environmental changes, such as reducing the display's dimension. In figure 5.6, the layout area's width has been reduced to half of that in figure 5.5. To reformat the objects, the designer need only reduce the article-name's point size and then re-apply the knowledge. The system automatically does the rest.

![Image](image_url)

**Figure 5.6.** Further re-application of the Scientific American "Table of Contents" rules to adapt to a new situation.
The system further differs from a macro program in that the encoded knowledge can be applied to only some of the objects, not requiring that all of the objects be affected each time the knowledge is re-used. When re-application is taking place only those objects which are selected at the time are affected. The designer may only want to format the article-author and the article-description, but leave the name and image as they are. In addition, only those attributes which are directly affected, are altered. Below in figure 5.8 we see that only the author and description's point size, color, transparency, and dimensional attributes have been altered, but the author's horizontal position, and the name and image's attributes remain the same.

Figure 5.7. Partial re-application "before" state.

Figure 5.8. Partial re-application "after" state.
In addition to partial re-application, the designer is not restricted to using the encoded knowledge within one layout in the same manner each time. In figure 5.9 each article listing, before the knowledge was applied, had its author-name's typeface and point size set to different values. While the fine details of each are different, all listings exhibit the same style.

Figure 5.9. A newly formatted Scientific American table of contents.
Figure 5.10. Outputted rules from the Scientific American "Table of Contents" example.
It is now time to test the encoded rules using the second method of output. Figure 5.10 illustrates the code generated from this demonstration. This code was written to a file and then incorporated into the Logic of Layout program. Figure 5.11 displays another new table of contents generated by the Logic of Layout using the encoded knowledge.

![Scientific American Table of Contents](image)

*Figure 5.11. Automatic generation of a new table of contents based on the encoded Scientific American “Table of Content” rules.*
The prototype system, *Abatan*, has been implemented to explore the possibilities of teaching design rules to a computer in a similar fashion as to how an accomplished designer would teach a novice; through observation and example. In practice, several example layouts, in a variety of configurations, are presented to a student to study. After examining these the student is capable of creating new variations of layouts which exhibit a similar style. *Abatan* attempts to perform a similar feat by watching how example layouts are created and translating a designer's graphical actions into computer code that is generalized and can be reapplied to create new variations of layouts.

Current methods of encoding design knowledge via human analysis and verbal translation have been replaced with an interaction model that more closely resembles the human-teacher dialog. From the designer's standpoint, their expertise is transmitted simply by creating a layout. Present at all times is the underlying computer system which observes and learns. Using such a model the *Abatan* system has been able to successfully encode a wide range of design rules using relatively few examples. The goal application of the system, to enable the translation of visual knowledge into a usable symbolic form, has proved not only useful but highly functional and easy to use. Once the knowledge has been transferred onto the computer its uses are limitless. Design Support systems can utilize it as a grammar to aid users during the actual design process and Automatic Layout programs can extract from it the constraint relationships needed to effectively present visual information in constantly changing computer environments.
While the attempts made in this thesis have proved worthwhile, they are still far from being accepted as a standard. If we are to continue to support visually oriented users and learn from their years of experience, several future issues must be confronted and met if any substantial use of such a model is to come.

At present, the Abatan system does not learn or make correlation between multiple examples which share similar features and does not utilize the experiences found in negative examples. In addition, the system only deals with capturing simple design rules about spatial and typographical information. It does not attempt to address higher order rules such as balance, proportion, harmony, or positive/negative space, all of which are important factors and foundations of design.

While these higher order rules are usually far more difficult to describe, they are not impossible, and in fact some lend themselves to further exploration in a computational environment. For example, the higher order rule of "white space balance" is probably easier to describe to a computer than it would be to a human. Take for instance the layout of this page. The amount of white space is calculated by adding up the total geometric area of all elements (i.e., their bounding boxes) and subtracting this value from the overall area of the page. From this value an initial test (the rule’s pre-condition) can be performed to see if enough white space is present. If there is, secondary tests can be used to determine if the layout is balanced. These tests could be performed using a number of techniques; explicit lower order rules which define boundaries, applying the Golden Section to the layout [Bringhurst 92], calculations on the grid, or by applying the Fibonacci series. If the layout is not balanced, constrain functions, possibly based on the same techniques used to determine if the layout was balanced in the first place, could be applied to the objects to conform them in such a way that would balance the layout.
If a future more mature version of Abatan were to be built, we could easily see and should hope to expect several other adaptations to the system (including those stated above). Lower level changes would include providing the system with a means of filtering out redundant information on the fly so that it does not record too much or too little of the design process, in addition to enabling the designer to explicitly control what the system records [Turransky 93]. Allowing the designer to define new lexical types and state what their internal description and function should be through graphical annotation [Lieberman 93b] is another. Using automatic inferencing to determine relationships from a static view of a layout [Krishna 93], not requiring that the designer explicitly state each and every one is a third. A fourth might be to automatically loosen or tighten the grammar rules depending on how much the system has learned or who is teaching it (i.e., expert vs. novice) [Weitzman 93]. Lastly, incorporating a case library such as that used in [MacNeil 90] or [Colby 92] to base the learning curve on is also a future direction to be taken up.

Respectively, several higher level issues also pave the way towards future research. Higher level changes would include the seamless integration of a gesture based interface [Donoghue 93], where standard graphic design editing notation could be used. Simultaneous access to multiple grammars, allowing the designer to mix styles to produce hybrid layouts is another. Providing direct access and editing control to the grammars and re-integration of the knowledge learned back into the grammar is just as important. In addition, providing support during the actual design process, as is done in existing Design Support systems, is also crucial.
The abundance of layout problems commonly associated with the presentation of visual information on computers demands that computer systems be incorporated with graphic design knowledge to aid users in their work. Incorporating the knowledge needed by these systems using conventional methods of encoding has proved restrictive. This thesis has shown that it is possible to take the next step and enable the translation of visual knowledge into a usable symbolic form simply by using Programming by Demonstration techniques that more closely model how this process is performed among humans. While it is not a complete solution, it does begin to lay the foundation for investigating new techniques which may make it easier for graphic designers to communicate with a computer in a manner which supports their skills and in a way which the computer can benefit from their experiences.
Visual Examples of Design Knowledge

The following figures illustrate two different types of layout, each formatted with distinct rules. In the first set (Layout 1, Versions 1 - 5), the rules are defined as follows.

1. The title has all of its attributes (spatial, dimensional, and typographical) fixed. Its position should always be located at the coordinates (200, 700), its dimensions in pixels 225 X 75, its color black, transparency 0, typeface Helvetica_Italic, and point size 72.

2. The description should be bottom aligned to the layout area's bottom margin.

All other attributes of the objects are variable.

For the second layout (Layout 2, Versions 1 - 4), the rules are defined as:

1. A piece of text and a rulebar are left aligned.

2. A headline or an author is left aligned to the layout area's left margin.

3. A description is top aligned with a caption.

4. A description is placed to the right of a caption at a fixed offset.

5. A piece of text is spaced above a rulebar's bottom edge.

6. All objects, with the exception of an author, have the same color, typeface, and point size.
Computer systems which can be trained to learn the techniques designers use to present visual information, by having a designer demonstrate their application on a working example may provide a more natural means of translating this type of knowledge from its original visual form into the electronic environment, without the necessity to first translate it into a textual representation.
Computer systems which can be trained to learn the techniques designers use to effectively present visual information, by having a designer demonstrate their application on a working example may provide a more natural means of translating this type of knowledge from its original visual form into the electronic environment, without the necessity to first translate it into a textual representation.
Computer systems which can be trained to learn the techniques designers use to effectively present visual information, by having a designer demonstrate their application on a working example may provide a more natural means of translating this type of knowledge from its original visual form into the electronic environment, without the necessity to first translate it into a textual representation.
Computer systems which can be trained to learn the techniques designers use to effectively present visual information, by having a designer demonstrate their application on a working example may provide a more natural means of translating this type of knowledge from its original visual form into the electronic environment, without the necessity to first translate it into a textual representation.

Abatan
Computer systems which can be trained to learn the techniques designers use to effectively present visual information, by having a designer demonstrate their application on a working example may provide a more natural means of translating this type of knowledge from its original visual form into the electronic environment, without the necessity to first translate it into a textual representation.
1.1 The direct translation of visual rules into symbolic form.
1.2 Applied design principles.
1.3 Weitzman's Design Support system, the Logic of Layout.
1.4 Techniques used to emphasize a "Title."
1.5 A "Title" object formatted with simple design rules.
1.6 Hierarchical structure of a layout.
1.7 Anchor objects of a layout.
1.8 Re-application of "Title" rules to new arguments.

2.1 Weitzman's Designer.
2.2 Colby's Liga.
2.3 MacNeil's Tyro.
2.4 Maulsby's Metamouse.
2.5 Lieberman's Mondrian.
2.6 Kurlander's Chimera.

3.1 The Abatan Design Environment.
3.2 Schematic diagram of the Abatan system.
3.3 The color and font palettes.
3.4 The Lexicon palette.
3.5 The Grammar palette.
3.6 The Graphical History.
3.7 The Editor.
3.8 A table of contents formatted with seven design rules.
3.9 The "Bottom," "Left," and "Right" aligning actions.
3.10 The "Equal Color" and "Equal Font" actions.
3.11 Graphically altering elements to reflect system interpretation.
3.12 The "Table of Contents" domino icon.
3.13 Re-applying the "Table of Contents" rules to new arguments.
# Contents

<table>
<thead>
<tr>
<th>Chapter</th>
<th>Title</th>
</tr>
</thead>
<tbody>
<tr>
<td>one</td>
<td>Introduction</td>
</tr>
<tr>
<td>two</td>
<td>Related Research</td>
</tr>
<tr>
<td>three</td>
<td>Overview of the Project</td>
</tr>
<tr>
<td>four</td>
<td>How it Works</td>
</tr>
<tr>
<td>five</td>
<td>Scenario</td>
</tr>
<tr>
<td>six</td>
<td>Conclusion</td>
</tr>
<tr>
<td>one</td>
<td>Visual Examples of Design Knowledge</td>
</tr>
<tr>
<td>two</td>
<td>Symbolic Examples of Design Knowledge</td>
</tr>
<tr>
<td></td>
<td>Bibliography</td>
</tr>
</tbody>
</table>

**Capturing Graphic Design Knowledge from Interactive User Demonstrations**

*Alan Turkansky*
### Supervisor

<table>
<thead>
<tr>
<th>Name</th>
<th>Title</th>
<th>Affiliation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ronald MacNeil</td>
<td>M.F.A.</td>
<td>Principle Research Associate Massachusetts Institute of Technology</td>
</tr>
</tbody>
</table>

### Reader

<table>
<thead>
<tr>
<th>Name</th>
<th>Title</th>
<th>Affiliation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Russell Kirsch</td>
<td>S.M.</td>
<td>Director of Research Sturvil Corporation</td>
</tr>
<tr>
<td>William Porter</td>
<td>Ph.D.</td>
<td>Professor of Architecture Massachusetts Institute of Technology</td>
</tr>
</tbody>
</table>
Submitted to the Program in Media Arts and Sciences, School of Architecture and Planning on July 15, 1993 in partial fulfillment of the requirements for the degree of Master of Science in Media Arts and Sciences at the Massachusetts Institute of Technology. This thesis describes a system which uses a machine learning technique called Programming by Demonstration to overcome this translation problem and enable the transformation of visual ideas into usable symbolic forms. It offers a working model, called the Abatan system, for capturing re-usable, graphic design knowledge from interactive user demonstrations.
Symbolic Examples of Design Knowledge

The following figures illustrate the computer code generated for the two groups of layouts in appendix 1. For both sets, a rule and a constrain function are given. Note, that the definition of the constrain functions are open ended (not all attributes are specified) so they can allow plenty of leeway for producing a robust variety of layouts.

Layout 1 Rule

(defrule (make-layout-1 the-grammar)
  (0 entry-cat (self (layout-area 0) 1
                (title 0) 2
                (description 0) 3))
  (1 layout-area-cat)
  (2 title-cat
    (title-of 2 1))
  (3 description-cat
    (description-of 3 1))
  :out
  (constrain-layout-1 1 2 3))

Layout 1 Constrain Function

(defun constrain-layout-1 (layout-area title description)
  (setf layout-area (lookup-realization layout-area *toc-store*))
  (setf title (lookup-realization title *toc-store*))
  (setf description (lookup-realization description *toc-store*))
  (set-tagged-value! title :width 225)
  (set-tagged-value! title :height 75)
  (constrain-align-bottom description layout-area offset))
  (let ((title-string (first (displayed-strings title)))
    (setf (background-color title) *black*)
    (setf (string-font title-string) "Helvetica_Italic")
    (setf (string-size title-string) 72)
    (setf (string-offset title-string) 200)
    (setf (string-offset title-string) 700)))

Capturing Graphic Design Knowledge from Interactive User Demonstrations 112
Capturing Graphic Design Knowledge from Interactive User Demonstrations
Bibliography


