GENERALIZED GRAPHICAL OBJECT EDITING

A DISSERTATION
SUBMITTED TO THE DEPARTMENT OF ELECTRICAL ENGINEERING
AND THE COMMITTEE ON GRADUATE STUDIES
OF STANFORD UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

By
John M. Vlissides
June 1990

© Copyright by John M. Vlissides 1997
All Rights Reserved

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a dissertation for the degree of Doctor of Philosophy.

Mark A. Linton
(Principal Advisor)

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a dissertation for the degree of Doctor of Philosophy.

David M. Ungar

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a dissertation for the degree of Doctor of Philosophy.

David M. Bloom

Approved for the University Committee on Graduate Studies:

Dean of Graduate Studies & Research

Abstract

Many editors use the graphics capabilities of personal workstations to provide a visual editing environment. Such editors present graphical representations of familiar objects and allow the user to manipulate the representations directly. This style of interaction is usually more intuitive to the user than typing statements in a command language. However, implementing a graphical object editor has been a difficult undertaking. Though many packages exist that aid in the construction of graphical user interfaces, none are designed specifically for building graphical object editors. This is significant because there is a substantial semantic gap between general user interfaces and the functionality of graphical object editors. For example, user interface packages usually provide buttons, scroll bars, and ways to assemble them, but they do not offer primitives for building drawing editors that produce PostScript or schematic capture systems that produce netlists. Higher-level abstractions are needed to make such applications easier to develop.

Unidraw is a framework for creating object-oriented graphical editors in domains such as technical and artistic drawing, music composition, and circuit design. The Unidraw architecture simplifies the construction of these editors by providing programming abstractions that are common across domains. Unidraw defines four basic abstractions: **components** encapsulate the appearance and semantics of objects in a domain, **tools** support direct manipulation of components, **commands** define operations on components and other objects, and **external representations** define the mapping between components and the file format generated by the editor. Unidraw also supports multiple views, graphical connectivity and confinement, and dataflow between components. This thesis describes the Unidraw design, implementation issues, and three experimental domain-specific editors we have developed with Unidraw: a drawing editor, a user interface builder, and a schematic capture system.

Our results indicate a substantial reduction in implementation time and effort compared with existing tools.

Acknowledgments

I am deeply indebted to my advisor, Professor Mark A. Linton, for his guidance, encouragement, and friendship throughout the course of my doctoral program at Stanford. His patience, insight, and technical talents have been integral to the success of this work and to my education as a researcher, and I am grateful. I also thank Professors David M. Ungar and David M. Bloom for serving diligently on my reading committee and Professor Mark A. Horowitz for serving on my oral examination committee.

I would also like to thank the many other people who have had an impact on this work from a technical standpoint. Foremost among these is Paul Calder, InterViews co-laborer and hacker *extraordinaire*, with whom I have had countless discussions, arguments, and melees concerning both relevant and irrelevant topics. Many ideas in this work are the direct result of interactions with Paul, and I'm sure things would have turned out much differently without his help. I also extend thanks to those brave souls who have taken the time to experiment with early versions of the Unidraw system and have provided helpful feedback. I especially thank Dr. Joseph Sacco and Craig Zarmer for their constructive comments. Thanks also to colleagues Russell Kao and Larry Soule for tutoring me in the ways of schematic capture.

Then there are those from whom I have benefited in other ways. I have had the pleasure of working alongside the particularly entertaining and stimulating people that make up the Allegro group, including Craig Dunwoody, John Interrante, Doug Pan, and Dr. Russell Quong (member emeritus). Arturo Salz and Rich Simoni deserve special mention as close friends and frequent dinner companions. I have also received invaluable assistance at every stage from very capable secretarial staff, including Lillian Betters, Cathy Dicker, Linda Kovach, Carmen Miraflor, Louise Peterson, Margaret Rowland, and Tanya Walker. Special

thanks are due Professor John G. Linvill for his visionary efforts to make Stanford's Center for Integrated Systems not just a reality but a truly splendid environment in which to nurture future researchers.

Finally, I am forever thankful to and for my parents, Matthew and Sophia, for their love, encouragement, support, and prayers. I dedicate this work to them.

This research has been supported by the NASA CASIS project under Contract NAGW 419, by the Quantum project through a gift from Digital Equipment Corporation, and by grants from the Charles Lee Powell foundation and Fujitsu America, Inc.

Contents

Abstract **iv**

Acknowledgments **vi**

1 Introduction **1**

1.1 Terminology . . . 3
1.2 Principles . . . 4
1.3 Goals . . . 7
1.4 Results and Contributions . . . 7

2 Related Work **9**

2.1 Domain-Specific Editors . . . 9
2.1.1 Drawing Editors . . . 9
2.1.2 CAD Tools . . . 10
2.1.3 Diagram Editors . . . 10
2.1.4 User Interface Builders . . . 11
2.1.5 Other Domains . . . 12
2.2 Multi-Domain Editors . . . 12
2.2.1 Sketchpad . . . 13
2.2.2 ThingLab . . . 13
2.2.3 Alternate Reality Kit . . . 14
2.2.4 Impulse-86 and GROW . . . 14
2.2.5 ThinkerToy . . . 15
2.2.6 LabVIEW . . . 16

2.3 Graphical Programming Environments 17
2.3.1 Visual Programming 18
2.3.2 Programming-by-Example 19
2.4 Unsolved Problems 19
2.5 Summary 21

3 Unidraw Architecture **22**
3.1 Overview 22
3.1.1 Basic Abstractions 25
3.1.2 Subjects and Views 26
3.1.3 Structure of a Unidraw-Based Application 27
3.2 Components 28
3.2.1 Subject and View Protocols 29
3.2.2 Graphical Components 31
3.3 Component Semantics 34
3.3.1 Connectors 34
3.3.2 Domain-Specific Connectivity 39
3.3.3 State Variables 41
3.3.4 Transfer Functions 42
3.3.5 Dataflow 42
3.4 Commands 44
3.5 Tools 45
3.5.1 Tool Protocol 46
3.5.2 Manipulator Protocol 47
3.5.3 Object Communication during Direct Manipulation 48
3.5.4 Tools versus Manipulators 51
3.6 External Representations 52
3.7 Application Framework 53
3.7.1 Viewer 53
3.7.2 Editor 54
3.7.3 Selection 55

3.7.4 Catalog 56
3.7.5 Unidraw 57
3.8 Summary 60

4 A Prototype Implementation 62
4.1 C++ 63
4.2 InterViews 63
4.3 Structured Graphics 64
4.3.1 Class Organization 66
4.3.2 Graphics State Concatenation 68
4.3.3 Incremental Update 70
4.4 Overview of the Unidraw Prototype 71
4.5 Components 73
4.5.1 Composition 74
4.5.2 View Consistency 76
4.6 Connectivity 81
4.6.1 Modeling Connectivity Semantics 82
4.6.2 Primitive Combinations and Equivalents 83
4.6.3 Recursive Substitution 85
4.6.4 Connector Translation 87
4.6.5 Algorithmic Complexity 88
4.7 Dataflow 90
4.8 Commands and Tools 93
4.9 Catalog Semantics 95
4.10 Summary 96

5 Experimental Applications 98
5.1 Drawing Editor 99
5.1.1 User Interface 99
5.1.2 Implementation 102
5.1.3 Experience 104
5.2 User Interface Builder 108

5.2.1 User Interface 109
5.2.2 Implementation 114
5.2.3 Experience 124
5.3 Schematic Capture System 125
5.3.1 User Interface 127
5.3.2 Implementation 132
5.3.3 Experience 137
5.4 Summary 139

6 Conclusion **145**
6.1 Summary of Work and Contributions 145
6.2 Observations 147
6.3 Future Work 147
6.3.1 Stricter Adherence to the Architectural Specification 148
6.3.2 Library Optimizations 148
6.3.3 Architectural Extensions 148
6.3.4 Additional Experiments 149
6.3.5 Graphical Object Editor Builders 149

Bibliography **151**

List of Tables

1 Component subject protocol . . . 30
2 Additional operations defined for graphical component subjects . . . 32
3 Additional operations defined for graphical component views . . . 32
4 Connector subject protocol . . . 38
5 Command protocol . . . 44
6 Tool protocol . . . 47
7 Manipulator protocol . . . 47
8 External view protocol . . . 52
9 Viewer protocol . . . 53
10 Editor object protocol . . . 55
11 Selection protocol . . . 56
12 Catalog protocol . . . 57
13 Unidraw object protocol . . . 58
14 Graphic library class hierarchy . . . 66
15 Unidraw prototype library code breakdown . . . 71
16 Predefined component subclasses . . . 72
17 Additional predefined subclasses . . . 72
18 Connector glue parameters for modeling standard connectivity semantics 82
19 Primitive connections . . . 84
20 Parameter definitions for equivalent connector glue in terms of primitive combination parameters . . . 84
21 Actual sizes for primitive combination connector glue in terms of equivalent glue actual sizes . . . 86

22 Drawing code breakdown 103
23 Run-time space requirements for idraw and Drawing 105
24 Comparison of idraw and Drawing run-time performance 106
25 UI class hierarchy 115
26 UI code breakdown 116
27 Schem class hierarchy 133
28 Schem code breakdown 134

List of Figures

1 Layers of software underlying a domain-specific editor 23
2 General structure a domain-specific editor based on Unidraw 29
3 Various highlighting styles: (a) unhighlighted, (b) handles, (c) color reversal, (d) substitution 33
4 Several connections and their semantics 36
5 Connector glue characteristics 37
6 Three pins connected in series 37
7 Possible composition of an inverter component 40
8 Communication between objects during direct manipulation 49
9 Interface to damage class 70
10 View structure update algorithm 76
11 Initialization and view rearrangement 78
12 Damage incursion 79
13 Recursive solution of connection network 85
14 Response to connection perturbation 87
15 Dataflow algorithm 91
16 Dataflow schematic 92
17 Drawing editor 100
18 User interface builder 109
19 Pop-up from a click on a glue component with the Examine tool 111
20 Dialog box containing glue component information 112
21 Examining a composition hierarchy 113
22 Part of the composition hierarchy presented in a separate view 114

23 Connector construction for tray children 118
24 Connections for a two-dimensional tray alignment 119
25 Sample UI external representation 121
26 Schematic capture system 126
27 Using the Examine tool on an inverter element 129
28 Specifying a NAND gate's appearance 134
29 Using nodes to define connectivity, logic simulation, and netlist semantics 140
30 Specifying element and node names and transmission methods 141
31 Defining the truth table for the NAND operation 142
32 Creating a graphical component tool for instantiating new NAND gates . 143
33 Creating NAND gates with the newly defined tool 144

Chapter 1

Introduction

Interactive graphics has become a standard part of application software for personal computers and workstations. A growing number of programs provide a visual metaphor in which the user deals with graphical representations of familiar objects. These representations are manipulated in ways that are more intuitive to the user than command languages. Well-designed graphical interfaces allow the user to perform complex operations with ease. The user can then concentrate on the design problem and not the computer—it becomes transparent to the task. Thus, more productive use is made of user and computer time alike.

While programs that use interactive graphics offer significant benefits, writing such programs is difficult. Even with modern programming environments, debugging facilities, user interface toolkits and other aids, developing a new drawing editor or printed circuit board layout system, for example, requires considerable effort both in design and implementation. This discourages development of new applications that take advantage of interactive graphics.

Whenever the complexity of a programming task becomes an obstacle, some mechanism must be introduced to augment the programmer's abilities. Historically, the most effective way to manage software complexity has been to provide abstractions that shield the programmer from extraneous detail. One way to provide effective abstractions is to distill common functionality from some class of programs into a set of primitives that can be used in each program. First we identify the class of applications we want to support. Then we characterize their similarities and determine the abstraction level that maximizes both

flexibility and expressiveness, two often conflicting goals. Once the abstractions are clear, we define a software architecture to specify and organize them. Finally, the architecture is implemented in a set of primitives that closely reflect the abstractions. Accompanying these primitives are mechanisms for assembling them into complete applications.

This thesis presents Unidraw, a system that supports the construction of a wide variety of interactive graphics editors. Unidraw comprises abstractions that are targeted specifically towards direct-manipulation graphical editors such as drawing, music, and schematic editors. Unidraw reduces the effort required to build these programs by providing a set of primitive objects and operations that programmers use to construct new kinds of graphical editors. An editor is created by using, modifying, extending, and combining elements from Unidraw's repertoire of predefined objects and operations.

Unidraw introduces a higher level of abstraction than existing systems by distinguishing and catering to a particular class of applications. Unidraw comprises abstractions that are characteristic of graphical editors: it does not offer toolkit functionality (it is used in conjunction with a toolkit), nor does it assume the role of a program development environment (it defines objects that are used in an existing environment). We have avoided replicating existing functionality in Unidraw; instead we have focused on providing new and previously unsupported capabilities.

The remainder of this chapter defines terminology that is used throughout the thesis, discusses the principles that underlie Unidraw's design and implementation, and describes the goals, results, and contributions of this research. In Chapter 2 we discuss work by others that relates to graphical editing systems. Chapter 3 introduces the Unidraw architecture, providing an overview of the system's design and basic concepts. Chapter 4 describes a prototype implementation of the architecture; it presents notable algorithms and the mechanisms that underlie the programming interface. Chapter 5 covers three experimental graphical editors we built with Unidraw to evaluate its effectiveness. We describe their functionality, user interfaces, and implementations, and we evaluate their performance relative to existing systems. Finally, Chapter 6 summarizes the approach to building graphical editors, discusses how successful Unidraw is in achieving the goals of this research, and suggests directions for future work.

1.1 Terminology

In the thesis, a **domain** refers to a physical or abstract context in which objects are manipulated. For example, technical drawing is a domain in which graphical elements are arranged to form drawings. Circuit layout is a domain in which electronic components are arranged to form circuits. Programming is a domain in which tokens are arranged to form programs.

An **editor** is a program that supports the manipulation of objects in a domain. An editor usually presents some abstraction of objects in the domain rather than the objects themselves. A **graphical object editor** uses interactive graphics to let the user manipulate pictures of objects from one or more domains. It also maps objects from their pictorial representation to at least one other representation.

Domain-specific graphical object editors support a single domain only. Graphical object editors have been built for several domains. Examples of such editors include

- drawing editors that let the user manipulate geometric objects and produce PostScript or other representations,
- diagram editors for creating pictures of finite state automata (FSAs) and generating state or excitation tables,
- schematic editors in which the user manipulates schematic circuit symbols to produce a netlist, and
- music editors that support graphical music composition and generate MIDI code.

Multi-domain editors can be used to manipulate objects in several domains. Current systems that can be classified as multi-domain graphical object editors are typically drawing editors with constraint specification capabilities. The user can supplement the graphical representations with semantics that define relationships between objects. These semantics depend on the domain, so several domains can be supported by specifying the appropriate constraints. Other systems that can support multi-domain editing are programmable domain-specific editors and graphical programming environments, which allow the user to write programs by manipulating graphical objects directly.

1.2 Principles

Once complexity reaches an unmanageable level in a software system, a new level of abstraction must be introduced to allow further increases in functionality. This fact has been demonstrated throughout the history of software development. In the context of interactive graphics, the progression to higher levels of abstraction has proceeded as follows:

- Device-dependent graphics libraries
- Device-independent graphics libraries
- Window systems
- User interface toolkits
- User interface management systems

Early interactive graphics applications accessed the graphics hardware directly. Later, libraries of routines were developed both to avoid replicating common graphics programming idioms in every application and to abstract the lowest-level details of generating graphical output. The next step was the introduction of device-independent graphics packages [5, 10, 18, 21, 44] that allowed applications to be developed independent of the graphics hardware. This eliminated the need to retarget applications to new graphics architectures. To make such portability possible, these packages define a layer of abstraction called an **imaging model** upon which to build applications.

The introduction of engineering workstations with large bitmapped displays made it feasible to run multiple graphical applications at once. To allow applications to share screen real estate, the **desktop metaphor** was developed in which applications generate graphics in one or more **windows**. Window systems [39, 41, 51, 52] were introduced to simplify the implementation of this interface; they support the window abstraction, control access to the screen, direct user input to the proper application, and manage shared resources in general.

As user interfaces became more sophisticated, the effort required to implement them increased dramatically. User interface toolkits [2, 22, 24, 33] once again raised the level

of abstraction by providing common user interface elements such as scroll bars, menus, buttons, and a framework for combining them into complete applications. User interface management systems (UIMSs) [35] offer an even higher level of abstraction than toolkits. UIMSs let an interface designer specify the semantics of interaction with a minimum of conventional programming, often at the cost of flexibility in the style of interface.

This history suggests a three step process in the development of a new level of abstraction:

1. Enough experience is gained with a class of applications so that their implementation is well understood.

2. The class of applications is examined as a whole to ascertain the fundamental elements they have in common.

3. A set of abstractions is developed that embodies the common elements and allows them to be used in concert.

This process is readily apparent in the progression of abstraction in interactive graphics. Device-dependent graphics libraries exploit the similarities between applications on specific graphics hardware; device-independent packages raise this level of abstraction just enough to accommodate the common attributes of most graphics hardware. Window systems use a unifying concept, the window, to let graphical applications coexist on the same display. When programmers tired of reimplementing the same user interface elements in each application, they developed toolkits. UIMS researchers recognized that the communication between user and application can be modeled as a dialog of inputs and outputs, and they sought to support this abstraction directly in their systems.

To make graphical object editors easier to build, therefore, we applied this three step process to create a new level of abstraction. Domain-specific graphical object editors are commonplace, and their implementation is understood well enough that a variety of systems are mass-marketed. The term "graphical object editor" is important because it groups a variety of programs under a common classification. This makes it possible to limit our search for commonality to a well-defined subset of existing applications.

There are other principles to consider. The abstractions that generalize the functionality of graphical object editors must have efficient implementations. Generalization often breeds

inefficiency, because a generalized system represents a compromise between constraints that do not exist for a specialized system. Specialization means the implementor can deal with a problem at a finer granularity. He can take advantage of special cases that do not apply in general. He can ignore considerations that may be critical to other specializations. To be a viable alternative to hand-crafted editors, a general system must yield editors with comparable performance.

Moreover, graphical editors seldom live in a vacuum; they generate output (in addition to that of their graphical interface) that is read by humans or consumed by other systems. For instance, drawing editors generate at least one representation of user drawings in a page description language such as PostScript. Music editors may generate both scores in PostScript and MIDI code for driving a synthesizer. Without provisions for such output, an editing system is of limited use in a larger environment.

Finally, there must be a balance between generality and functionality. It is unlikely that a very general system will offer abstractions that closely match the semantics of a particular editor. On the other hand, a highly functional system necessarily narrows its focus to support specific abstractions well, while other abstractions are unsupported or even discouraged. The tradeoff between generality and functionality depends on the level of abstraction, and achieving a good balance is difficult.

1.3 Goals

This research has focused on creating a software system with three key attributes:

1. It supports graphical object editing in a broad range of domains.
2. It significantly reduces the time it takes to develop a domain-specific editor compared to implementation from scratch.
3. It can be used to create stand-alone editors with performance and utility comparable to their from-scratch counterparts.

We designed and built Unidraw to fulfill these goals. To determine how well they were achieved, we used Unidraw to write three experimental editors: one for drawings, one for

user interfaces, and one for schematics. Later in the thesis we summarize this implementation experience and discuss the benefits of using Unidraw to build graphical object editors. We also discuss the impact of the major design decisions on the final system and suggest areas for improvement.

1.4 Results and Contributions

This thesis contributes a new level of abstraction for building graphical object editors. By identifying and characterizing this class of applications, we were able to design and build a general software system for supporting their development. The Unidraw architecture simplifies the design of graphical object editors, while our prototype implementation of that architecture simplifies their realization.

This work has demonstrated the viability of the Unidraw architecture and its implementation by using it to build editors for three different domains. Though these editors do not represent polished systems, they have proven themselves useful tools for their intended purposes. These editors were created in a fraction of the time needed to implement them from scratch, and their performance is more than adequate for production use—only a slight performance penalty is introduced by the Unidraw architecture.

Unidraw is a new tool for creating specialized graphical object editors. Since producing an editor for a domain is significantly easier than before, it becomes economical to develop editors for domains that have never benefited from interactive graphics. Programmers can create Unidraw-based graphical front-ends to a variety of programs that are currently batch-oriented. Users can then model and manipulate graphical representations of their data and let the editor produce the corresponding external representations required by the batch program. The key concept is that with Unidraw, programmers with little or no experience in the development of graphical object editors can build useful systems with relatively little effort, and users can enjoy the benefits of interactive graphics in a wider range of applications. In the past, systems that have put such flexibility into the hands of people have been used in ways that were never foreseen by their creators.

Chapter 2

Related Work

Work in the area of graphical object editing can be divided into three types of systems: domain-specific editors, multi-domain editors, and graphical programming environments. In this chapter we describe systems that characterize these three types of editors. We conclude by discussing the shortcomings of existing systems in achieving the goals of this research.

2.1 Domain-Specific Editors

Domain-specific editors are the most common graphical object editors. Most commercial graphical editors are targeted and optimized for a particular domain. Of the three types of editors discussed in this chapter, the design and implementation of domain-specific editors is understood best.

2.1.1 Drawing Editors

The prototypical graphical object editor is the drawing editor, many of which have been implemented and are in wide use. MacDraw [1] is probably the best-known example. MacDraw produces PostScript and QuickDraw representations of drawings and provides tools for creating geometric shapes and textual elements. Shapes can have different line styles and fill patterns, and text can appear in a variety of typefaces, fonts, and sizes.

MacDraw also provides simple alignment capabilities for positioning objects relative to one another. Other drawing editors provide more sophisticated features. For example, Juno [30] uses a constraint solver for enforcing relationships between objects, while Gargoyle [4] has mechanisms for specifying complicated graphical constructions precisely.

2.1.2 CAD Tools

Computer-aided design (CAD) tools that provide a direct manipulation metaphor for producing design specifications qualify as domain-specific graphical object editors. Magic [32] is a CAD tool for editing hierarchical VLSI layouts. It provides tools for manipulating representations of circuits directly. Leaf components are represented with geometric objects, typically rectangles. A cell is the basic unit of hierarchy that can contain leaf components and subcells. Magic generates CIF (Caltech Intermediate Form) files that can be processed by other design aids, as well as files containing Magic's internal representation of the circuit and subcircuit extraction information. Schematic capture systems such as Valid's SCALDsystem [54] and Cadence's EDGE [8] support interactive specification of hierarchical schematic diagrams and generate corresponding netlist representations. The user can assign attributes to schematic elements to add information to the netlist. Solid modeling systems such as Worldview [20] let users build three-dimensional volumetric models of solid objects such as spheres and parallelepipeds to produce a database of topological and geometric information. Once accumulated, the user can peruse the "world" of objects using two-dimensional interfaces to specify arbitrary points of view. Other CAD tools that qualify as domain-specific editors include mechanical design tools and project management systems.

2.1.3 Diagram Editors

Graphical abstractions are often used to specify, model, and document physical or mathematical processes. Finite-state diagrams and Petri nets [34] are examples of such abstractions for which graphical editors have been built. Finite state diagrams are used to model the behavior of finite-state automata. Jacob [19] describes a system that supports graphical specification and execution of finite state automata. The system is actually a visual version

of a state diagram language. It lets the user create a state diagram with a drawing editor-like interface and produces a corresponding textual representation of the diagram in the state diagram language. The user labels each state with an identifier and associates each transition with an action procedure, which is implemented in a conventional programming language. The system can then execute the state diagram, highlighting each state as it is encountered. The user can make the automaton enter any state immediately by clicking on its graphical representation.

Petri nets are diagrams that model asynchronous concurrent processes. They are often used to describe and analyze the parallel and nondeterministic aspects of computer hardware and software systems. SPAN [25] is a graphical editor developed at Carnegie-Mellon University for the construction and analysis of stochastic Petri nets. The SPAN interface provides iconic representations of Petri net elements (places, arcs, and transitions) that the user assembles to form complete networks. The user can peruse the network with scrolling and zooming operations. The system also provides analysis capabilities, including commands for generating the network's reachability set, for finding and solving the Markov matrix, and for animating the propagation of tokens through the network.

2.1.4 User Interface Builders

Domain-specific editors have also been developed for building user interfaces. Examples of experimental user interface builders are Carnegie-Mellon's GLO [31], Myer's Peridot [27], and Cardelli's dialog editor [9]. SmethersBarnes markets Prototyper [45], perhaps the first commercial user interface builder, while NeXT's Interface Builder [59] popularized the concept. These systems let the user arrange user interface elements into a prototype of the final interface and generate a textual specification for the interface. The textual specification might be conventional source code or code in a specialized user interface language; source code is compiled and linked into the application, while user interface language code is usually interpreted by a run-time library in the application.

2.1.5 Other Domains

The number and variety of domain-specific editors continues to increase as researchers and developers find new ways to use bitmapped displays and pointing devices. The systems mentioned do not represent an exhaustive list; other domains for which graphical object editors have been developed include music composition, calculator simulation, even pinball game construction.

2.2 Multi-Domain Editors

Characterizing an editor as either domain-specific or multi-domain is not always easy. Many of the editors described in Section 2.1 as being domain-specific can in fact be used in other domains. For example, a user of the Petri net editor who recognizes that Petri nets are a superset of finite state diagrams[1] can use the Petri net editor to edit finite state diagrams. A schematic capture system that can produce a PostScript representation of the schematic can be used as a special-purpose drawing editor.

In this discussion, we distinguish between domain-specific and multi-domain editors based on the following criteria:

1. *What is the editor's intended purpose?* If the editor is designed to be used for a particular problem, such as creating musical scores or editing VLSI layouts, then this suggests a specific domain.

2. *How easy is it to program or otherwise modify the editor's semantics?* If the editor cannot be extended conveniently, either by programming or other means, then it is doubtful that it can be used in a variety of other contexts.[2]

3. *How is the editor generally used?* If users do not apply the editor to problems in a significant range of domains, then probably it is not an effective multi-domain editor.

---

[1] A state machine can be represented by a Petri net that has been restricted so that each transition has exactly one input and one output.

[2] There is a limit, however, to how extensible such an editor can become before it is no longer a multi-domain editor but a general graphical programming system. This issue is addressed in Section 2.2.6.

We use these criteria to justify a particular system's classification if there is any doubt about which one is appropriate.

2.2.1 Sketchpad

Perhaps the most famous multi-domain editor is also one of the earliest graphical object editors. Sketchpad [53], created by Ivan Sutherland at MIT in the early '60s, was a pioneering computer drawing system with support for constraint specification and solution. A light pen was used to draw and manipulate images of objects on a vector display. Constraints were specified in a **generic block** that recorded how many variables were constrained, which of these variables could be changed to satisfy the constraint, and how many degrees of freedom were removed from the constrained variables. Constraints were implemented by adding new subroutines to the system. A number of constraints were predefined. Sutherland demonstrates in his thesis how the program's drawing and constraint capabilities were used to model mechanical linkages, trusses under load, and electrical circuits. He also shows how he created drawings in which dimension labels were constrained to reflect the proper values as objects were resized.

2.2.2 ThingLab

Borning's ThingLab [7, 6] owes much to the ideas developed in Sketchpad. ThingLab is a constraint-based drawing system that uses more contemporary technology to provide what Sketchpad did and more. ThingLab is implemented in Smalltalk [13, 14] and takes advantage of the object-oriented model to provide part-whole and inheritance hierarchies (also known as instance and class hierarchies) for describing the structure of a graphical simulation. ThingLab uses constraints to describe relations between the parts of the simulation, and it compiles constraints into Smalltalk code immediately after they are specified. The Smalltalk system then incorporates the code into the running program and executes it to satisfy the constraints. Thus, new constraints can be specified and implemented more easily than in Sketchpad. Moreover, Sketchpad required a large dedicated computer; ThingLab runs on a variety of personal workstations, and it uses bitmapped graphics with familiar user interface elements such as pop-up menus, windows, and scroll bars.

2.2.3 Alternate Reality Kit

The Alternate Reality Kit (ARK) [47] is a simulation environment with a strong emphasis on direct manipulation. ARK's design centers on a real-world metaphor in which graphical objects exhibit physical properties such as mass, velocity, and acceleration. A user manipulates these objects in artificial worlds called **alternate realities**. Objects called **interactors** govern the behavior of other ARK objects in an alternate reality just as the rules of nature govern real objects. For example, ARK includes a gravity interactor that defines a gravitational constant for an alternate reality. The gravity interactor provides an interface for changing the gravitational constant, thereby affecting the dynamic behavior of objects in the alternate reality. The ARK metaphor thus seeks to model reality as faithfully as possible to maximize ease of learning and use, but it balances this goal against the need to empower the user beyond the limitations of the physical world. Like ThingLab, ARK is built on top of the Smalltalk environment, and any Smalltalk object can have a graphical representation in an alternate reality.

2.2.4 Impulse-86 and GROW

Impulse-86 [48] is an editor for knowledge bases implemented in the Strobe extension to the Interlisp-D programming environment. Its designers wanted developers and users of knowledge-based systems to be able to design domain-specific interfaces without requiring extensive expertise in interactive graphics.

The system provides editing tools that are specialized for each of three semantic levels: objects and their internal structure, relationships between objects, and systems of objects, relationships, and overall behavior. Editors for new domains are constructed from five building blocks: **Editor**, **EditorWindow**, **PropertyDisplay**, **Menu**, and **Operations**. Each of these building blocks is an object in the Impulse knowledge base that embodies information specifying its function. Most Impulse-86-based editors require some code to be written to implement domain-specific commands. Editors have been built for domains such as graphs and dataflow diagrams.

GROW [3] builds on the capabilities of Impulse-86 to simplify the implementation of an editor's graphical objects. GROW helps the user specify the appearance and semantics of

graphical objects, including techniques for composing them hierarchically and for defining arbitrary dependencies between them. The system can also enforce layout constraints on the graphical objects using its dependency mechanism, in contrast to the more general constraint-solving approaches of Sketchpad and ThingLab. GROW's creators deliberately narrowed the role of constraints in their system, observing that general-purpose constraint management is not necessary in most graphical editors.

2.2.5 ThinkerToy

ThinkerToy is a problem modeling tool for decision support that replaces traditional mathematical models with concrete, directly-manipulable models. Like ThingLab and ARK, ThinkerToy is implemented on top of the Smalltalk environment. Since ThinkerToy attempts to enhance the decision-making process in disciplines as diverse as landscape planning, bond trading, and flow modeling, the system is necessarily multi-domain in nature. Central to ThinkerToy is the **ManiplIcon**, a graphical object whose semantics are exhibited through direct manipulation. ManiplIcons can be assembled into a **tableau** that comprises a decision support tool for a particular domain. Gutfreund [16] describes four experimental tableaus he developed with the system:

1. **Array**, a tableau for storing a group of ManiplIcons (numeric, textual, graphical) that supports spreadsheet functionality,

2. **Chart**, a tableau for analyzing two-dimensional data,

3. **TerrainMap**, a tableau for analyzing three-dimensional data, and

4. **DataFlow**, a tableau for modeling flow rate problems.

Although ThinkerToy is targeted explicitly at decision support and thus could be deemed a domain-specific editor, in practice this "domain" encompasses a broad range of disciplines. Moreover, ThinkerToy can be extended through composition of existing ManiplIcons to form new tableaus, and new ManiplIcons can be defined in Smalltalk.

2.2.6 LabVIEW

LabVIEW [29, 42] is a commercial system that, like ThinkerToy, is not targeted specifically towards multiple domains but nonetheless satisfies our criteria for a multi-domain editor. LabVIEW was originally intended to provide a graphical interface to instruments connected by a IEEE 488 bus. Over time, however, the system developed into a editor that nearly qualifies as a graphical programming environment. The key element of LabVIEW is the **virtual instrument**, an object having a graphical representation with underlying semantics. A virtual instrument can accept inputs from the user or from an external source (such as a file, I/O port, or another virtual instrument), perform arbitrary computations, and generate output in the form of control signals, ASCII data, or graphics. An instrument on the 488 bus can be represented on the screen with a virtual instrument; entirely abstract virtual instruments, having no corresponding device on the bus, can also be defined. Virtual instruments are connected to each other much like real instruments using graphical wires.

Virtual instruments are defined in two parts: the user must specify the appearance of the instrument's **front panel** and define the instrument's semantics by constructing a **block diagram** in a graphical programming language. The front panel is composed with various predefined, realistically rendered controls and indicators such as switches, meters, dials, and paper tape generators. The parameters controlled by these elements are represented by graphical variables in the block diagram. The user defines computations on these variables in a graphical syntax resembling dataflow, complete with graphical representations of familiar programming language constructs such as conditional branches, iterative loops, and subroutines (nested diagrams).

We classify LabVIEW as a multi-domain editor because the virtual instrument concept is general enough and supported well enough that users can use LabVIEW in different domains. The manufacturer describes LabVIEW as a "software construction environment for engineering and scientific applications"; as such it is designed to be extended easily by letting the user create and compose virtual instruments. The product literature [28] demonstrates how to create instruments for graphical calculation and theorem proving; ecological, process control, and signal transmission simulations; and data acquisition and analysis.

Based on the classification criteria, it is clear that LabVIEW is not restricted to a single domain. We have stopped short of calling it a graphical programming environment, however, even though it supports graphical programming. LabVIEW it is not intended to replace or augment a conventional programming language and environment, which is an attribute common to the graphical programming environments described in the next section.

2.3 Graphical Programming Environments

It has been a long-standing dream of researchers to let users program by drawing pictures. Experienced programmers often use graphical notations to diagram their algorithms before turning them into code. Novices often find programming difficult because they are uncomfortable with the rigid syntax of textual languages. By specifying programs in graphical terms that closely match the programmer's mental pictures, the expert can simply draw his algorithms; the novice can show the computer how to perform its task. Graphical programming environments would thus make programming easier for everyone, and creating domain-specific editors would be a natural extension of their capabilities.

Many systems have been designed to support graphical programming. Myers [26], Raeder [36], and Shu [43] provide surveys of work in the field. Myers classifies systems that use graphics for program specification into two groups: **visual programming** systems, and **programming-by-example** systems.

2.3.1 Visual Programming

A visual programming system allows the user to specify algorithms and data structures graphically as an alternative to conventional textual languages. Flow chart, data flow, and message passing notations are typically used as graphical programming languages. Two representative systems are Pict [12] and Garden [37, 38].

Pict uses a flowchart paradigm to model programs graphically. Icons representing conventional programming language constructs are connected by paths that indicate flow of control. The interface is almost entirely non-textual. Color is used liberally to enhance the appearance and understandability of the program. Icons, paths, and four six-digit,

nonnegative integer variables are color coded to indicate their interrelationships. Pict aids in the layout of program components by finding the shortest and straightest path possible between selected icons, and it redefines and moves paths previously laid down if necessary. The user's program is continuously checked for correct syntax; illegal constructs are disallowed. The Pict run-time supervisor interprets and animates the program to depict its execution visually.

Garden differs from Pict in that instead of limiting the programmer to a flow chart paradigm, Garden lets him design and implement one or more graphical languages in which to program. The environment supports the creation of these languages and their use in the programming process.

A new graphical language is defined in three steps. First, the programmer defines the objects that underlie the graphical representations in the language. Next, the semantics of the objects are specified. Finally, a visual and textual syntax is associated with each object. Once the graphical language is developed, it can be used in conjunction with the other graphical languages in the environment to create complete programs. Thus, the programmer can use whatever language is most appropriate to express a particular computation.

2.3.2 Programming-by-Example

A programming-by-example system allows the user to demonstrate how to manipulate graphical representations of data objects to accomplish a task. The system then infers algorithms from the user's actions and generates a program to implement them. Programming-by-example emphasizes "doing" rather than "telling." The purpose of this is to make programming easier, especially for the novice, by abstracting away implementation details and letting the user program by carrying out familiar actions on concrete representations of data.

Pygmalion [46] was one of the first programming-by-example systems. Later systems include Rehearsal World [15], which pioneered a theatrical programming metaphor, Halbert's SmallStar [17], and ThinkPad [40], a programming-by-example system incorporating constraints.

2.4 Unsolved Problems

None of the systems categorized in the preceding sections fulfills the goals outlined in Section 1.3 for supporting the construction of practical graphical object editors, largely because none is designed specifically for this purpose. In this section we discuss the weaknesses in this regard of each type of system.

The obvious problem with domain-specific editors is that they support editing only in the domain for which they were designed. Thus, the versatility per unit implementation effort of these editors is low. For each domain-specific editor, the programmer must design the user interface, deal with many implementation details, recode functionality common to many other editors, and provide enough flexibility to avoid reprogramming later.

In general, multi-domain systems offer few abstractions for graphical object editing. These systems support extensibility by imperative or declarative means without providing more focused abstractions, for example, for direct manipulation and generating alternate representations of domain-specific objects. In particular, multi-domain editors that rely solely on constraint mechanisms for generality suffer performance problems when they are used for substantial editing tasks. Constraint specifications tend to grow quickly as the relationships between objects become more complex. Solutions to the resulting simultaneous equations are not guaranteed to be computable in reasonable time or even to exist at all. Moreover, most multi-domain systems are tightly coupled to an underlying exploratory programming environment, essentially precluding their integration into other environments. The expressiveness of a general-purpose programming language is indeed necessary for supporting graphical editing in multiple domains. But systems that have relied on a few broad abstractions and an elaborate run-time environment have proven uncompetitive with hand-crafted applications in terms of their efficiency and ability to work with other systems.

Graphical programming environments have proven inadequate as well. Though many such environments have been developed, none has succeeding in supplanting textual programming. Graphical programming languages generally lack efficiency of expression. They are adequate for describing simple algorithms and data structures but quickly become unwieldy for specifying more sophisticated constructs. Since the program is expressed in

pictures, it grows spatially, often resulting in a sprawling, unstructured mass of lines and boxes. The user must then pan about the diagram to search for information of interest, a task complicated by the lack of suitable sorting and searching utilities—facilities taken for granted in textual systems. Moreover, most graphical programming systems are interpretive and must deal with considerable overhead associated with pictorial representations. Thus, performance is acceptable only for simple graphical programs.

Programming-by-example has added shortcomings. Deducing non-trivial algorithms from a limited number of user actions is difficult. Often there is no way to infer the precise behavior for a program because the demonstration simply does not convey all the necessary information. A minor change in the demonstration can radically alter the code produced. Choosing the definitive examples usually requires considerable forethought to ensure that unambiguous and non-redundant information is supplied to the system, thus limiting its utility.

A practical graphical programming environment remains an elusive goal, and it is arguably more than we need in the first place. Our goal is to support graphical object editing, not general-purpose programming. Graphical object editors share many features and have interfaces that are considerably less flexible than a programming language. This suggests that a subset of the abstractions needed for graphical programming will suffice in creating editors for new domains.

2.5 Summary

The goals of this thesis were to create a framework for building a variety of graphical object editors in less time and with performance and functionality comparable to conventionally-developed graphical editors. Domain-specific editors are not sufficient because they are designed for a particular domain. Current multi-domain editors are generally not flexible or efficient enough to substitute for a given domain-specific editor. Graphical programming environments have not proven to be a practical replacement for traditional textual programming environments, especially for building large systems.

Unidraw avoids these shortcomings by providing an object-oriented architecture that

- is organized as a collection of classes some or all of which are incorporated into an editor for a particular domain,
- provides abstractions that embody functionality common to graphical object editors,
- avoids overgeneralization by supporting simple subsets of more general facilities (for example, constraint solving),
- supports a compiled implementation for efficiency, and
- leverages existing programming environments and user interface tools to aid in editor development.

The next chapter describes the Unidraw architecture in detail.

Chapter 3

Unidraw Architecture

There are two users in the Unidraw context: the programmer who uses Unidraw objects to build a domain-specific editor, and the end-user of that editor. It is the programmer who must understand the Unidraw architecture, for it is his responsibility to create a working editor; the end-user, on the other hand, does not need to be at all familiar with Unidraw or even know that the editor was built with it. To the end-user, the domain-specific editor is just another stand-alone application.

The purpose of this chapter is to describe the Unidraw architecture in enough detail to allow one to develop a useful implementation. We begin with an overview of the architecture, relating the philosophy behind it, outlining its major elements, and showing how the elements are assembled to form an editor for a particular domain. Then we consider the elements in detail, describing their semantics and relationships. We conclude the chapter with a summary of the architecture.

3.1 Overview

Unidraw spans the semantic gap between traditional user interface toolkits and the implementation requirements of graphical object editors. An editor for a particular domain relies on Unidraw for its graphical editing capabilities, on the toolkit for supporting the "look and feel" of the user interface, and on the window and operating systems for managing workstation resources.

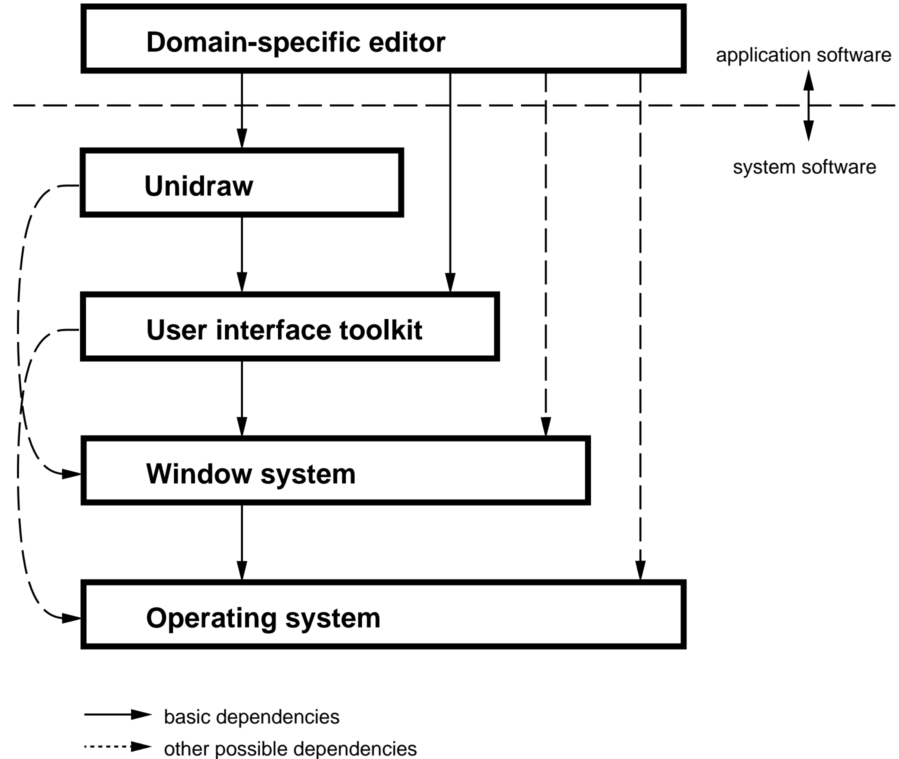


Figure 1: Layers of software underlying a domain-specific editor

Figure 1 depicts the dependencies between the layers of software that underlie a domain-specific editor based on Unidraw. The operating and window systems provide the lowest level support. Above the window system level are the abstractions furnished by the user interface toolkit, including buttons, scroll bars, menus, and a framework for composing them into generic interfaces. Unidraw stands at the highest level of system software, contributing abstractions that are closely matched to the requirements of graphical object editors.

A fundamental design decision in Unidraw was to adopt an object-oriented model in which objects encapsulate the common attributes of domain-specific editors. Objects are appropriate for modeling these attributes for the following reasons:

- *Objects facilitate the conceptualization and representation of an idea.* The concept of a transistor, for example, is readily modeled and implemented in an object of class

Transistor that mimics the behavior of real transistors. The transistor abstraction is thus reflected directly in its representation.

- *Objects are a good program structuring and complexity control mechanism.* The operations defined on an object class form a **protocol** that subclasses understand. Since objects encapsulate state and behavior, inter-object dependencies are limited by the semantics of the protocol. Objects are thus insulated from one another, promoting modularity and reusability.

- *Inheritance and dynamic binding make extension easy.* Programmers can create new classes of object incrementally, that is, by specifying only those aspects that differ from another class. A general concept can be embodied in a base class from which classes with particular behavior are derived. Dynamic binding allows objects from the same class hierarchy to be treated uniformly, independent of their subclass, simply by manipulating them in terms of the protocol defined in the base class.

An object-oriented architecture has several advantages over more traditional approaches. Objects that model reality give the application programmer insight into the "proper" way to create a domain-specific editor. Such objects are suggestive of their intended purpose and thus focus the programmer's attention on their role rather than the mechanics of their use.

Moreover, partitioning editor functionality into a set of base classes narrows the design space. This frees programmers from making design decisions that are independent of the editing domain. For example, how the system supports multiple, mutually-consistent views of a domain-specific diagram is a design decision that spans many kinds of editors. The architecture can provide for multiple views with standard objects and protocols, allowing programmers to concentrate on more domain-specific issues.

Finally, the inherent extensibility of an object-oriented model makes library-based implementations attractive. The object classes and protocols can be packaged as a library of components dedicated to the construction of domain-specific editors. Non-object-oriented libraries are usually far less extensible than object-oriented ones primarily because they lack inheritance. Such libraries compensate by offering a broader range of functionality to lessen the need for extension, but this is not feasible in a library that is meant to support graphical object editing in general. An object-oriented model thus enhances a library-based

implementation, which in turn gives programmers more flexibility to create production-quality editors.

3.1.1 Basic Abstractions

In designing the architecture we focused on the common attributes of domain-specific editors. The attributes we identified are reflected in four class hierarchies:

1. **Components** represent the elements in a domain: for example, geometric shapes in technical drawing, schematics of electronic parts in circuit layout, and notes in written music. Components encapsulate the appearance and semantics of these elements. A domain-specific editor's main objective is to allow the user to arrange components to convey information in the domain of interest.

2. **Tools** support direct manipulation of components. Tools employ animation and other visual effects for immediate feedback to reinforce the user's perception that he is dealing with real objects. Examples include tools for selecting components for subsequent editing, for applying coordinate transformations such as translation and rotation, and for connecting components.

3. **Commands** define operations on components and other objects. Commands are similar to messages in traditional object-oriented systems in that components can receive and respond to them. Commands can also be executed in isolation to perform arbitrary computation, and they can reverse the effects of such execution to support undo. Examples include commands for changing the attributes of a component, duplicating a component, and grouping several components into a composite component.

4. **External representations** convey domain-specific information outside the editor. Each component can define one or more external representations of itself. For example, a transistor component can define both a PostScript representation for printing and a netlist representation for circuit simulation; each is generated by a different class of external representation. An external representation object thus defines a one-way mapping between a component and its representation in an outside format.

The Unidraw architecture provides base classes for component, command, tool, and external representation objects. Subclasses implement the behavior of their instances according to the semantics of the protocol defined by their base class. For example, components support operations that define how commands affect their internal state. The partitioning of graphical object editor functionality into components, commands, tools, and external representations is the foundation of the Unidraw architecture.

3.1.2 Subjects and Views

A well-established user interface concept is the distinction between (1) the state and operations that characterize objects, and (2) the way the objects are presented in a particular context. In Unidraw this distinction is manifest in the separation of components into **subject** and **view** objects.[1] A subject encapsulates the context-independent state and operations of a component. A view supports a context-dependent presentation of the subject. A component subject may have any number of component views, each offering a different representation of and interface to the subject. A subject notifies its views whenever its state is modified to allow them to change their state or appearance to reflect the modification.

A component subject maintains information that characterizes the component; in the case of a logic gate component, for example, the subject might contain information about what is connected to the gate and its current input values. Different views of the subject can reflect this information in distinctive ways and can provide additional information as well. One view can depict the gate graphically by drawing the appropriate logic symbol, and it might also define what it means to manipulate the gate with a tool. Another view can provide the external representation of the gate by generating a netlist from the connectivity information in the subject.

Subjects and views also let users edit components out of context. While one view of a logic gate shows it buried in a larger circuit, another view can be edited in isolation. Since different views of a subject are independent of one another, the isolated view can be scrolled and zoomed to make it easier to edit without having to extract the component from the circuit. This makes it easy to manipulate an otherwise inaccessible component.

[1] Throughout the thesis, the term "component" unqualified by either "subject" or "view" refers to both the subject and its view(s).

3.1.3 Structure of a Unidraw-Based Application

Figure 2 shows the general structure of a domain-specific editor based on the Unidraw architecture. The diagram depicts five classes of object:

1. Two component subjects appear at the bottom level in the diagram. Component subjects can contain a hierarchy of subcomponent subjects; the component subject in the lower left of the diagram contains two subcomponent subjects. Hierarchical composition of component subjects is the basis for such features as grouping in a drawing editor and eliding circuit elements in a schematic capture system. An entire domain-specific drawing is represented by a composite component subject that can be incorporated into a larger work.

2. At the second level from the bottom are the corresponding views of the subjects. Note that the right-hand component subject has two views attached. A component view can be composed of other views to reflect its subject's structure, or the view and subject structures can be independent.

3. Each component view is placed in a **viewer** at the third level. A viewer displays a graphical component view, most often the root view in a hierarchy. A viewer provides a framework for displaying the view, supporting such "non-semantic" manipulations as scrolling and zooming. Viewers also take raw window system or toolkit events and translate them to conform to standard Unidraw protocols.

4. An **editor** associates tools and user-accessible commands with one or more viewers and combines them into a coherent user interface. An editor also uses a **selection** object, which manages a list of distinguished component views. A Unidraw-based application can create any number of editor objects, allowing the user to work on multiple views of the same or different domain-specific components.

5. Operations that require inter-editor communication or coordination access the **unidraw** object, a one-of-a-kind object maintained by the application. For example, commands that open and close editors and quit the application must access this object. The unidraw object also maintains logs of commands that have been executed and reverse-executed to support arbitrary-level undo and redo.

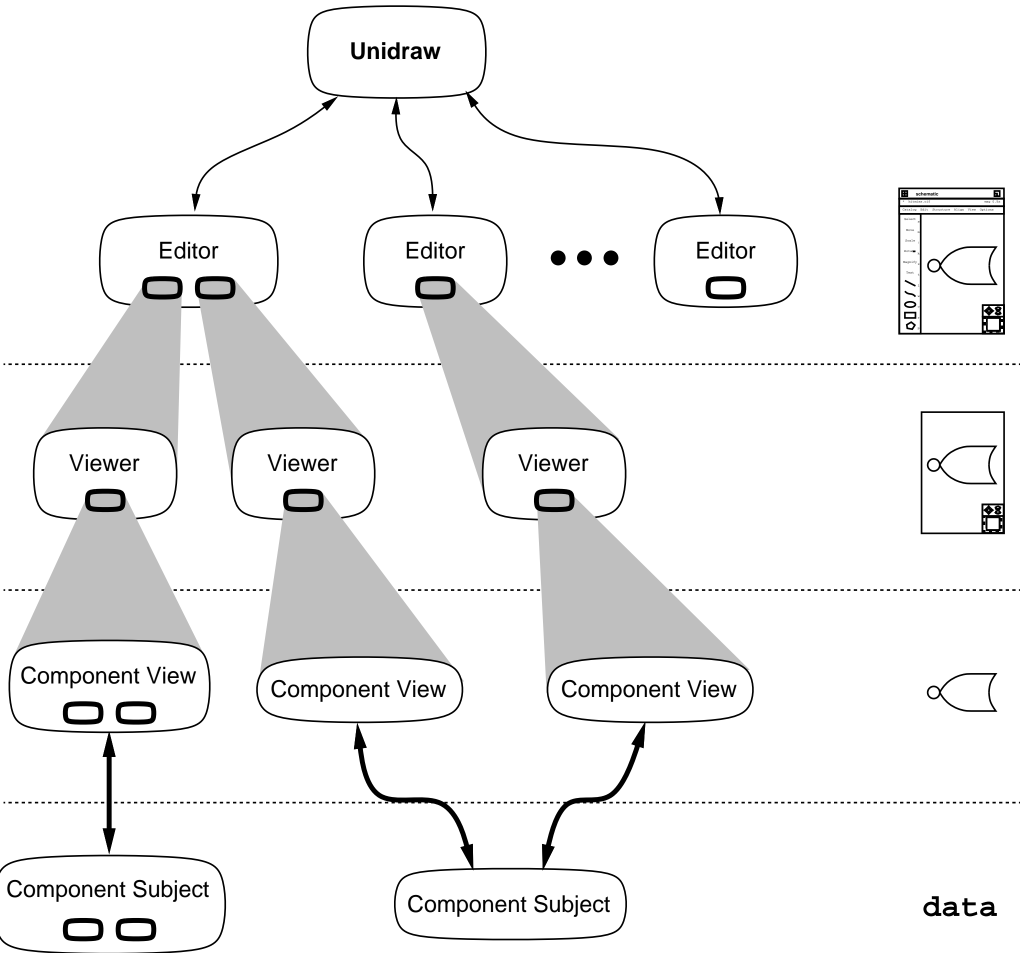


Figure 2: General structure a domain-specific editor based on Unidraw

Not shown in the diagram is the **catalog** object, which manages a database of components, commands, and tools. At minimum, a domain-specific editor uses the catalog to name, store, and retrieve components that represent user drawings. An editor could also access uninstalled commands and tools and incorporate them into its interface at run-time.

This structure provides a standard framework for building domain-specific editors, yet it allows substantial latitude for customized interfaces. Nothing in this architecture dictates, for example, a particular look and feel for a given editor object. A domain-specific editor may define editor objects that use separate windows for their commands, tools, and

return values	*operation*	*arguments*
	Attach	component view
	Detach	component view
	Notify	
	Update	
	Interpret	command
	Uninterpret	command
component subject	GetParent	
	{*child iteration and manipulation operations*}	
state variable	GetStateVariable	name
transfer function	GetTransferFunction	

Table 1: Component subject protocol

viewers. The architecture only specifies how the editor mediates communication between components and the commands, tools, and viewers that affect them.

3.2 Components

Components are perhaps the most important class of objects in Unidraw. Components represent the objects of interest in the editing domain. They define and manage information to mimic the behavior of real-world objects. They can maintain one or more graphical representations for themselves and can support non-graphical representations as well. Components also define how they respond to commands and tool manipulation. Developing an editor for a new domain centers largely on choosing, designing, and implementing a good set of components.

3.2.1 Subject and View Protocols

The Unidraw architecture defines separate communication protocols for component subjects and views. Table 1 lists the subject protocol's basic operations.

Component subjects define Attach and Detach operations to establish or destroy a connection with a component view. These operations assume the specified view is compatible

with the subject; undefined behavior will result if an incompatible view is specified. Notify alerts the subject's views to the possibility that their state is inconsistent with the subject's. Upon notification, a view reconciles any inconsistencies between the subject's state and its own. The Update operation notifies the subject that some state upon which it depends has changed. The subject is responsible for updating its state in response to an Update message.

A component subject can be passed a command object to interpret via the Interpret operation. The semantics of this operation are component-specific; the subject typically retrieves information from the command for internal use or executes the command. The Uninterpret command allows the component to negate the effects of interpreting a command; the subject might undo internal state changes based on information on the command, or it might simply reverse-execute the command.

The GetParent operation returns the component subject's parent (if any) to allow traversal up the subject hierarchy. Component subjects also define a family of operations for iterating through their child subjects (if any) and for reordering them. Only the immediate children of the subject can be accessed and manipulated with this interface; other descendants must be accessed through the children. Finally, components can define one or more **state variables** and one **transfer function** (described in Section 3.3) that can be accessed via the GetStateVariable and GetTransferFunction operations, respectively.

The component view protocol duplicates some of the subject protocol's operations (Update, Interpret, Uninterpret, GetParent, and those for iteration and child manipulation) and adds SetSubject and GetSubject operations that set and return the view's subject. SetSubject should be accessible only to the view's subject for use in such operations as Attach and Detach. A subject's Notify operation usually sends an Update message to each of its views. Interpret and Uninterpret are defined on views because some objects (notably viewers and selections) manipulate component views rather than their subjects; thus it is convenient to send a command to a view for (un)interpretation, which may in turn send it to its subject. A component view may have a subcomponent view structure, which may or may not reflect its subject's structure, so the view protocol also defines child iteration and manipulation operations.

return values	*operation*	*arguments*
graphic	GetGraphic	
	SetMobility	mobility
mobility	GetMobility	

Table 2: Additional operations defined for graphical component subjects

return values	*operation*	*arguments*
graphic	GetGraphic	
	Highlight	
	Unhighlight	
manipulator	CreateManipulator	tool, event
command	InterpretManipulator	manipulator

Table 3: Additional operations defined for graphical component views

3.2.2 Graphical Components

An important extension of the basic component protocols adds operations to support **graphical components**—components that can be displayed in a viewer. To support this extension, we introduce the concept of a **graphic**, an object that contains graphics state and geometric information. A graphic uses this information to draw itself and to perform hit detection. Graphical components use graphic objects in both their subjects and views to define their appearance. Since components can be structured hierarchically, graphics must also support hierarchy; thus, a composite graphical component can define its appearance by assembling its subcomponents' graphics into a composite graphic.

Table 2 shows the operations added to the basic component subject protocol for graphical component subjects (or "graphical subjects" for short). By definition, graphical subjects encapsulate their geometric and graphics state in a graphic, permitting a standard interface for retrieving this information. The GetGraphic operation returns the information in the subject's graphic. Graphical subjects can also have **mobility**, and the protocol provides operations for assigning and retrieving this attribute. Later we show how mobility affects the component's connectivity semantics.

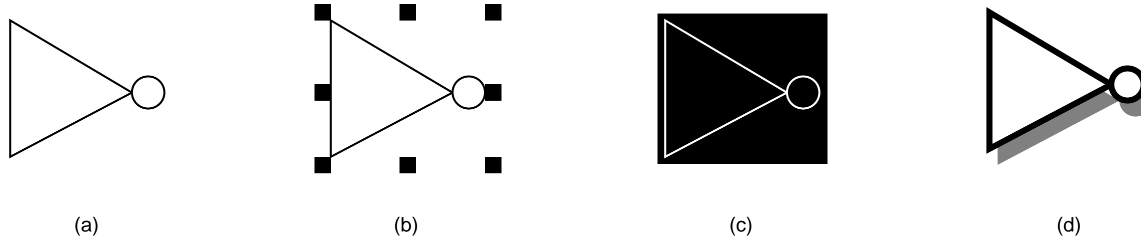


Figure 3: Various highlighting styles: (a) unhighlighted, (b) handles, (c) color reversal, (d) substitution

Several operations augment the basic component view protocol to support graphical component views (“graphical views” for short). These operations are shown in Table 3. Graphical views maintain their own graphic to characterize their appearance. The view’s graphic is thus independent of the subject’s graphic. In practice, however, most views define some dependency between the two graphics. For example, an eighth note subject in a music editor might keep a graphic that defines the note’s color and size; the corresponding eighth note view would have a graphic with a matching color and size plus a B-spline defining the note’s outline on the screen.

Selection is a fundamental concept in graphical object editors. The user must be able to specify components of interest, and those components should distinguish themselves graphically once they are specified. The graphical view protocol adds Highlight and Unhighlight operations to allow components to distinguish themselves graphically in an appropriate way. For example, highlighting may take the form of “handles” surrounding the component (Figure 3b), reversing its color scheme (Figure 3c), or substituting another graphic altogether (Figure 3d). The Highlight operation lets the graphical view define its highlighted appearance, and the Unhighlight operation allows it to revert to its unhighlighted state.

The last two operations in Table 3, CreateManipulator and InterpretManipulator, define how a graphical view reacts when it is manipulated by a tool and how the tool affects the component following manipulation. Both operations rely on an intermediary object called a **manipulator** to characterize the manipulation. Manipulators abstract and encapsulate the mechanics of direct manipulation. They provide a standard interface to an abstract state

machine that a tool can use to implement its interaction semantics. Tools and manipulators are described in greater detail in Section 3.5.

Since a graphical view can be manipulated by many tools and can exhibit different dynamic behavior for each, the view must be responsible for creating the proper manipulator for a given tool. The CreateManipulator operation lets the graphical view do this. Once the view has returned a manipulator, the controlling object (usually the enclosing viewer) can use it to carry out the direct manipulation. Following manipulation, only the view that contributed the manipulator is in a position to carry out its desired effect. The InterpretManipulator operation allows the graphical view to extract any information it deems significant from the manipulator and convert it into a command that encapsulates the overall effect of the manipulation. The command can then be processed in same way as any other command, as will be discussed in Section 3.4.

3.3 Component Semantics

The preceding section introduced the basic component classes, including the protocols that govern their appearance and response to commands and direct manipulation. Domain-specific components implement the protocols to fit their semantics. This section discusses Unidraw abstractions that are useful for implementing component semantics common to many domains.

3.3.1 Connectors

A common semantics requires that components maintain connectivity or stay within a prescribed area. For example, a schematic capture system should allow the user to connect circuit components, and it should guarantee that the components stay connected when they are moved. A music editor should confine notes to their position on the staff, allowing them to move only laterally unless their vertical position (signifying pitch) is changed explicitly. Moreover, lateral motion should be limited so that a note cannot slide off the staff.

Unidraw supports connectivity and confinement semantics with the **connector** graphical component subclass. Since connectors are components, each consists of a subject and zero

or more views and can be manipulated directly. Often, however, connectors are embedded in larger components that use them to define their own connectivity semantics.

Standard Connector Subclasses

A connector can be connected to one or more other connectors. Once connected, two connectors can affect each other's position in specific ways as defined by the semantics of the connection. Connector subclasses support different connection semantics, and the architecture defines three such subclasses:

1. A **pin** contributes zero degrees of freedom to a connection. A degree of freedom is an independent variable along a particular dimension, which for connectors is a cartesian coordinate. A pin defines its position.

2. A **slot** supports one degree of freedom within certain bounds. Slots define their position, a finite length, and a slope.

3. A **pad** provides two degrees of freedom within certain bounds. Pads define their position and size and are rectangular.

The semantics of the connection depends on the connectors in it; different combinations of pin, slot, and pad connections yield different semantics. Figure 4 shows how connectors behave in several connections, using the connectors' default graphical representations. The centers of two connected pins must always coincide (Figure 4a). A pin connected to a slot (Figure 4b) is free to move along the slot's major axis until it reaches either end of the slot; the pin cannot move in the transverse dimension. Two connected slots (Figure 4c) can move relative to each other as long as the center lines of their major axes share a point.[2] Finally, Figure 4d shows how a pad-pin connection constrains the pin to stay within the confines of the pad.

[2]These semantics are equivalent to those of a pin connected to two slots. We have defined them explicitly to obviate the introduction of another connector and to allow for a more efficient implementation. The same is true of slot-pad and pad-pad connections.

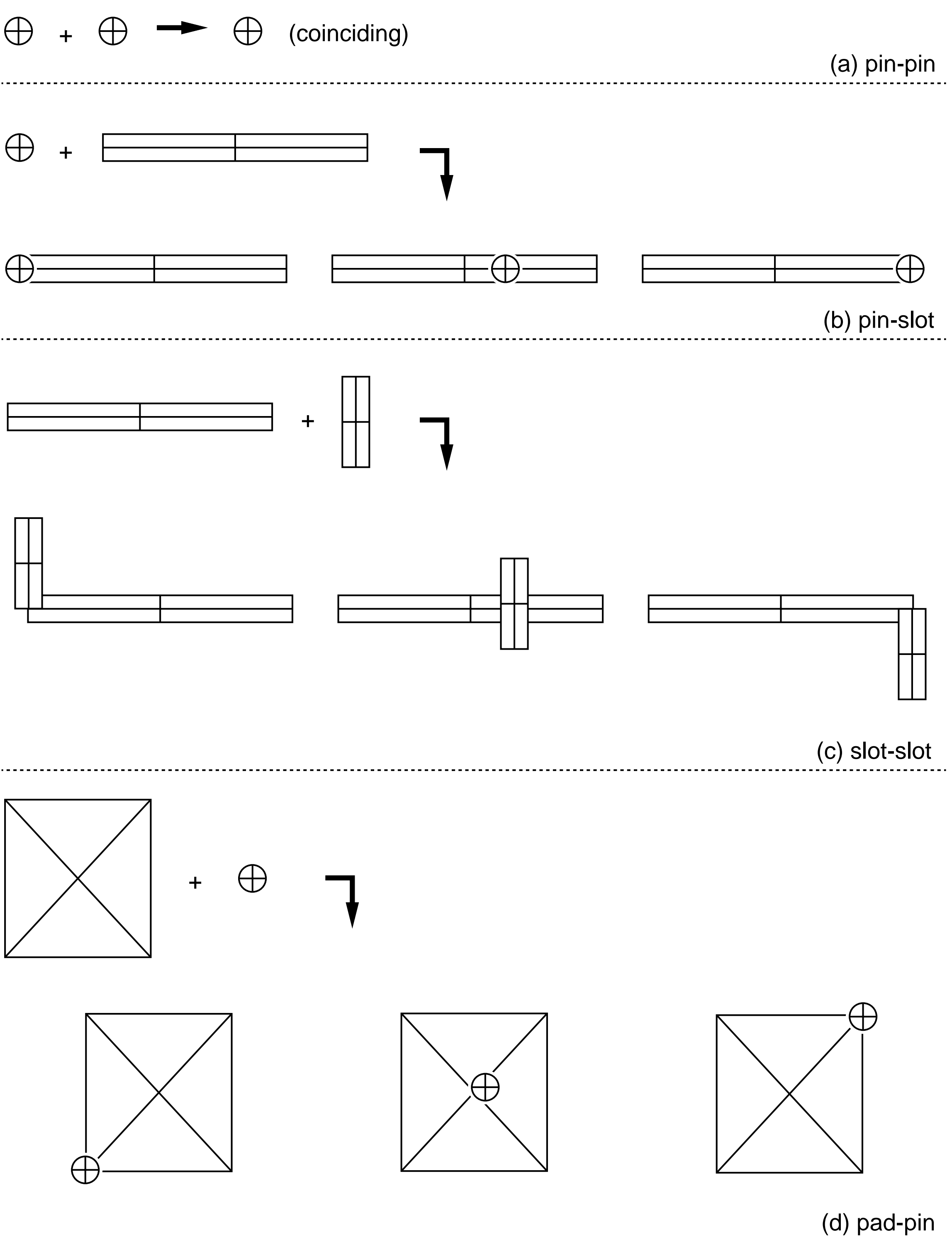


Figure 4: Several connections and their semantics

return values	*operation*	*arguments*
	Connect	connector[, connector glue]
	Disconnect	connector
(x, y)	GetCenter	
	SetBinding	state variable
state variable	GetBinding	
	SetTransmission	transmission method
transmission method	GetTransmission	
	Transmit	

Table 4: Connector subject protocol

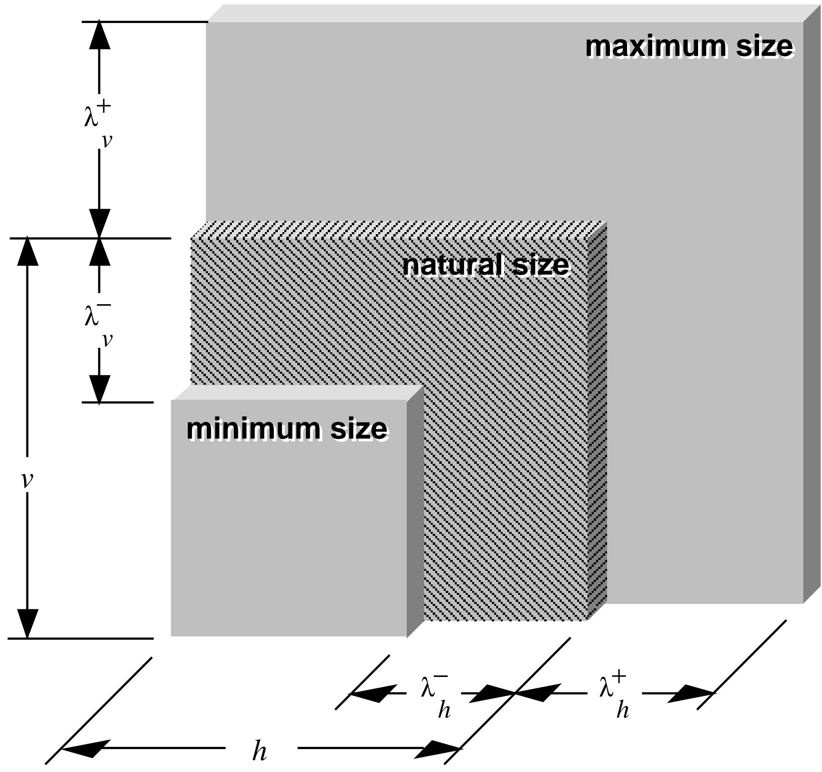
Connector Protocol

Table 4 depicts the connector subject protocol, which extends the graphical subject protocol. The connector view protocol is identical to the basic graphical view protocol.

The Connect operation connects the centers of two connectors, optionally with a piece of **connector glue** interposed. Connector glue is characterized by a natural size, elasticity, and deformation limits (see Figure 5). Elasticity is specified in terms of independent stretchability (ε^+) and shrinkability (ε^-) parameters. Deformation limits are expressed as independent limits on the total amount the glue can stretch (λ^+) and shrink (λ^-). Elasticity is dimensionless; it determines how deformation is apportioned to a collection of mutually-dependent connections.

For example, Figure 6 shows three pins P1, P2, and P3 connected “in series,” which implies the two connections share a connector. Glue is represented schematically (and in one dimension only) with resistor symbols. The elasticity (both shrinkability and stretchability) of glue G1 is twice that of G2. Thus if P1 and P3 are pushed together or pulled apart, then G1 will shrink or stretch twice as much as G2. In general, ideal connector glue behaves like two non-linear springs, one horizontal and the other vertical, each having independent spring constants and travel limits in tension (ε^+, λ^+) and compression (ε^-, λ^-).

The Disconnect operation dissolves a connection to a component, deleting any associated connector glue. GetCenter returns the coordinates of the connector’s center, which



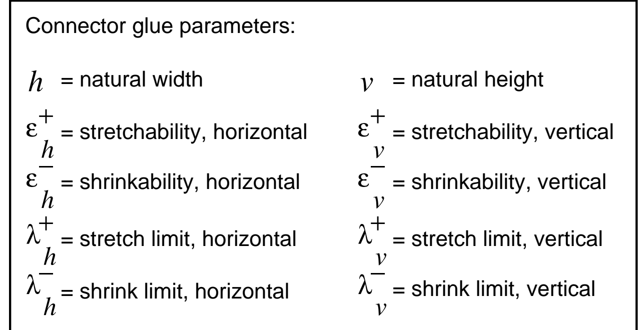


Figure 5: Connector glue characteristics

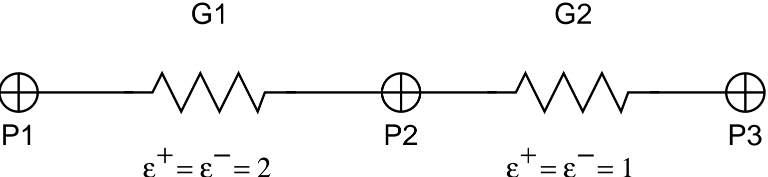


Figure 6: Three pins connected in series

defines its position. The last five operations support dataflow semantics and are described in Sections 3.3.4 and 3.3.5.

Mobility

Figure 4 presents the behaviors of several connections without specifying which connector moves relative to the other. The connectors' mobilities characterize how each connector moves to satisfy the connection constraints.

Recall that the graphical subject protocol defines operations for setting and retrieving a mobility attribute. The attribute can have one of three values: **fixed**, **floating**, or **undefined**. In general, a fixed component's position cannot be affected by a connection, regardless of the connection's semantics, while a floating component will move to satisfy the connection's semantics. The behavior of a component with undefined mobility is indeterminate. Composite components often have undefined mobility to avoid overriding their children's mobilities.

Mobility specifications disambiguate the semantics of a connection. In Figure 4b, for example, it is unclear which connector, the pin or the slot, actually moves. If, however, the slot's mobility is fixed and the pin's is floating, then the pin will always move to satisfy the connection constraints. If the slot is moved explicitly, then the pin will follow to stay within it. An attempt to explicitly move the pin beyond the slot's bounds will fail; in fact, if the pin is also connected to another, orthogonal slot, *any* attempt to move it explicitly will fail. As a corollary, a connection can have no effect on two fixed connectors.

The concept of mobility is essential to connectors, but it applies to all graphical components. Often, a composite component containing several connectors will not define a mobility attribute itself but will define SetMobility to set its components' mobilities. Its components (such as connectors) that define a mobility attribute will then set it accordingly. The composite component might compute its own mobility from the mobilities of its components.

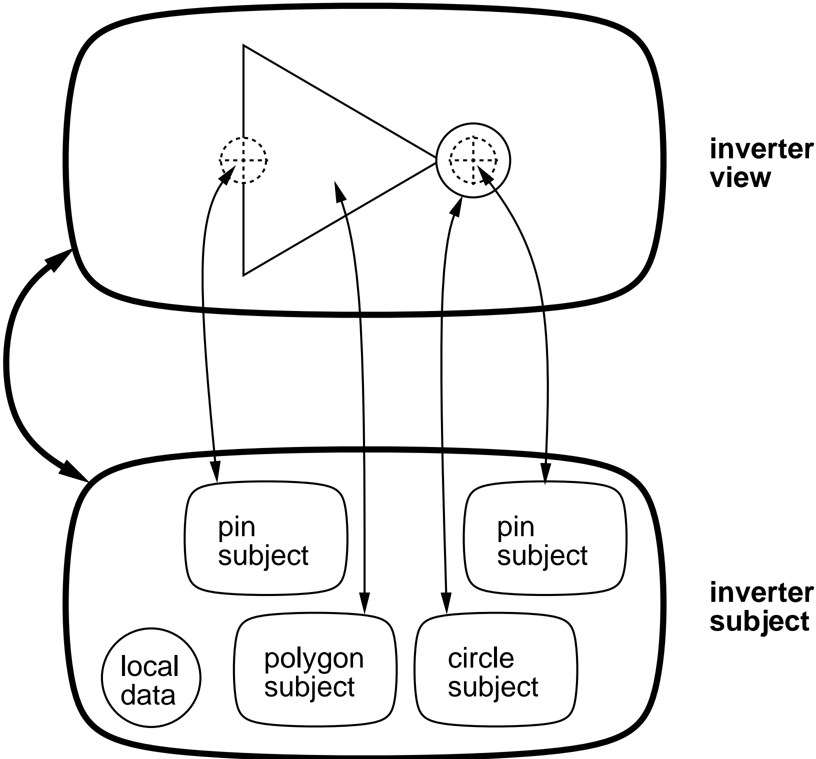


Figure 7: Possible composition of an inverter component

3.3.2 Domain-Specific Connectivity

We have discussed connectors and their semantics, but we have not explained how they support connectivity between domain-specific components. For instance, how does one define an inverter schematic component whose wires remain connected when the user moves it? Such domain-specific components are often compositions of simpler components.

Figure 7 shows how the inverter subject and view can be composed with polygon, circle, and pin subjects and views. Note that the pins are treated as any other component in the composition, but they have a special responsibility to define the inverter's connectivity semantics. The inverter sets their mobility according to the desired semantics. For example, it will fix both if their position should not be affected by any connections in which they participate. When the inverter interprets a command to move itself, it moves all its components accordingly, including the pins. Since the pins are fixed, they will not be

affected by their connections; rather, any floating connectors that are connected to the pins will move according to the connection semantics.

To illustrate further, consider a wire component composed of a line and a floating pin at each endpoint. The wire implicitly connects its pins with a piece of connecting glue reflecting its size and desired elasticity. This connection has several implications. The pins could shift temporarily relative to the line as they move to satisfy connectivity constraints imposed by external connectors. The wire subject will be notified automatically of its pins' movement via its Update operation. It might then change its shape to reflect the new pin position(s). The wire can thus deform as the connecting glue stretches or shrinks; in effect it acquires the elastic properties of the glue. Therefore, when the user moves an inverter with a wire connected to it, the wire will stretch or shrink to maintain the connection.

Composite components can thus define connections between their internal connectors, and they can base their appearance on their connection semantics. This provides a simple way to extend the canonical connectivity behavior to domain-specific components.

3.3.3 State Variables

Component subjects often maintain state on which other subjects depend or state that must be accessible to the user. State variables provide a standard way to represent and access this state. State variables are commonly used to allow user modification of component attributes and to support dataflow between components.

Like components, state variables are partitioned into subjects and views. The state variable subject represents a typed datum, and views provide a graphical interface that lets a user examine and modify the subject. The GetStateVariable operation defined on component subjects (see Table 1) provides external access to a component's state variables through a component-specified naming convention.

The basic state variable subject protocol defines Attach, Detach, and Notify operations analogous to those for component subjects. Subclasses often augment the basic protocol to include assignment and arithmetic operators or other functions. The view protocol defines SetSubject and GetSubject operations and an Update operation for reconciling the view's

state with its subject's. Because state variable views do not interact with other Unidraw objects, their implementation can be based directly on toolkit objects.

If the inverter component discussed previously was meant to support logic simulation, then the inverter subject might define two state variables, **input** and **output**, to represent the logic levels at its input and output. These state variables could be instances of a **LogicLevel** subclass of the state variable subject. The LogicLevel subject maintains a boolean value to store the logic value. Instances of a **LogicLevelView** class could provide a user interface to viewing or changing the logic level using LED and switch metaphors, respectively.

The relationship between a component and its state variables is analogous to that of an object and its instance variables. Indeed, if the architecture is implemented in an object-oriented language, it may be unclear to the programmer whether to use state or instance variables for a component's local data. As a rule, state variables are appropriate for data the user will view or change, or for data that can be affected by other components through Unidraw's dataflow mechanism (described in Section 3.3.5). Instance variables may be more appropriate for other local data.

3.3.4 Transfer Functions

As was shown in Section 3.2.1, the component subject protocol defines an operation for retrieving a component's transfer function. The transfer function is an object that defines relationships between state variables. For example, the inverter could use an **Invert** transfer function to establish a dependency between a pair of logic level state variables; Invert assigns the inverse value of the input variable to the output variable.

The basic transfer function protocol consists of a single operation, Evaluate, that instructs the transfer function to evaluate the dependencies on its state variables. A component can have an arbitrary number of state variables, but the protocol allows for only one transfer function. Thus transfer function subclasses normally add operations for specifying and distinguishing between different state variables. Components that require a combination of existing transfer functions must use a transfer function subclass that composes other transfer functions.

3.3.5 Dataflow

Communication between components is often tied closely to their connectivity. Unidraw provides a standard way for components to communicate via dataflow and for associating dataflow with their connectivity. The dataflow model unites components, connectors, state variables, and transfer functions to make dataflow semantics straightforward to implement.

A state variable can be **bound** to a connector. The SetBinding and GetBinding operations defined by the connector subject protocol (see Table 4) respectively set and get the state variable currently bound to the connector. This binding is meaningless unless the connector is connected to another connector also having a bound state variable. Connectors define "parameter passing" or **transmission** semantics for any bound state variable, one of **in**, **out**, or **inout**. When connected, two connectors with bound state variables will pass their values according to their respective transmission methods. For example, an in connector's state variable will receive the value of an out connector's variable. The parameter passing semantics between incompatible connectors (such as two out connectors) are undefined; such connections should be disallowed by the tool or command making the connection. The SetTransmission and GetTransmission operations set and get the transmission method on a connector.

Transfer functions complete the dataflow model by participating in the propagation of state variable values. The component's transfer function maintains dependencies between state variables, modifying one set of variables based on the values of another set. Thus values can change as they propagate from one component to another.

Transmission of state variable values occurs "instantaneously" between connected connectors to which they are bound. Values are guaranteed to propagate until they reach a connector through which propagation has already occurred; circularities are thus avoided. Propagation begins whenever Transmit is called on a connector (for example, following modification of the bound state variable), and the rate at which propagation proceeds is nondeterministic. For example, if a state variable can be affected through two connection paths, it is indeterminate which path will affect it first. However, since transfer functions can define dependencies between state variables, transfer functions can be used to "latch"

return values	*operation*	*arguments*
	Execute	
	Unexecute	
boolean	Reversible	
	Store	component subject, any
any	Recall	component subject
	SetClipboard	clipboard
clipboard	GetClipboard	
	SetEditor	editor
editor	GetEditor	
	{*child iteration and manipulation operations*}	

Table 5: Command protocol

the values of input state variables onto outputs through a control input. Transfer functions can thus support synchronization.

3.4 Commands

Commands are analogous to messages because they can be interpreted by components. Commands are also like methods in that they are executable, and they resemble transactions because they can be reverse-executed to a previous state. Some commands may be directly accessible to the user through menus, while others are used only by the editor internally. In general, an undoable operation should be carried out by a command object.

Table 5 shows the basic operations defined by the command protocol. Execute performs computation to carry out the command's semantics. Unexecute performs computation to reverse the effects of a previous Execute, based on whatever internal state the command maintains. A command is responsible for maintaining enough state to reverse one Execute operation; repeated Unexecute operations will not undo the effects of more than one Execute. Multilevel undo can be implemented by keeping an ordered list of commands to reverse-execute. It may not be meaningful or appropriate, however, for some commands to reverse their effect. For example, it is probably not feasible to undo a command that

generates an external representation. The Reversible operation returns whether or not the command is unexecutable and uninterpretable. If the command is irreversible, then it can be ignored during the undo process.

Since a command can affect more than one component, the command protocol must allow components that interpret the command to store information in it that they can use later to reverse its effects. The Store operation allows a component to store information in the command as part of its Interpret operation. The component can retrieve this information later with the Recall operation if it must uninterpret the command. Commands that operate on selected or otherwise distinguished components must also maintain a record of the component subjects they affected, in the order they were affected. Commands therefore store a **clipboard** object, which can be assigned and retrieved with the SetClipboard and GetClipboard operations. A clipboard keeps a list of component subjects and provides operations for iterating through the list and manipulating its elements. Typically, the clipboard is initialized with the component subjects whose views are currently selected when the command is first executed. Purely interpretive commands should define their Execute and Unexecute functions to invoke Interpret and Uninterpret on the components in their clipboard.

It is often convenient to create "macro" commands, that is, commands composed of other commands. The command protocol includes operations for iterating through and manipulating its child commands, if any. By default, (un)executing or (un)interpreting a macro command is identical in effect to performing the corresponding operations on each of its children. Finally, the protocol provides SetEditor and GetEditor operations to set and get the editor that owns the command, whose state the command potentially affects. SetEditor is generally called once as part of the command's initialization.

3.5 Tools

By definition, a graphical object editor supports the direct manipulation model of interaction. Unidraw-based editors use tool objects to allow the user to manipulate components directly. The user *grasps* and *wields* a tool to achieve a desired *effect*. The effect may involve a change to one or more components' internal state, or it may change the way components

return values	*operation*	*arguments*
manipulator	CreateManipulator	event
command	InterpretManipulator	manipulator
component subject	GetPrototype	
	SetEditor	editor
editor	GetEditor	

Table 6: Tool protocol

are viewed, or there may be no effect at all (if the tool is used in an inappropriate context, for example). Tools reinforce the user's belief that the components are physical objects that can be changed and controlled by gesture. For example, tools often use animated graphical effects as they are wielded to suggest how they will affect their environment. The architecture defines how tools interact with other Unidraw objects and how tools implement a wide range of manipulation semantics.

3.5.1 Tool Protocol

The basic tool protocol is shown in Table 6. Conceptually, tools do their work within viewers, in which graphical component views are displayed and manipulated. Whenever a viewer receives an input event (such as a mouse click or key press), it in turn asks the current tool (defined by the enclosing editor object) to produce a manipulator object. A tool implements its CreateManipulator operation to create and initialize an appropriate manipulator, which encapsulates the tool's manipulation semantics by defining the three phases (grasp, wield, effect) of the manipulation. A tool may modify the contents of the current selection object (also defined by the enclosing editor) based on the event. Moreover, a tool can delegate manipulator creation to one or more graphical views (usually among those in the editor's selection object) to allow component-specific interaction. A tool's InterpretManipulator operation accesses and analyzes information in the manipulator that characterizes the manipulation and then creates a command that carries out the desired effect. If a tool delegated manipulator creation to a graphical view, then it must delegate its interpretation to the same view.

return values	*operation*	*arguments*
	Grasp	event
boolean	Manipulating	event
	Effect	event
	{*child iteration and manipulation operations*}	

Table 7: Manipulator protocol

The GetPrototype operation is defined by the **graphical component tool** subclass. Graphical component tools maintain a prototype component and define how that component is created and added to the component hierarchy in the viewer. The prototype is a graphical subject that the tool copies and modifies to conform to the direct manipulation. The tool then inserts the copy into the component hierarchy using an appropriate command. Lastly, tools (like commands) store a reference to the editor that owns them and provide SetEditor and GetEditor operations for assigning and retrieving this reference.

3.5.2 Manipulator Protocol

The manipulator protocol (Table 7) reflects the grasp-wield-effect behavior of tools. The Grasp operation takes an input event and initializes whatever state is needed for the direct manipulation (such as animation objects). During direct manipulation, the Manipulating operation is called repeatedly until the manipulator decides that manipulation has terminated (based on its own termination criteria) and indicates this by returning a false value. The Effect operation gives the manipulator a chance to perform any final actions following the manipulation.

Because some sorts of direct manipulation may require several sub-manipulations to progress simultaneously (for instance, the editor may allow the user to manipulate more than one component at a time), a manipulator may have children. Thus the manipulator protocol also includes operations for iterating through and manipulating its child manipulators.

This simple protocol is sufficient to describe direct manipulations ranging from text entry and rubberbanding effects to simulating real-world dynamics such as imparting momentum to an object. Since manipulators must maintain information that characterizes the final

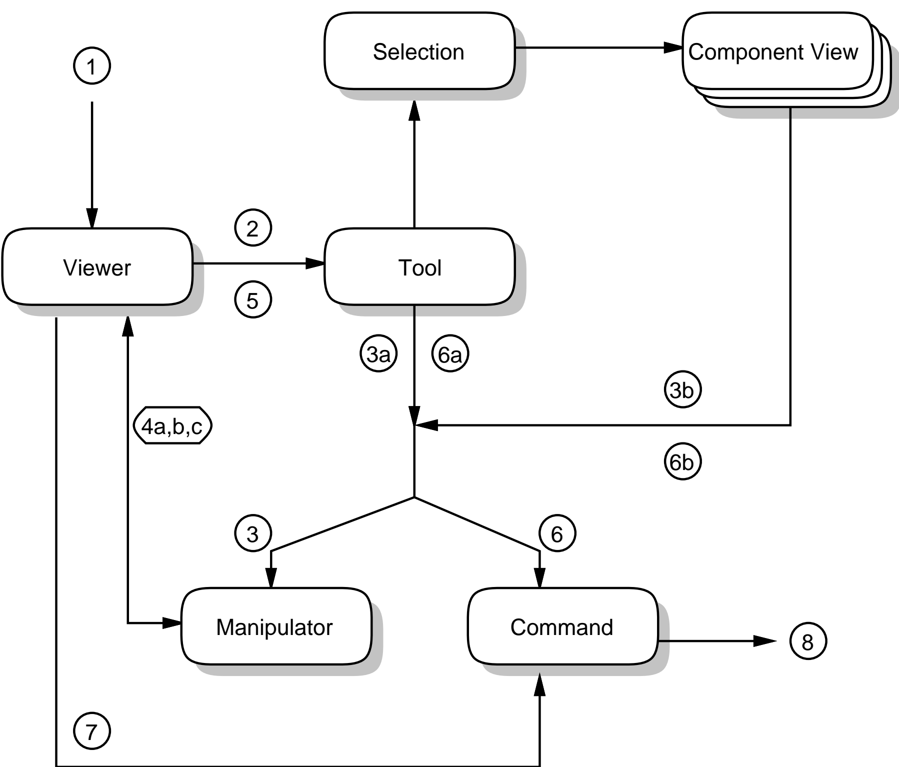


Figure 8: Communication between objects during direct manipulation

outcome of a manipulation, subclasses usually augment the protocol with operations for retrieving state that determines this outcome. For example, a manipulator that supports dragging the mouse to translate a graphical component will define an operation for retrieving the distance moved.

3.5.3 Object Communication during Direct Manipulation

Figure 8 diagrams the communication between objects during direct manipulation to clarify the roles of tools, manipulators, commands, viewer, and graphical views. The numeric labels in the diagram correspond to the transmission sequence:

1. The viewer receives an input event, such as the press of a mouse button.

2. The viewer asks the current tool to CreateManipulator based on the event.

3. *Manipulator creation:* the tool either

    (a) creates a manipulator itself (based on the selection or other information), **or**

    (b) asks the component view(s) to create the manipulator(s) on its behalf. The tool must combine multiple manipulators into a composite manipulator. Each class of component view is responsible for creating an appropriate manipulator for the tool.

4. *Direct manipulation:* the viewer

    (a) invokes Grasp on the manipulator, supplying the initiating event;

    (b) loops, reading subsequent events and sending them to the manipulator in a Manipulating operation (looping continues until a Manipulating operation returns false);

    (c) invokes Effect on the manipulator, supplying the event that terminated the loop.

5. The viewer asks the current tool to InterpretManipulator.

6. *Manipulator interpretation:* the tool either

    (a) interprets the manipulator itself, creating the appropriate command, **or**

    (b) asks the component view(s) to interpret the manipulator(s) on its behalf. The view(s) then create(s) the appropriate command(s). The tool must combine multiple commands into a composite (macro) command.

7. The viewer executes the command.

8. The command carries out the intention of the direct manipulation.

To illustrate this process, consider the following example of direct manipulation in a drawing editor. Suppose the user clicks on an rectangle component view (**RectangleView**) in the drawing area (viewer) with the **MoveTool**. The viewer receives a "mouse-button-went-down" event and asks the current tool (the MoveTool, as provided by the enclosing

editor) to CreateManipulator based on the event. MoveTool's CreateManipulator operation determines from the event which component view was hit and adds it to the selection. More precisely, the selection object provided by the enclosing editor appends the view to its list.

If the selection object contains only one component view, then MoveTool's Create-Manipulator operation calls CreateManipulator on that component view. This gives the component view a chance to create the manipulator it deems appropriate for the MoveTool under the circumstances. Since the user clicked on a RectangleView, the component view will create a **DragManipulator**, a manipulator that implements an downclick-drag-upclick style of manipulation. DragManipulators animate the dragging portion of the manipulation by drawing a particular shape in slightly different ways in each successive call to their Manipulating operation. The definition of DragManipulator parameterizes the shape so that subclasses of DragManipulator are not needed to support dragging different shapes.

Once the viewer obtains the DragManipulator from the MoveTool, the viewer creates the illusion that the user is grasping and wielding the tool. First the viewer calls Grasp on the manipulator, which allows the manipulator to initialize itself and perhaps draw the first frame of the animation. Then the viewer loops, forwarding all subsequent events to the manipulator's Manipulating operation until it returns false. Successive calls to Manipulating produce successive frames of the animation. Once manipulation is complete, the viewer invokes the manipulator's Effect operation, which gives the DragManipulator a chance to finalize the animation and the state it maintains to characterize the manipulation. The viewer then asks the tool to InterpretManipulator; in this case, the MoveTool in turn asks the RectangleView to InterpretManipulator. In response, RectangleView constructs and returns a **MoveCommand**, which specifies a translation transformation. The RectangleView initializes the amount of translation in the MoveCommand to the distance between the initial and final frames of the animation, which it obtains from the DragManipulator.

3.5.4 Tools versus Manipulators

Early in the architecture's design, there was no distinction between tools and manipulators—tools defined their own manipulation semantics. We then found that many tools shared the same semantics, some shared similar ones, some could exhibit different semantics

return values	*operation*	*arguments*
error	Emit	stream
error	Definition	stream

Table 8: External view protocol

at different times, and others had unique (but potentially idiomatic) semantics. Tying distinctive behavior to individual tool subclasses was unsatisfactory because we found no way to factor the classes so that different behaviors could be mixed and matched with different tools.

The solution, of course, was to encapsulate the manipulation semantics in an independent object, the manipulator. The more we experimented with this concept, the more apparent it became that it was a good solution. Manipulators package direct manipulation semantics nicely. Components can define independent manipulation semantics for the same tool, though even divergent semantics for a given tool should reflect the spirit of the tool. The notion of allowing a component to create a specification of its manipulation semantics and later produce a command based on the manipulation's outcome is hard to recast into a model that lacks manipulators. Manipulators also increase the chances for code reuse, because they can be used in many contexts and can be composed easily. For example, the DragManipulator described above can be used by other component-modifying tools in addition to the MoveTool. Thus we found it useful to abstract the mechanics of direct manipulation with tools in a separate manipulator object.

3.6 External Representations

An external representation of a component is simply a non-graphical view of its subject. Domain-specific external representations are derived from the **external view** subclass of component view.

The external view protocol is shown in Table 8. The protocol defines two operations, Emit and Definition, that generate a stream of bytes constituting the external representation. Emit initiates external representation generation and calls Definition recursively. Emit

normally calls the external view's own Definition operation first. Then if the external view contains subviews, Emit must invoke the children's Definition operations in the proper order to ensure a syntactically correct external representation.

Emit is often used to generate header information that appears only once in the external representation, while Definition produces component-specific, context-independent information. For example, a drawing editor may define a **PostScriptView** external view subclass that defines Emit to generate global procedures and definitions. Component-specific subclasses of PostScriptView then need only define Definition to externalize the state of their corresponding component. Thus when Emit is invoked on an instance of any PostScriptView subclass, a stand-alone PostScript representation (known as "encapsulated" PostScript) will be generated. When the same instance is buried in a larger PostScriptView, only its definition will be emitted.

The architecture predefines **preorder**, **inorder**, and **postorder** external views. These subclasses manage subviews and support one of three common traversals of the external view hierarchy.

3.7 Application Framework

Having described components, commands, tools, and external representations in detail, we can now focus on how they function in a domain-specific editor. This section describes the protocols associated with the objects shown in Figure 2, beginning with the viewer object and proceeding upward toward the unidraw object.

3.7.1 Viewer

A viewer displays a graphical component view and provides an interface to scrolling and zooming it. The viewer can also process user input events and translate them to conform to Unidraw protocols. Viewers are therefore implemented in terms of window system or toolkit abstractions; the viewer class might be derived from **Window**, a window system object that can receive events and draw on the screen. The underlying window system or toolkit must allow the viewer to examine all input events that such a window can receive.

return values	*operation*	*arguments*
	Update	
	Handle	event
	SetGraphicalView	graphical view
graphical view	GetGraphicalView	
	SetEditor	editor
editor	GetEditor	
	{*scrolling and zooming operations*}	

Table 9: Viewer protocol

The viewer generates Unidraw requests in response to one or more "interesting" events, usually those characteristic of direct manipulation. For example, a viewer by default may ignore all events until a mouse-button-down event arrives, at which point begins the process outlined in Section 3.5.3 for using a tool. From then on, the viewer forwards all events to the manipulator until manipulation stops.

The viewer protocol is shown in Table 9. Update informs the viewer that some part of the graphical view has changed. The viewer then redraws the graphic object defined by the graphical view. Viewer implementations can use incremental techniques to speed the process of redrawing complicated graphics that change only slightly in the common case. Graphical views should thus delegate redrawing themselves to the viewer, which can often optimize away many individual redraws.

The Handle operation provides an interface to the corresponding window system or toolkit mechanism for receiving input events. Handle is responsible for converting these events into Unidraw protocol requests. The SetGraphicalView and GetGraphicalView operations set and get the graphical view the viewer displays. SetEditor and GetEditor set and get the editor object that owns the viewer. Finally, the viewer protocol should provide window system or toolkit-compatible operations for scrolling and zooming the graphical view.

return values	*operation*	*arguments*
	Update	
	Open	
	Close	
state variable	GetStateVariable	name
	SetComponent	component subject
component subject	GetComponent	
	SetViewer	viewer, int
viewer	GetViewer	int
	SetCurrentTool	tool
tool	GetCurrentTool	
	SetSelection	selection
selection	GetSelection	

Table 10: Editor object protocol

3.7.2 Editor

An editor provides a complete user interface to editing a graphical component subject. It unites one or more viewers with the commands and tools that act upon the component and its subcomponents. Editors are normally derived from toolkit objects that can compose buttons, menus, and the like into a finished user interface. Each window of a domain-specific editor is usually an instance of an editor subclass designed for the domain.

Table 10 depicts the editor object protocol. The Update operation simply invokes the corresponding operation on the editor's viewers. Open informs the editor that it has just become visible and accessible to the user, in case it needs to know, and Close signals that it is no longer needed and should perform any final housekeeping operations. For example, the editor may display a copyright message when it is first opened, or it may take the opportunity when closed to warn the user to save a modified component.

The protocol also defines several operations for setting and retrieving its characteristic state. GetStateVariable provides a standard interface to accessing editor state variables, which may store, for example, the name of the component being edited, the current magnification of a viewer, or the font with which new text components appear. SetComponent and GetComponent respectively set and get the component subject that the user edits with the

return values	*operation*	*arguments*
	{*graphical view iteration and manipulation operations*} Show Hide Clear	

Table 11: Selection protocol

editor. SetViewer and GetViewer set and get the viewer(s) owned by the editor. Multiple viewers are identified serially, and each contains a view of the editor's graphical component subject. The SetCurrentTool and GetCurrentTool operations involve the editor's notion of the current tool, that is, the tool that the user will wield should he initiate direct manipulation. Similarly, the SetSelection and GetSelection operations set and get the selection object used by the editor. How the current tool and selection operations are implemented depends on the particular interface: if the application defines a single, global selection, then all editors should share the same selection object; if each editor provides its own palette of tools, then each one must store the current tool for itself.

3.7.3 Selection

A selection object is nominally a convenient interface to managing a set of distinguished graphical component views. Its protocol is shown in Table 11. Each selection object maintains a list of graphical views and provides operations for iterating through and manipulating the list. Show invokes the Highlight operation on each view (see Table 3), while Hide invokes Unhighlight. The Clear operation is equivalent to calling Hide and then removing all the views from the list.

This protocol provides bare-minimum functionality for handling selected graphical views; convenient operations can be built on top. For example, the order in which views appear in the list usually reflects the order in which the user selected them with a tool. This order is probably not the order in which the views occur in their parent view. If the user cuts and pastes the components, they must retain their original relative ordering. A Sort

return values	*operation*	*arguments*
	Register	component subject, name
	Unregister	component subject
component subject	GetComponent	name
name	GetName	component subject
	Register	command, name
	Unregister	command
command	GetCommand	name
name	GetName	command
	Register	tool, name
	Unregister	tool
tool	GetTool	name
name	GetName	tool

Table 12: Catalog protocol

operation that orders the views according to their position in their parent view would make cut-and-paste editing commands easier to implement.

3.7.4 Catalog

The structure of Unidraw objects can be quite complex. For example, a component can have several views and can contain other components with their own views. Since Unidraw objects must live beyond a particular editing session, some provision must be made for maintaining these objects in non-volatile storage. The potential complexity of these objects makes it difficult for a programmer to manage their storage manually. The catalog abstraction is designed to free the programmer from this responsibility by letting him store and retrieve only those objects in which he is explicitly interested; other objects are managed automatically by the system.

A catalog provides independent name-to-object mappings for component subjects, commands, and tools. Table 12 depicts the catalog protocol. Name mappings are defined and undefined with the Register and Unregister operations, respectively, for each of the three base classes. The GetComponent, GetCommand, and GetTool operations take a name as

return values	*operation*	*arguments*
	Run	
	Open	editor
	Close	editor
	Update	
	Quit	
	Log	command
	Undo	int
	Redo	int
	SetHistoryLength	int
int	GetHistoryLength	
	SetCatalog	catalog
catalog	GetCatalog	

Table 13: Unidraw object protocol

an argument and return the corresponding object, if the name has been mapped to one. The GetName operations carry out the reverse mappings.

These catalog semantics imply that an object stored in the catalog must survive across activations of the domain-specific application. Note that the protocol does not mandate a flat name space; implementations can extend the protocol as needed to support name spaces of unrestricted structure.

3.7.5 Unidraw

A domain-specific editor application must create exactly one instance of a unidraw object, which implements the main loop of the program, makes editors appear and disappear on the screen, maintains a log of all commands that have been (un)executed and (un)interpreted, and stores a reference to the (one and only) catalog. The unidraw object obeys the protocol shown in Table 13.

The Run operation defines the application's main loop. Run's implementation is generally window system/toolkit-specific; it may read events explicitly and forward them to their target objects, or it may simply encapsulate the equivalent operation defined by the underlying system. Run might also convert raw events into Unidraw protocol requests itself

if convenient or necessary. For example, Run might execute a **QuitCommand** or invoke the Quit operation (described below) directly if a special key is pressed. Open makes the given editor appear on the display, carrying out any system-specific initialization, while Close removes the editor from the user's viewpoint. Open and Close call the corresponding operations on the editor. Update synchronizes the application and the state of its constituent objects: connectors move to satisfy their connection semantics, and viewers redraw parts of their graphical views that have changed. Unidraw objects may experience many internal state changes before the corresponding effects are made visible by Update, which is usually issued as the last protocol request in user-accessible commands. The application is terminated by invoking the Quit operation. Quit performs whatever housekeeping is necessary before the program exits.

The architecture supports arbitrary-level undo and redo semantics through the Log, Undo, and Redo operations defined on the unidraw object. Log places a command on a list of previously-executed commands. We refer to this list as the **past log**, or **past** for short, which represents the history of undoable operations. Log insures that only reversible commands are logged. Commands should be logged wherever they are executed, preferably immediately thereafter; however, commands are not responsible for calling Log themselves within their (Un)execute or (Un)interpret operation—in fact, doing so will log the command twice.

The Undo operation reverse-executes the specified number of past commands, removing them from the past and placing them on another list of commands, the **future log**, or simply **future**. Redo performs the reverse operation: it re-executes the specified number of commands from the future and moves them to the past. Note that Undo and Redo assume that past and future commands are either executable or unexecutable; there is no way for these operations to arrange to have the command interpreted or uninterpreted by the proper components. However, this is not a problem if interpretive commands define their (Un)execute operation to call (Un)interpret on their stored components, as suggested in Section 3.4.

This model directly supports both history undo/undo and linear undo/redo as described by Vitter [55]. In history undo/undo, undo itself is an undoable operation; undoing it corresponds to a redo, and no explicit redo is required. This model can be implemented

with a reversible undo command. Linear undo/redo specifies that undo and redo are “meta-operations” that are not included in the command history, which lists only “primitive” commands. Undo and redo simply manipulate the history of primitive commands, as suggested by the corresponding unidraw object operations. Linear undo/redo can be implemented by defining irreversible undo and redo commands. A domain-specific application can extend the unidraw object’s undo model via subclassing to support more sophisticated semantics.

The last four operations defined by the unidraw object protocol set and get two of its attributes. SetHistoryLength specifies the past log’s (and, consequently, the future log’s) maximum length, from zero to infinity; GetHistoryLength returns this attribute’s current value. Finally, SetCatalog and GetCatalog set and get the catalog appropriate for the application. SetCatalog should be called once by the unidraw object when the application creates it.

3.8 Summary

The Unidraw architecture defines object-oriented abstractions that simplify the construction of graphical object editors. Unidraw spans the gap between general-purpose toolkits and domain-specific editors by providing functionality that is common across domains but not supported by existing systems. This functionality is partitioned into four basic classes: components encapsulate the appearance and semantics of objects in a domain, tools support direct manipulation of components, commands define operations on components and other objects, and external representations specify the mapping between components and the file format generated by the editor. Domain-specific editors use specializations of these objects to suit their needs.

In addition to these basic classes, the architecture provides abstractions that aid in implementing domain-specific semantics. Editors for many domains require that components remain connected to one another, both visually and in an information-transfer sense. The connector component subclass supports various connectivity and confinement semantics. Connectors also support dataflow between components when used in concert with state variables and transfer functions. Tools use manipulators to define their behavior during

direct manipulation and commands to define their effects afterwards, while commands use clipboards to store enough information to reverse their effects.

Lastly, the architecture specifies a framework for assembling the basic objects into a complete application. A viewer acts as a gateway through which the user manipulates a component and its subcomponents with tools, converting raw window system or toolkit events into Unidraw protocols. An editor combines viewers and toolkit objects for engaging commands and tools into a self-contained interface that the user perceives to be the domain-specific application. An editor uses a selection object, which stores a list of distinguished components, to manage components in which the user has expressed interest. The application uses a single catalog object to keep track of components (both predefined and user-defined) and commands and tools (both those already incorporated into the application and those that are available for incorporation). The application also creates and maintains a single unidraw object that acts as prime mover and global coordinator of the application.

While a detailed architectural specification is an important part of achieving the goals of this thesis, the only way to determine whether they truly have been achieved is to build an implementation from the specification and use it to develop real domain-specific editors. These undertakings are the subjects of the next two chapters.

Chapter 4

A Prototype Implementation

We developed a prototype Unidraw library to test the viability of the architecture. Subsequently, we used the prototype to build three domain-specific editors; that effort is described in the next chapter.

In this chapter we focus on the salient aspects of our Unidraw implementation. We do not present a detailed description of every object; instead, we consider the overall structure of the system, the subsystems on which it is built, key objects, relevant algorithms, and novel techniques we applied. We hope this discussion proves useful not only to Unidraw implementors but to user interface developers in general.

First we describe briefly the object-oriented language we chose and the user interface toolkit that we developed and used with the prototype. Later we introduce the fundamentals of the prototype, explaining its organization in terms of the major class hierarchies. Then we consider the implementation of components, arguably the most complex objects in Unidraw. The special requirements of connector components are examined next (how connectivity is maintained and how dataflow works), followed by a section on the command and tool implementations. Finally, we discuss the implementation of catalog semantics and close with a summary of the chapter.

4.1 C++

Our experience with user interface development has underscored the importance of using an object-oriented rather than procedural language to express the implementation of an object-oriented architecture. Simulating inheritance, dynamic method binding, and data abstraction in a procedural language is a burden for programmers, forcing them to do book-keeping that is far removed from the architectural concepts being implemented. A language that reflects object-oriented concepts explicitly narrows the gap between architecture and implementation, freeing the programmer to concentrate on complexity at higher levels. We have found that code produced by a skilled programmer in an object-oriented language is more readable, maintainable, and extensible than a functionally-equivalent procedural implementation.

In mid-1986, after our research group had built several object-oriented systems in Modula-2, we switched to C++ [50], a C-based object-oriented language developed at AT&T Bell Laboratories. C++ retains C syntax and adds Simula-style classes with single and multiple inheritance, optional run-time method lookup, and full static type checking, plus a host of ancillary features such as operator overloading and reference parameters. C++ is an attractive language because it supports object-oriented programming without compromising the execution efficiency of C. C++ compilers generate conventional object files that are linked into stand-alone applications. While C++ lacks useful features such as run-time access to class information, untyped message passing, and garbage collection, its explicit object-oriented semantics greatly simplifies the implementation of object-oriented systems compared to procedural languages, and its efficiency was highly desirable for the purposes of our research.

4.2 InterViews

The prototype does not include toolkit functionality because a toolkit was available from the start. Originally to support our group's programming environment research, we developed InterViews [23], an object-oriented toolkit implemented in C++. InterViews provides a library of predefined objects and a set of protocols for composing them. A user interface

is created by composing simple primitives in a hierarchical fashion, allowing complex user interfaces to be implemented easily.

InterViews supports composition of two categories of object. Each category is implemented as a hierarchy of object classes derived from a common base class. Composition subclasses within each class hierarchy allow hierarchical composition of object instances.

1. Interactive objects such as buttons and menus are derived from the **interactor** base class. Interactors are composed by **scenes**. Scene subclasses define specific composition semantics such as tiling or overlapping.

2. Structured graphics objects such as circles and polygons are derived from the **graphic** base class. Graphic objects are composed by **pictures**, which provide a common coordinate system and graphical context for their components.

Domain-specific editors use objects from these hierarchies to implement their basic user interface, and they use Unidraw objects to support graphical object editing. The Unidraw prototype uses InterViews structured graphics objects extensively; these are described in more detail in the next section. Several other InterViews objects are used in the prototype and experimental applications; these are described where they are introduced in the remainder of the thesis.

4.3 Structured Graphics

In addition to providing primitive and composition objects for building the controlling elements of a user interface, InterViews includes abstractions for creating and manipulating graphical objects. **Structured graphics** refers to a graphics model in which geometric primitives such as circles and polygons can be assembled into hierarchies. Each primitive has some associated state, such as its geometric and coordinate system specification and rendering information. Primitives can be composed hierarchically to impose structure on the graphics being displayed. In contrast, the **immediate mode graphics** model does not associate state with graphics; shapes are drawn by calling procedures that simply modify display memory. The shapes have no storage associated with them and cannot be accessed or

manipulated after they are drawn. Most window systems and toolkits (including InterViews) rely on immediate mode graphics at some level to produce graphical output.

Structured graphics can simplify the implementation of graphical object editors because structured graphics packages implement much of the basic graphics capabilities such editors require. Programmers can use structured graphics to represent the data on which a graphical editor operates. For example, drawing editor operations for translating and scaling geometric shapes, enlarging and reducing the drawing, and storing and later reconstructing its representation on disk are supported by most structured graphics packages. Graphical hierarchies could be used to compose and manipulate groups of notes on staves in a music editor. A project management system could define the elements of bubble charts using graphical primitives and allow the user to make structural changes interactively by calling operations for editing the hierarchy. Moreover, structured graphics packages usually provide operations for detecting when a user selects an element with the mouse and for redrawing the image efficiently following modification.

However, there are drawbacks to using traditional structured graphics packages [18, 21, 44]. The library of procedures they provide is typically large and monolithic, rich in functionality but difficult for the programmer to extend. Extensibility usually requires access to and manipulation of internal data structures, which can compromise the reliability of the system. Also, it is often difficult to edit and manipulate the graphical structure, particularly when its elements are represented procedurally, because there is no way to refer to graphic and geometric attributes directly. Editing the graphical structure may be inefficient as well. For example, if the structure is compiled into a more compact or efficient form, then recompilation is required before a modified drawing can be rendered. These deficiencies make it likely that the structure provided by the package will not map well to that required by the application, forcing the programmer to define data structures and procedures that parallel the library's.

We addressed these problems in InterViews with an object-oriented approach to structured graphics. InterViews partitions structured graphics functionality into a collection of classes that correspond to individual graphical primitives. The graphic base class and derived classes collectively form the InterViews Graphic library. The prototype Unidraw implementation relies quite heavily on this library to facilitate building domain-specific

Graphic	Label Line Picture Point RasterRect Stencil		
	Rect	S_Rect F_Rect SF_Rect	
	Ellipse	S_Ellipse	S_Circle
		F_Ellipse	F_Circle
		SF_Ellipse	SF_Circle
	Vertices	MultiLine	S_MultiLine SF_MultiLine
		Polygon	S_Polygon F_Polygon SF_Polygon
		OpenBSpline	S_OpenBSpline F_OpenBSpline SF_OpenBSpline
		ClosedBSpline	S_ClosedBSpline F_ClosedBSpline SF_ClosedBSpline

Table 14: Graphic library class hierarchy

editors. The following sections describe the Graphic library and the advantages of its object-oriented approach over those of traditional structured graphics packages. A more detailed description and analysis of the library appears in an earlier report [57].

4.3.1 Class Organization

Table 14 lists the classes in the Graphic library according to the inheritance hierarchy. The hierarchy's design was guided by the desire to share code as much as possible without

compromising the logical relationships between the classes. The derived classes define the following graphical objects:

- **Point**, **Line**, **MultiLine**: point and line objects
- **Label**: a string of text
- **Stencil**: a bitmapped, monochrome image
- **RasterRect**: a bitmapped, color image
- **Picture**: a collection of graphics
- **S_Rect**, **F_Rect**, **SF_Rect**: stroked, filled, and stroked-filled rectangles
- **S_Ellipse**, **F_Ellipse**, **SF_Ellipse**: stroked, filled, and stroked-filled ellipses
- **S_Circle**, **F_Circle**, **SF_Circle**: stroked, filled, and stroked-filled circles
- **S_MultiLine**, **SF_MultiLine**: stroked and stroked-filled connected line segments
- **S_Polygon**, **F_Polygon**, **SF_Polygon**: stroked, filled, and stroked-filled polygons
- **S_OpenBSpline**, **F_OpenBSpline**, **SF_OpenBSpline**: stroked, filled, and stroked-filled open B-splines
- **S_ClosedBSpline**, **F_ClosedBSpline**, **SF_ClosedBSpline**: stroked, filled, and stroked-filled closed B-splines

All graphics maintain graphics state and geometry information. Graphics state parameters are defined in separate base classes, including **transformer** (transformation matrix), **color**, **pattern** (for stippled area fills), **brush** (for line drawing), and **font**. Each graphics state class implements operations for defining and modifying its attributes. For example, transformers have translation, scaling, rotation, and matrix multiplication operations, and colors provide an interface for varying their component intensities.

The graphic base class contains a minimal set of graphics state that includes a transformer and foreground/background colors. Derived classes maintain additional graphics state

according to their individual requirements. For example, the label class includes a font in addition to inherited state, filled objects maintain a pattern, and outline objects include a brush.

All graphics implement a set of operations defined in the base class. These include operations for

- drawing and erasing, optionally clipped to a rectangle,
- setting and retrieving graphics state values,
- translating, scaling, and rotating,
- obtaining a bounding box, and
- ascertaining whether the graphic contains a point or intersects a rectangle.

4.3.2 Graphics State Concatenation

A picture is a composite graphic that composes other graphics into a single object. Composite graphics are like other graphics in that they maintain their own graphics state information, but they do not have their own geometric information. Composition allows us to define how the composite's state information affects its components. The graphic base class implements a mechanism for combining, or **concatenating**, graphics state information. The default behavior for concatenation is described below. Composite graphics can redefine the concatenation operations as needed.

Given two graphics states A and B, we can write their concatenation as $A \oplus B = C$, where C is the resultant graphics state. Concatenation associates but is not commutative; B is considered **dominant**. C receives attributes defined by B. Attributes that B does not define are obtained from A. An exception is the transformation matrix: C's transformer is defined by postmultiplying A's transformer by B's. B thus dominates A in that C inherits B's attributes over A's, and C's coordinate system is defined by A's transformation with respect to B's.

A graphic might not define a particular attribute either because it is not meaningful for the graphic to do so (a filled rectangle does not maintain a font, for instance) or because

the attribute's value has been set to nil deliberately. Defined attributes propagate through successive concatenations without being overridden or modified by undefined attributes. For example, suppose graphics state A defines a font but B does not. Moreover, C maintains a font but its value has been set to nil. Then $D = A \oplus B \oplus C$ will receive A's font attribute. If A's transformer is nil but B and C's are non-nil, then D will receive a transformer that is the product of B's and C's. If $D = C \oplus A \oplus B$, then D will receive a transformer that is the product of C's and B's.

Composite graphics use graphics state concatenation to define their appearance. Pictures are the basic mechanism for building hierarchies of graphics. Each picture maintains a list of component graphics. A picture draws itself by drawing each component with a graphics state formed by concatenating the component's state with its own. Thus, operations on a picture affect all of its components as a unit.

The semantics for concatenation as defined in the base class are useful for describing how composite graphics are drawn, but derived graphics can implement their own concatenation mechanism. This creates the potential for concatenation semantics that are more powerful than the default precedence relationship. For example, the concatenation operation could be redefined so that concatenating two colors would yield a third that is the sum or difference of the two. Two patterns could combine to form a pattern corresponding to an overlay of the two. This behavior could be used to define how to draw overlapping parts of a VLSI layout.

The ability to redefine concatenation semantics demonstrates how inheritance lets the programmer extend the graphics library easily. Flexibility is thus achieved without complicating or changing the library.

4.3.3 Incremental Update

Structured graphics can be used to represent and draw arbitrarily complicated images. Many images (and most interesting ones) cannot be drawn instantaneously. Incremental techniques can speed the process of keeping the screen image consistent with the underlying graphical structure. Such techniques will be effective if the user makes small changes most of the time, and experience with interactive graphics editors shows this to be the case.

```
void Incur(Graphic*);
void Incur(BoxObj&);
void Incur(Coord, Coord, Coord, Coord);
void Repair();
void Reset();
boolean Incurred();
```

Figure 9: Interface to damage class

The Graphic library includes a **damage** base class to support incremental update. A damage object is used to keep the appearance of graphics consistent with their representation. Damage objects try to minimize the work required to redraw corrupted parts of a graphic. The base class implements a simple incremental algorithm that is effective for many applications. The algorithm can be replaced with a more sophisticated one by deriving from the base class.

The C++ interface to the damage class appears in Figure 9. When a damage object is created it is passed a graphic (usually a picture) for which it is responsible. The Incur operation is called by the client program whenever the graphic is **damaged**. The programmer must supply the damaged region to the Incur call, either by passing the graphic that contributed the region or by specifying a rectangular area explicitly. The graphic is incrementally updated when Repair is called. Reset discards accumulated damage without updating the graphic. Clients can determine whether any damage has been incurred using the Incurred operation.

4.4 Overview of the Unidraw Prototype

The Unidraw prototype is a library of C++ classes built on top of InterViews and the X Window System [41]. Table 15 presents a breakdown of the prototype implementation in classes and lines of source code.

		code (lines)	
	classes	*interface*	*implem.*
components	38	1090	4810
commands	42	900	2390
tools/manipulators	16	500	1510
ext. representations	18	300	1150
state vars./transf. fns.	20	410	1000
applic. framework	5	460	2450
creator	1	30	170
toolkit-derived classes	22	390	1260
utilities/globals	18	940	3260
totals	180	5020	18000

Table 15: Unidraw prototype library code breakdown

The prototype defines C++ classes corresponding to each of the major architectural classes. **Component** and **ComponentView** base classes provide the protocols for component subjects and views, respectively, while **GraphicComp** and **GraphicView** subclasses furnish the additional protocol required by graphical components. **ExternView** is a subclass of ComponentView that defines the external representation protocol, from which **PreorderView**, **InorderView**, and **PostorderView** classes are derived. **Connector** and **ConnectorView** classes and corresponding pin, slot, and pad subclasses implement connector semantics. **Command** and **Tool** base classes embody the command and tool protocols. Also included are **Catalog**, **Clipboard**, **Editor**, **Manipulator**, **Selection**, **StateVar**, **StateVarView**, **TransferFunct**, **Unidraw**, and **Viewer** classes.

To these elementary classes the prototype adds many predefined subclasses, shown in Tables 16 and 17, that programmers can use immediately to build domain-specific editors. These subclasses include basic graphical components (such as lines and polygons), their views and PostScript external representations; commands for manipulating components and changing their attributes; commands that let the user access the catalog; tools for selecting, transforming, and otherwise modifying graphical components; manipulators

Component	GraphicComp	GraphicComps EllipseComp LineComp LinkComp RectComp TextComp			
		VerticesComp	SplineComp ClosedSplineComp MultiLineComp PolygonComp		
		Connector	PinComp PadComp		
			SlotComp	HSlotComp VSlotComp	
ComponentView	GraphicView	GraphicViews EllipseView LineView LinkView RectView TextView			
		VerticesView	SplineView ClosedSplineView MultiLineView PolygonView		
		ConnectorView	PinView PadView		
			SlotView	HSlotView VSlotView	
	ExternView	InorderView PostorderView			
		PreorderView	PostScriptView	PostScriptViews PSEllipse PSLine PSLink PSRect PSText PSPin PSSlot PSPad	
				PSVertices	PSSpline PSClosedSpline PSMultiLine PSPolygon

Table 16: Predefined component subclasses

Command	AlignCmd BackCmd BrushCmd CenterCmd CloseEditorCmd ColorCmd ConnectCmd CopyCmd CutCmd DeleteCmd DupCmd FontCmd GravityCmd GridCmd GridSpacingCmd MacroCmd MobilityCmd NewCompCmd NormSizeCmd OrientationCmd PasteCmd PatternCmd PrintCmd QuitCmd RedToFitCmd RedoCmd ReplaceCmd RevertCmd RotateCmd SaveCompAsCmd SaveCmd ScaleCmd SlctAllCmd UndoCmd ViewCompCmd	
	StructCmd	GroupCmd UngroupCmd
Tool	ConnectTool GraphicCompTool MagnifyTool MoveTool ReshapeTool RotateTool ScaleTool SelectTool StretchTool	
StateVar	BrushVar ColorVar FontVar GravityVar MagnifVar ModifStatusVar NameVar PatternVar	
StateVarView	BrushVarView ColorVarView FontVarView GravityVarView MagnifVarView ModifStatusVarView NameVarView PatternVarView	
TransferFunct	TF_2Port	TF_Direct
Manipulator	ManipGroup TextManip	
	DragManip	ConnectManip VertexManip

Table 17: Additional predefined subclasses

supporting common direct manipulation such as "downclick-drag-upclick" interactions and text editing; state variables for basic graphical component attributes (such as color and fill pattern) together with sample views; and transfer functions that support simple dependencies between state variables.

4.5 Components

The implementation of the component subject and view base classes is straightforward. The subject maintains a list of views that have been attached; it calls Update on each of these views when Notify is called. The view keeps a pointer to its subject. The base classes do not maintain local state apart from these attributes; a component subclass that defines state variables and transfer functions, for example, must allocate them and redefine the corresponding access functions to return them.

One complication stems from C++ in that it disallows sending arbitrary messages to objects. Messages are sent via strongly-typed procedure calls, so a class must declare all acceptable messages at compile-time. As a result, component operations such as Interpret and Uninterpret cannot be implemented by accepting untyped messages from commands. In lieu of this capability, components must query the command to determine its class, but C++ cannot provide this information at run-time. Our implementation defines an IsA operation based on programmer-managed class identifiers to solve this problem, but ideally the language would provide either run-time class resolution or untyped (or dynamically-typed) method lookup.

Another subtlety concerns how to create an appropriate view given a subject. For example, a command responsible for generating an external representation for a particular graphical subject must create the corresponding external view of that subject; a command that produces a graphical view of a component must instantiate the proper graphical view, if it exists. This implies that component views should be categorized and that a mapping is required between a component subject and its view for a given category.

The prototype predefines **COMPONENT_VIEW** and **POSTSCRIPT_VIEW** view categories and extends the component protocol to include a Create operation, which takes the view category as an argument and returns a component view. Calling Create on a

graphical subject with COMPONENT_VIEW as its argument yields the corresponding graphical view, while doing so with POSTSCRIPT_VIEW produces the corresponding PostScript external view. For example, if `pinComp` is a pin component, then

```
Create(pinComp, COMPONENT_VIEW)
```

returns an instance of **PinView**. The mapping between subjects, views, and view categories is defined by the **Creator** object, discussed in Section 4.9. Additional categories are supported by deriving domain-specific creators.

4.5.1 Composition

The architecture encourages programmers to build complex components from simpler ones. To be practical, composite components must work nearly as efficiently as components built from scratch. One of the most common composite components is a hierarchy of graphical components, often representing graphical elements that are treated as a unit. In this case, the leaves of the hierarchy are primitive graphical components such as lines, ellipses, and polygons.

Assume these primitives respond to commands for graphical transformations such as scaling and rotation. The naive approach to transforming the composite component would have the root component interpret, say, a command to scale itself about its center by some amount. The root would then issue this command to each of its children for interpretation, the children would issue it to their children, and so on until the leaf components interpret the command. Such an approach incurs the significant overhead of command interpretation at each node in the hierarchy: each component checks the command against those it is prepared to interpret. Thus, basic graphical operations will take time proportional to the size of the composition multiplied by the average number of commands each component can interpret.

To speed this common case, the base class for composite graphical subjects, **Graphic-Comps**, relies on the graphics state concatenation mechanism described in Section 4.3 to propagate graphics state throughout graphical component hierarchies. Each GraphicComps instance stores a picture into which each child's graphic is inserted. When the Graphic-Comps object interprets a graphical transformation or attribute-modifying command and

applies it to its picture, it will affect the children's graphics automatically via the concatenation mechanism. An added benefit of this approach stems from limiting changes to a single subject, namely the root of the hierarchy. Had each child subject interpreted the command, each would in turn notify its view(s). Instead, only the views associated with the root are notified, and the same optimization is used to propagate graphics state through the view hierarchy. Thus the concatenation mechanism takes care of propagation in the views as well.

This optimization assumes that graphical components will not interpret graphics-oriented commands in special ways. Graphical components that must do so can define new Graphic subclasses with special concatenation semantics; otherwise, the programmer must employ the naive approach described earlier, or he might consider avoiding composition altogether. This raises an important question: When should components be defined in terms of other components, and when should they be built from scratch? Maximum flexibility is realized by composing existing components into new ones, and, all else being equal, implementors should strive to define most domain-specific components through composition. Only when efficiency or other pragmatic considerations are an issue should one try to implement a significant number of components from scratch. Doing so often limits how extensible the application is from the user's perspective, and it usually complicates the implementation.

4.5.2 View Consistency

A component view must reconcile its internal state with its subject's when its Update operation is called. This is usually trivial for leaf components, which ordinarily reconcile only a limited number of attributes, but components with children must be prepared to restructure themselves to conform to their subject's child structure. To accomplish this, the view could assume that all its children are inconsistent with its subject's children and just rebuild them from scratch based on the subject's structure. That approach is simple but potentially expensive. Moreover, the subject's structure usually stays the same or changes only slightly, so an incremental approach in which the view reuses most of its children is preferable. Our implementation supports the common case where the view's child structure

UpdateViewStructure(s, v) {
$s \equiv$ component subject;
$v \equiv$ component view;
$\{s_1, s_2, \ldots, s_m\} \equiv C_s \equiv$ set of children of s;
$\{v_1, v_2, \ldots, v_n\} \equiv C_v \equiv$ set of original children of v;
$L \equiv$ list of triples (s_h, v_i, j), $1 \leq h \leq m, 1 \leq i \leq m, 1 \leq j \leq m$;
$L_k \equiv k$th triple in L;

Initialize(s, v, L);
Rearrange(s, v, L);
Damage(s, v, L);
}

Figure 10: View structure update algorithm

is identical to the subject's. Components with differing subject and view structures must implement their own update algorithm.

The algorithm consists of three routines, as shown in Figure 10: an initialization routine, a view rearrangement routine, and a damage routine.

Initialization and View Rearrangement

Figure 11 details the Initialize and Rearrange routines. Initialize discards any child views that no longer have a corresponding child subject in the parent subject, and it creates views for child subjects that have no child views in the parent view. The graphics of newly added or deleted views are damaged in the process. This routine also initializes a list of triples, each associating a child subject with its view in the parent and the position of that view. This information is used in the remaining routines to expedite view rearrangement and to calculate damage information.

```
Initialize(s, v, L) {
  s, v, L, C_s, C_v, n, h, i ≡ (as above);
  1 ≤ k ≤ n;

  for each (v_k) {
    if (Subject(v_k) ∉ C_s) discard v_k, damaging its graphic;
    else {
      h ← index of Subject(v_k);
      L_h ← (s_h, v_k, k);
    }
  }
  for each (s_h) {
    L_h ↔ (s_h, v_k, k);

    if (v_k ∉ C_v) {
      create new view of s_h, making it thełast child v_i of v and damaging its graphic;
      L_h ← (s_h, v_i, i);
    }
  }
}

Rearrange(s, v, L) {
  s, v, L, h, i, j ≡ (as above);

  for each (s_h) {
    L_h ↔ (s_h, v_i, j);

    if (v_h ≠ v_i) move v_i from position i to position h;
  }
}
```

Figure 11: Initialization and view rearrangement

Damage(s, v, L) {
  $s, v, L, m, h, i \equiv$ (as above);
  $0 \le i_{min} \le m + 1$; $i_{min} \leftarrow 0$;
  $0 \le i_{max} \le m + 1$; $i_{max} \leftarrow m + 1$;
  $D_{fwd}, D_{bwd} \equiv$ sets of views, initially empty;

  for each (s_h) {
    $L_h \leftrightarrow (s_h, v_h, i)$;

    if ($i < i_{min}$) add v_h to D_{fwd};
    else $i_{min} \leftarrow i$;

    if ($m - i + 1 > i_{max}$) add v_h to D_{bwd};
    else $i_{min} \leftarrow m - i + 1$;
  }
  if ($D_{fwd} < D_{bwd}$) damage graphics from views in D_{fwd};
  else damage graphics from views in D_{bwd};
}

Figure 12: Damage incursion

Rearrange simply iterates through the child subjects and views in tandem, comparing each subject with the corresponding view's subject. If there is a mismatch, Rearrange finds the proper view for the subject in the list of triples and moves it to the current view position.

Damage Incursion

The Damage routine, shown in Figure 12, determines a set of graphics to damage so that the parent view's appearance will reflect any restructuring. We assume in this algorithm that minimizing the number of damaged graphics maximizes efficiency. While this assumption is not always valid (it may be cheaper, for instance, to damage five primitive graphics rather than one complicated picture), it is reasonable for the common case where siblings in a graphic hierarchy have similar complexities.

Each child view contributes one graphic. To incur the minimum damage, we first find a maximum-size set of views whose relative order has stayed the same. We need not damage the graphics from views in this set because restructuring did not change the order in which they are drawn; graphics in the set that obscured others still obscure them, and graphics that were visible are still visible. However, the remaining graphics are out of order with respect to the final ordering; their graphics must be damaged so that they (and anything they can affect) will be drawn correctly.

The Damage routine scans through the child views in forward and reverse order, looking up each one's original position. As it scans, it records the largest original position encountered so far in the forward direction and the smallest in the reverse direction. Any views whose original position is less that the largest in the forward direction are out of order with respect to those already encountered; these views are added to a set of views to damage. Similarly, any views whose original position is greater than the smallest in the reverse direction are out of order with respect to those already encountered; these are added to another set of views to damage. Once these sets have been accumulated, the algorithm damages the graphics from the views in the smaller set.

To incur damage, the view follows the chain of ancestor views up to the root view, which is being displayed in a viewer (otherwise the view is not visible and we need not damage anything). Each viewer maintains a damage object and places the root view in a special graphical view called a **ViewerView**, which stores a pointer to the viewer that contains it. The graphical view base class defines a GetViewer operation that the ViewerView class redefines to return its viewer (by default, GetViewer calls itself on the view's parent). Thus the interior view retrieves the viewer with its GetViewer operation; it then incurs damage on the viewer's damage object. The viewer repairs the accumulated damage by calling Repair on the damage object in its Update operation.

The Damage routine is suboptimal in that it does not always find a maximum-sized set of ordered views; it merely determines a set that is maximum-sized in the common case. Most structural editing operations involve moving some of a component's children ahead of or behind their siblings (the bring-to-front and send-to-back operations). In these cases, either the forward or the backward scan will identify exactly those views as having changed position. The algorithm falls down when relatively few views are out of order, and they are

encountered early in *both* forward and backward directions. The worst case occurs when the first and last views are swapped with respect to their initial order, and the remaining views have not changed order. In this case, both scans will encounter an out-of-order view immediately after the first and will damage all other views' graphics, when all that should have been damaged were the swapped views' graphics. Fortunately, few editors require a restructuring capability that will give rise to such a case.

The advantages of this damage incursion algorithm are its simplicity and an $\mathcal{O}(n)$ running time in the number of views being updated.[1] The search for triples uses a hash table and thus takes constant time on average. Since UpdateViewStructure does a constant number of linear scans and hashed lookups, its running time is linear in the average case.[2]

4.6 Connectivity

Connectivity semantics are enforced by a **CSolver** object that manages **connection networks**, or disjoint sets of connections. A connection is defined by two connectors and a piece of connector glue. The connector glue characterizes the relationship between the connectors' centers in conformance with their connectivity semantics. The three standard connector subclasses define a limited number of connection semantics; each semantics is modeled with connector glue of specific natural size, elasticity, and deformation limits.

The csolver is responsible for solving connection networks that have been perturbed, meaning it must position the connectors to satisfy all connection semantics. The csolver stores each connection network as a list of connections. It solves a network first by recursively identifying primitive combinations of connections and then replacing them with equivalent connections. Recursion terminates either when a single connection remains or when all connectors are fixed, at which point the connectors' positions are determinate. The csolver then unwinds the recursion, apportioning the amount of stretch or shrink applied to each equivalent connection to the connections they replaced until the original network is obtained. Then the csolver issues translation commands to the connectors that must move.

---

[1] Floyd reports an unpublished $\mathcal{O}(n \log n)$ algorithm he invented "around 1963" [11] that is optimal. My thanks to Barry Hayes for bringing that algorithm to my attention.

[2] This supersedes an $\mathcal{O}(n^2)$ UpdateViewStructure algorithm reported earlier [58].

This approach is a compromise between supporting general connector semantics (including connections with connector glue interposed) and ensuring efficient run-time performance. The recursive subdivision technique assumes that an equivalent connection is indistinguishable from the connections it replaces, an assumption that does not hold for connections involving inherently non-linear connector glue. In practice, however, strict adherence to the architectural specification is not necessary for the majority of connections that occur in domain-specific editors. Our implementation models the most common connections (pin-pin) accurately and offers reasonable behavior for more complex connections.

4.6.1 Modeling Connectivity Semantics

The connector glue parameters for each combination are shown in Table 18. The natural size parameters (h and v) are not shown in the table because connected connectors are centered by default; thus all glue used to model pin, slot, and pad connections has zero natural size. Where horizontal and vertical dimensional subscripts ($_{h,v}$) are not specified in the table, the value given applies for both the horizontal and vertical parameter. The notation h_1, v_2, etc., refers to the width or height of the connector with the matching subscript. Also, all elasticity and limit values are identical for both stretching and shrinking modes; that is, $\varepsilon^+ = \varepsilon^-$ and $\lambda^+ = \lambda^-$ for all ε and λ in the table.

To simplify the csolver, slots and pads are limited to orthonormal orientations; that is, they cannot be rotated. This lets us ignore dependencies between horizontal and vertical dimensions. To be useful, however, the implementation must allow at least horizontal and vertical slot orientations. The prototype therefore provides separate horizontal (**HSlot**) and vertical (**VSlot**) connector subclasses, allowing ten different connections between pins, horizontal and vertical slots, and pads.

Table 18 shows that connector glue of zero natural size and elasticity ($h = v = \varepsilon = \lambda = 0$) models pin-pin connection semantics. Pin-pad connections are modeled with a piece of glue of infinite elasticity within limits that keep the pin inside the pad. In connections where connector glue is interposed explicitly between the connectors, the connection's glue is defined by the series equivalent of the specified glue and the glue that normally models

	pad_2	$hslot_2$	$vslot_2$	pin_2
pin_1	$\varepsilon = \infty$ $\lambda_h = h_2/2$ $\lambda_v = v_2/2$	$\varepsilon_h = \infty$ $\varepsilon_v = 0$ $\lambda_h = h_2/2$ $\lambda_v = 0$	$\varepsilon_h = 0$ $\varepsilon_v = \infty$ $\lambda_h = 0$ $\lambda_v = v_2/2$	$\varepsilon = 0$ $\lambda = 0$
$vslot_1$	$\varepsilon = \infty$ $\lambda_h = h_2/2$ $\lambda_v = (v_1 + v_2)/2$	$\varepsilon = \infty$ $\lambda_h = h_2/2$ $\lambda_v = v_1/2$	$\varepsilon_h = 0$ $\varepsilon_v = \infty$ $\lambda_h = 0$ $\lambda_v = (v_1 + v_2)/2$	
$hslot_1$	$\varepsilon = \infty$ $\lambda_h = (h_1 + h_2)/2$ $\lambda_v = v_2/2$	$\varepsilon_h = \infty$ $\varepsilon_v = 0$ $\lambda_h = (h_1 + h_2)/2$ $\lambda_v = 0$		
pad_1	$\varepsilon = \infty$ $\lambda_h = (h_1 + h_2)/2$ $\lambda_v = (v_1 + v_2)/2$			

Table 18: Connector glue parameters for modeling standard connectivity semantics

name	*topology*	*equivalent*
series	1 *a* *b* 2	1 *a + b* 2
parallel	1 *a* *b* 2	1 *a* ∥ *b* 2
Y	1 *a* *b* 2 *c* 3	1 a_Y 2 b_Y 3 c_Y

Table 19: Primitive connections

the connection. Thus, a pin-glue-pad connection is modeled with the equivalent of the given glue in series with the glue already described for regular pin-pad connections.

4.6.2 Primitive Combinations and Equivalents

Csolver identifies three primitive connection combinations: **series**, **parallel**, and **Y**. Table 19 shows the topology of each primitive and its equivalent. The S-shaped terminations denote the points at which the connections fit into the rest of the network. These points are numbered to clarify the relationship between the primitive combination and its equivalent.

The equivalents' glue parameters are calculated from the original glue parameters as shown in Table 20. The row labels refer to the glue for each equivalent connection as labeled in Table 19, and the columns refer to their corresponding glue parameters defined in terms of the original glue parameters. Unless otherwise noted, the relationship shown in each table entry holds in both horizontal and vertical dimensions and in both shrinking and stretching modes; thus n refers to either natural width h or natural height v, and most dimensional subscripts ($_{h,v}$) and elasticity mode superscripts ($^{+,-}$) are omitted. For example, the vertical stretch limit for the series equivalent glue is the sum of the limits from the two pieces of glue in the original connection, that is,

$$\lambda^{+}_{v_{a+b}} = \lambda^{+}_{v_a} + \lambda^{+}_{v_b},$$

glue	n	ε	λ
$a+b$	$n_a + n_b$	$\varepsilon_a + \varepsilon_b$	$\lambda_a + \lambda_b$
$a\|b$	$max(n_a, n_b)$	$min(\varepsilon_a, \varepsilon_b)$	$min(\lambda_a, \lambda_b)$
a_Y	$n_a + n_b$	$min(\varepsilon_a + \varepsilon_b, \varepsilon_c)$	$\lambda_a + \lambda_b$
b_Y	$n_c - n_b$	$\begin{cases} \varepsilon^+_{b_Y} = min(\varepsilon^-_b + \varepsilon^+_c, \varepsilon_a) \\ \varepsilon^-_{b_Y} = min(\varepsilon^+_b + \varepsilon^-_c, \varepsilon_a) \end{cases}$	$\lambda_c - \lambda_b$
c_Y	$n_a + n_c$	$min(\varepsilon_a + \varepsilon_c, \varepsilon_b)$	$\lambda_a + \lambda_c$

Table 20: Parameter definitions for equivalent connector glue in terms of primitive combination parameters

and the natural width of the glue labeled a_Y in the Y equivalent connection is defined in terms of the original glue widths as

$$h_{a_Y} = h_a + h_b.$$

These relationships provide reasonably-behaved connector semantics without attempting to precisely model the non-linear nature of connector glue. Many applications require only standard connectivity semantics and do not use connector glue in connections. It is in these cases and others (where the glue parameters are limited to those shown in Table 19) that the equivalents model their combinations most accurately. Moreover, series and parallel connections are by far the most common; true Y connections (where the central node is not fixed) are rare.

4.6.3 Recursive Substitution

Figure 13 depicts the process of recursive substitution. The network contains three connections. Circles represent connectors, and resistor symbols represent connector glue. The shaded connectors have fixed mobility, while the others are floating. On the initial recursion, the csolver identifies the parallel combination of G2 and G3 and replaces it with an equivalent connection. It replaces the resulting series combination with another equivalent connection on the second recursion, leaving a single connection. Both connectors in the

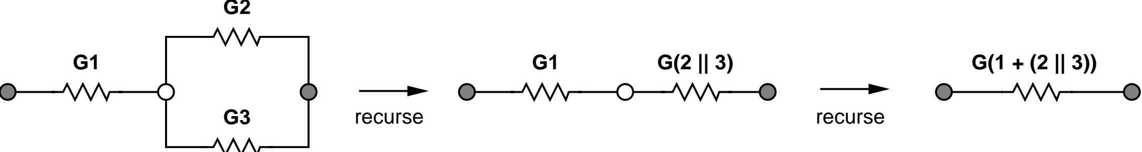


Figure 13: Recursive solution of connection network

primitive	$\langle a \rangle$	$\langle b \rangle$	$\langle c \rangle$
parallel	$lim_a(lim_b(\langle a \Vert b \rangle))$	$\langle a \rangle$	—
series	$\begin{cases} n_a & \text{if } \varepsilon_a = 0 \\ lim_a(\langle a+b \rangle - n_b) & \text{if } \varepsilon_b = 0 \\ lim_a(\langle a+b \rangle \varepsilon_a/(\varepsilon_a+\varepsilon_b)) & \text{otherwise} \end{cases}$	$lim_b(\langle a+b \rangle - \langle a \rangle)$	—
Y	$\begin{cases} n_a & \text{if } \varepsilon_a = 0 \\ lim_a(\langle a_Y \rangle - n_b) & \text{if } \varepsilon_b = 0 \\ lim_a(\langle a_Y \rangle \varepsilon_a/(\varepsilon_a+\varepsilon_b)) & \text{otherwise} \end{cases}$	$lim_b(\langle a_Y \rangle - \langle a \rangle)$	$lim_c(\langle c_Y \rangle - \langle a \rangle)$

Table 21: Actual sizes for primitive combination connector glue in terms of equivalent glue actual sizes

remaining connection are fixed, thus determining both their positions and the amount by which the glue must stretch or shrink.

In general, recursion terminates whenever such a **degenerate** network is reached. A network is degenerate if and only if

1. all connections are fixed, or

2. no primitive combinations remain.

If all connectors are fixed, then the actual size of all glue is determinate. If the network contains floating connectors and no primitive connection combinations are found, then no further recursion is possible; remaining floating connectors are treated as fixed connectors. Once the actual size of all glue is known, the csolver unwinds the recursion, replacing equivalent connections with the corresponding primitive combinations. Csolver calculates the actual size of the glue in primitive combinations from the actual size of the equivalent glue.

The formulas for these calculations appear in Table 21. The notation $\langle x \rangle$ means "the actual size of glue x," and the function lim_i is defined for $i = \{a, b, c\}$ as

$$lim_i(x) \equiv max(min(x, n_i + \lambda_i^+), n_i - \lambda_i^-).$$

This function imposes the deformation limits of the primitive combination glue on its argument.

The final positions of the floating connectors in the network are based on the positions of fixed connections and the actual size of all glue between them. Note that the architecture does not specify which connectors' positions will be affected if all are floating. In the implementation, however, it is certain under such circumstances that some connectors will behave as though they were fixed—namely those that remain at the end of the recursion. Exactly which connectors remain depends on the order of substitution of equivalent connectors, and that depends on both the order in which connections are listed and the order in which csolver searches for primitive combinations. The prototype csolver searches first for fixed connections (those with both connectors fixed, which can be removed outright), then series (which tend to be most common), parallel, and Y connections.

4.6.4 Connector Translation

Once the csolver has computed the positions of the individual connectors, it must then translate them to those positions and notify their parent components of the change. Figure 14 shows what happens when a connection is perturbed.

Assume a transformation (step 1 in the figure) is applied to a component having connector subcomponents, and at least one of these connectors participates in a connection. At some point after the transformation, Update is called on the unidraw object to synchronize the screen with the editor's internal state. Unidraw's Update operation in turn calls Solve on the csolver (step 2), which solves the connection networks associated with the drawing. The csolver then issues translation commands (step 3) to the affected connector subjects, which in turn call Update on their respective views (step 4). As the csolver issues translation commands, it detects when all the connectors of a given component have been translated; it then notifies the component subject by calling its Update operation (step 5). Notifying the subject only once avoids the overhead associated with notifying the subject each time the

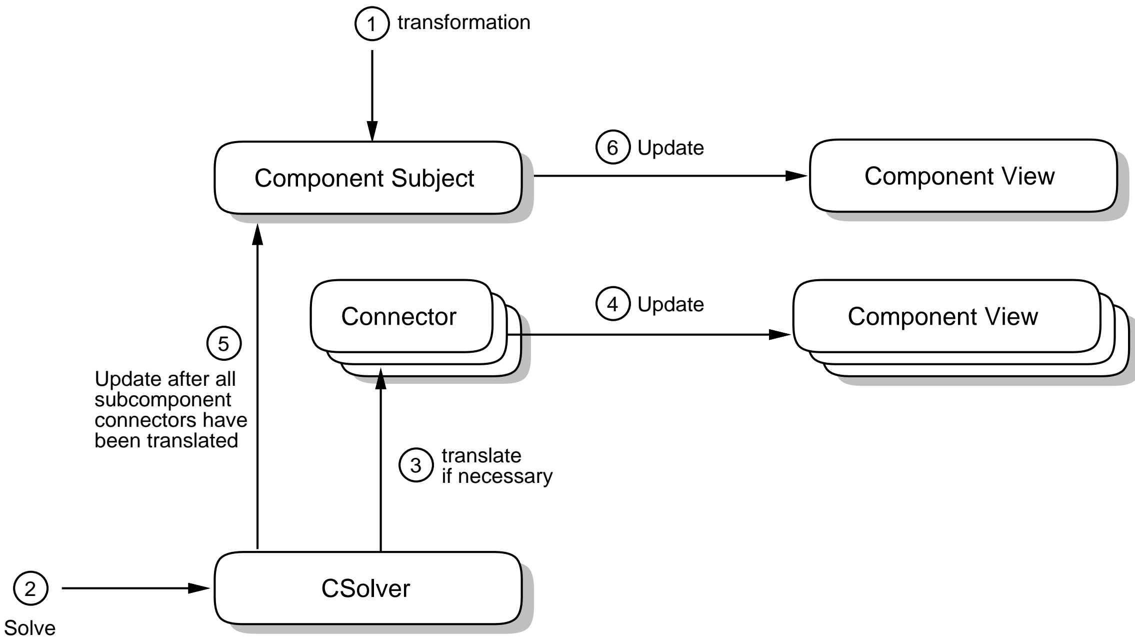


Figure 14: Response to connection perturbation

csolver translates one of its connectors. Finally, the component subject updates its views in response to the csolver's notification (step 6).

4.6.5 Algorithmic Complexity

The time required to solve the connection networks depends on their number and size. We characterize the running time in terms of the number of connections.

At one extreme, all connections are disjoint—each network has only one connection. Here each network is degenerate by definition, so its solution takes constant time. Therefore the time to solve all such networks is linear. At the other extreme, there are no disjoint sets of connections—there is only one network. The running time is proportional to the number of connections multiplied by the average length of a search for a primitive combination. The worst case occurs when all connections are examined on each recursion. Though the list of connections shortens on each recursion, it still takes $n(n+1)/2 \propto n^2$ searches, where n is the number of connections. If we can identify a primitive connection in constant time, then overall the algorithm is $\mathcal{O}(n^2)$ at worst.

The csolver stores information in each connector that it uses to identify primitive combinations in constant time. Whenever a connection to a previously **free** (i.e., unconnected) connector is established, the csolver stores in the free connector the address of the connection network in which the connection (and consequently the connector) resides. Csolver also stores a reference to the free connector's **peer** (the connector to which it is connected) if they have not already been connected; otherwise the csolver increments a count representing the number of connections between the two connectors. It also increments a count of the total number of redundant connections to the free connector. Likewise, csolver records similar information in the peer. If the peer is also free, then a new network is created. If two non-free connectors are connected, and if their networks differ, then the two networks are merged, guaranteeing that networks remain disjoint.

The network information expedites finding the right network for a connection, while the connection information helps identify primitive combinations quickly. For example, two connections in series share a connector; this connector will have a record of exactly two connections in which it participates, each to a different peer. Similarly, three Y-connected connections will share a single connector; that connector will have a record of exactly three connections in which it participates, and no peers can be the same. If a connection is in parallel with another, their connectors will record at least one redundant connection. Thus all primitive combinations can be detected in a linear search through the network.

We can expect significantly better than $\mathcal{O}(n^2)$ performance on average in many applications. Editors that use only the standard connectors with no glue interposed realize linear average running times if we assume that few if any multiple connections occur; that is, each pin, slot, or pad is connected to one other. Each connection is disjoint in this case, and the running time is linear in the number of connections. In general, nodes-and-arcs-style connectivity is maintained in linear time, unless the nodes and arcs have elastic characteristics. Elastic components make internal connections with glue interposed, thus connecting more connectors together and reducing the number of disjoint networks. Often, however, such elastic behavior is not necessary; only simple connectivity or confinement is required, and components do not define internal connections. For these applications, network solution takes linear time. But even editors that produce few disjoint networks fare better than the worst case. Even a few smaller networks reduce the complexity substantially compared to

solving one big network, where the squared term is several times larger. Yet applications that rarely produce disjoint networks are conceivable, and these will have $\mathcal{O}(n^2)$ running time at worst.

4.7 Dataflow

Connectors, state variables, and transfer functions are the sole participants in the dataflow model defined by the architecture. The prototype implementation adds another class of object, called **Path**, to detect circularities. A path maintains a record of connectors that have been **visited**, that is, connectors through which data has passed. The path class defines two functions: Visit registers a connector with the path as having been visited, and Visited returns whether or not a connector has been visited. By default, transfer functions and components call Transmit on a connector if a path does not record that the connector has been visited. In the implementation, Transmit and Evaluate take a path as an optional argument on which to base this decision.

Figure 15 presents the dataflow algorithm, and Figure 16 depicts the participating objects schematically. Dataflow is initiated when Transmit is called on a connector. A component might call Transmit because it modified one of its state variables in response to a command and it must propagate this change via dataflow. A path is not normally supplied to the initial Transmit operation; Transmit creates a path if one is not supplied and then proceeds with the dataflow algorithm.

Let c_0 represent the connector on which Transmit is called initially, and c_0 has a bound state variable s_0 (otherwise no data can flow). For each unvisited peer of c_0 with a bound state variable, Transmit

1. selectively assigns the value of s_0 to the peer's bound state variable according to the connectors' transmission methods,

2. tells the transfer function maintained by the peer's parent (if any) to Evaluate itself,

3. notifies the parent to Update its internal state, and finally

4. calls Transmit on the peer.

$c_0 \equiv$ connector on which Transmit was called initially;
$c_1, c_2, \ldots, c_n \equiv$ peers of c_0;
$s_1, s_2, \ldots, s_n \equiv$ state variables bound to $c_1, c_2, \ldots, c_n$, respectively;
$s_i', s_i'', \ldots, s_i^{(m)} \equiv$ state variables dependent on s_i, $1 \leq i \leq n$;
$X_i \equiv$ transfer function defining dependency $\mathcal{F}^{(j)}$ between s_i and $s_i^{(j)}$, $1 \leq j \leq m$;
$c_i', c_i'', \ldots, c_i^{(m)} \equiv$ connectors bound to $s_i', s_i'', \ldots, s_i^{(m)}$, respectively;

```
c_i : Transmit(P_j) {
  P_j ≡ a path;

  for all (c_k, 1 ≤ k ≤ n) {
    if (not P_j : Visited(c_k)) {
      if (k > 1 and k ≠ j) P_k ≡ copy(P_j);
      s_k ← s_i;
      X_k : Evaluate(P_k);
      Parent(c_k) : Update();
      c_k : Transmit(P_k);
    }
  }
}

X_i : Evaluate(P_i) {
  P_i ≡ a path;

  for all (s_i^(k), 1 ≤ k ≤ m) {
    s ≡ F^(k)(s_i);

    if (s_i^(k) ≠ s) {
      s_i^(k) ← s;
      c_i^(k) : Transmit(P_i);
    }
  }
}
```

Figure 15: Dataflow algorithm

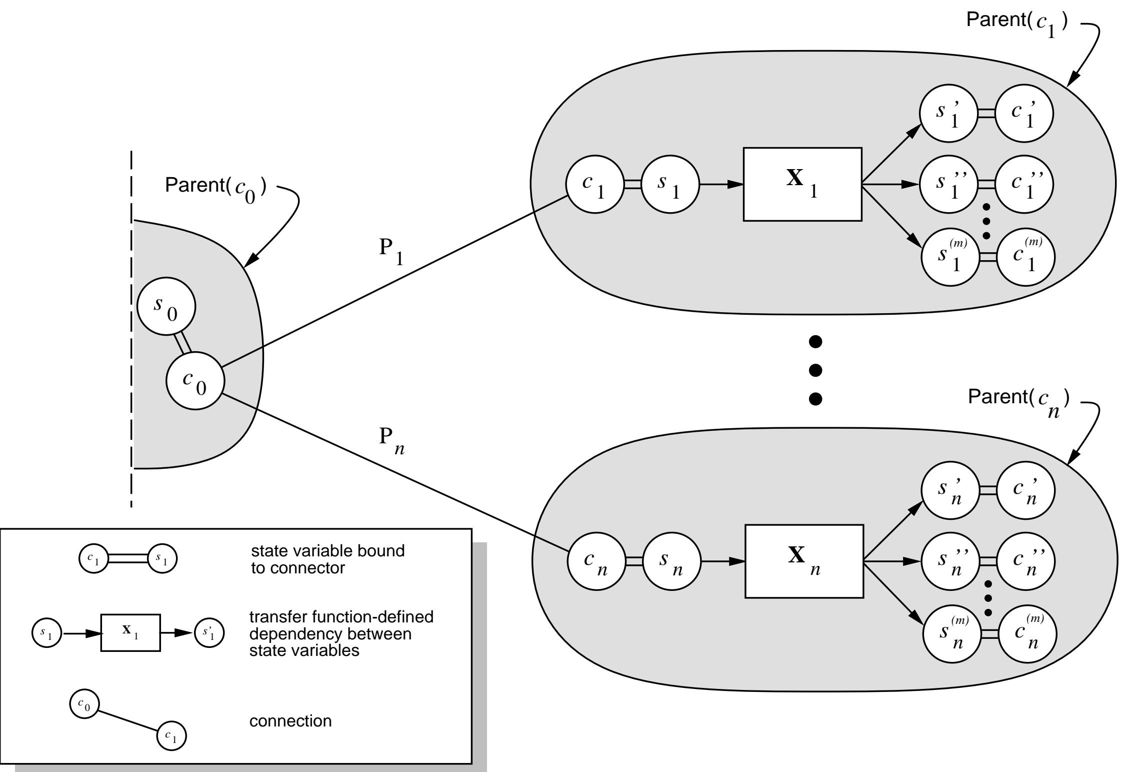


Figure 16: Dataflow schematic

Steps 1 and 2 actually propagate data. The semantics of assignment between state variables is defined by the state variables themselves, since only they know what information should be transferred. The direction of the assignment, however, is determined by the transmission methods of the two connectors. By default, information is transferred through a connection between c_0 and peer c_i with bound state variable s_i if and only if c_0 is either out or inout *and* c_i is either in or inout. This prohibits transfer between two in or two out connectors and disallows information flow from an in to an out connector.

Step 3 gives the peer's parent a chance to update its state in response to a possible change in one of its state variables. For example, a component that provides visual feedback about the state of a system might use a state variable to represent that state. If that state variable is then affected through dataflow, step 3 insures that the component can modify its appearance to reflect the variable's new value. Finally, step 4 initiates transmission on the peer should

it have peers of its own (including c_0 itself, though potential backflow is disallowed because c_0 has been visited already).

Now let $s_i', s_i'', \ldots, s_i^{(m)}$ denote the state variables that depend on s_i as defined by the transfer function of the peer's parent. In response to the Evaluate message, the transfer function computes the new value of each dependent state variable $s_i^{(k)}, 1 \leq k \leq m$. If the current value of $s_i^{(k)}$ does not correspond to the new value, then the transfer function assigns the new value to $s_i^{(k)}$ and calls Transmit on the connector $c_i^{(k)}$ bound to it, if any. Then the dataflow algorithm is applied recursively to $c_i^{(k)}$.

4.8 Commands and Tools

The main complication in the implementation of commands stems from strict type checking in C++. A component cannot interpret a command without knowing the type of command, so we embed type information in commands explicitly. Tools also require such information to let component views determine the proper manipulator to create for a given tool. We therefore add type information to tools as well. We also realized that type information would be required to support catalog semantics, described in Section 4.9, so we put type information into all objects managed by the catalog.

A common implementation decision involves static versus dynamic command composition. The library defines a **MacroCmd** command subclass that supports composition of command instances. When a MacroCmd is executed or unexecuted, it simply (un)executes its children. By default, the component base class (un)interprets a MacroCmd by (un)interpreting its children. The MacroCmd thus provides an alternative to defining a command statically, assuming the command is decomposable into a sequence of simpler commands.

Dynamic command composition is desirable because it is often easier to compose existing commands than to implement a new command from scratch. Moreover, composition affords the flexibility to create new commands and modify existing ones at run-time. On the down side, a dynamic composition can be somewhat less efficient in space and time than a command specified statically, but the difference is usually imperceptible to the user.

The key to realizing the benefits of composition rests in the library—the commands it provides must be designed with composition in mind. This is a recurring theme in library design: the more general and less monolithic the library-supplied objects are, the more likely they will be reused and the greater the leverage delivered to the programmer. But even if such objects are supplied, they are of limited use without techniques for composing them into more sophisticated objects. The prototype currently provides only one composition mechanism for commands, namely MacroCmd. Other useful command composition classes might support classic programming language constructs such as control structures for looping and iteration as well as data structures for specifying stacks and queues of commands.

Some tool semantics can be specified dynamically as well. While the library does not support composition of tools themselves, it does allow manipulator composition via the **ManipGroup** subclass of manipulator. A ManipGroup manages an arbitrary number of child manipulators. Calling Grasp, Manipulating, or Effect on the ManipGroup calls the corresponding operation on each child. The ManipGroup's Manipulating operation returns true if any child's Manipulating operation returns true, and it calls Manipulating only on those children that have not yet returned a false value. ManipGroup thus provides a simple form of manipulator composition in which several manipulations can proceed at once; however, it does not support mixing the way different manipulators interpret events.

The prototype predefines only four other manipulators:

1. **DragManip** supports a downclick-drag-upclick style of interaction, with optional constraints on motion (for example, horizontal or vertical only).

2. **VertexManip** is a DragManip that supports multiple downclick-and-drag interactions terminated by a distinguished downclick.

3. **ConnectManip** is a DragManip that adds a gravitational bias towards connector views.

4. **TextManip** provides a text editing interface.

These manipulators cover most of the direct manipulation needs of current graphical object editors. This is possible because these manipulators are parameterized to handle different

animation effects. For example, DragManip takes a **rubberband** as a parameter. Rubberbands are InterViews objects that abstract the animation effects common to graphical object editors.

A rubberband varies its appearance as its **tracking point** changes, and it ensures that that graphical update happens quickly enough so that changes appear animated. InterViews predefines many rubberbands, such as **RubberRect** (a rectangle of varying shape), **SlidingEllipse** (an ellipse that moves to follow the tracking point), and **GrowingBSpline** (a B-spline that reshapes itself as control points are added and moved). Thus, a single DragManip class supports creation and modification by direct manipulation of most graphical components.

4.9 Catalog Semantics

The architecture specifies that the catalog manages components, commands, and tools and that these objects remain accessible indefinitely; consequently they must survive beyond the application's lifetime. The prototype maps component, command, and tool names to file names and stores a representation of the object in the corresponding file. Such objects must therefore define what information is written to and read from disk by defining Write and Read operations. Since components can contain state variables and transfer functions, moreover, these too must define Read and Write.

Each class of object that can be written to and read from disk must store type information so that it can be reconstructed when it is read. Programmer-specified class identifiers uniquely identify instances of each catalog-managed class. A **Creator** object maps each class identifier to a constructor for the corresponding class. The catalog prepends the identifier to the information that the instance writes to disk. When the instance is read from disk, the catalog reads the identifier and supplies it to the creator object, which creates an instance of the object. The catalog then calls Read on the new object to initialize its internal state.

This approach is applied recursively to build component and command hierarchies. The catalog manages references between objects in a hierarchy (connections between connectors, for example) by keeping a table of unique instance identifiers when it reads

and writes the hierarchy. More specifically, whenever the catalog saves a connector in a component hierarchy, it first checks to see if that connector has been written previously; if not, it writes the connector out in its entirety along with a unique instance identifier, which it records in a table. If the connector has been written already, then the catalog looks up its instance identifier in the table and writes it out in the connector's place. When the catalog later reads the component hierarchy, it instantiates connectors as it encounters them, and it records their instance identifier in a table. The catalog recognizes subsequent references to instantiated connectors by looking up their instance identifiers as it finds them.

Obviously, the library must let domain-specific editors define their own objects for the catalog to manage. The programmer must ensure uniqueness among class identifiers, both within the application and with the respect to the library. The prototype reserves a range of class identifiers but places no restrictions on those outside the range. Similarly, domain-specific editors must have domain-specific creator objects. The programmer derives from the creator object to add constructors for the new class identifiers.

Finally, the prototype extends the catalog protocol to support reading and writing of **EditorInfo** objects, which store a list of strings or string tuples. Domain-specific editors can use these objects to store information about what components, commands, and tools they incorporate in their interface. The user could then specify the configuration of the editor by editing the file that contains this information.

4.10 Summary

Our prototype Unidraw implementation benefits greatly from both C++ and InterViews. C++ allowed us to express the implementation in object-oriented terms without sacrificing the efficiency or portability of C. InterViews provided a comprehensive user interface toolkit with object-oriented structured graphics, which eliminated considerable low-level graphics work. As a result, the Unidraw library is much smaller than it would have been had we used C and a less powerful toolkit.

The implementation of components was the hardest part of the development effort. In particular, it took a disproportionate amount of time to formulate and implement our approach to enforcing connectivity semantics, while dataflow, view consistency, and other

implementation problems were less difficult to solve. Overall, however, the time spent on the implementation effort was less than that spent designing and refining the architecture. And though the architecture and implementation efforts did overlap and complement each other, the implementation did not affect the basic architecture, which has not changed fundamentally since it was proposed [56].

The prototype demonstrates that the Unidraw architecture is realizable. It implements all the functionality prescribed by the architecture except non-orthonormal slots and pads and truly non-linear connector glue. Yet these compromises had minimal impact on the performance of the three experimental domain-specific editors we built, which are described in the next chapter.

Chapter 5

Experimental Applications

We implemented three domain-specific editors to evaluate both the Unidraw architecture and the prototype implementation. Our aim was to demonstrate that the implementation (and, by implication, the architecture) supports diverse domains, that it reduces development time significantly, and that the resulting graphical object editors are comparable in utility to their conventionally-developed counterparts.

The editors we built include a drawing editor, a user interface builder, and a schematic capture system. We chose these three applications because each represents a traditional stand-alone application. Moreover, there is little overlap in their design goals; they are different enough to preclude easily turning one into another using existing tools. For example, user interface builders are not usually designed to produce drawings, and schematic capture systems are not meant to facilitate building user interfaces. Hence a tool that simplifies the development of all three applications should do the same for other domain-specific editors as well.

We experimented with the three editors to evaluate their utility and to discover their strengths and weaknesses relative to conventional implementations. This chapter covers both the design of these domain-specific editors and our experience with them.

5.1 Drawing Editor

The drawing editor, called **Drawing**, is similar to MacDraw in that it provides an object-oriented, direct-manipulation editing environment for producing drawings and diagrams. It allows the user to instantiate geometric objects, arrange them spatially, and compose them hierarchically. The user can apply affine transformations to the objects and specify graphical attributes such as color, font, and fill pattern. The user can also pan the drawing and view it at different magnifications. Drawing generates a PostScript external representation for printing the drawing and incorporating it into larger works.

Unlike MacDraw, Drawing also supports multiple views. The user can edit in one view, say, at high magnification for detailed work, while the drawing is fully visible in another view for editing at low magnification. Unidraw ensures that changes to the drawing made in one view appear automatically in other views. Drawing also gives users considerable control over its interface, allowing them to include only the components, commands, and tools they need.

5.1.1 User Interface

Figure 17 shows a Drawing editing session and depicts the default interface schematically. The application presents one or more editor instances, each enclosing a viewer, pull-down menus containing **controls** that execute a specific command, controls for engaging the current tool, a **panner** for panning and zooming the viewer, and state variable views that display the values of state variables maintained by the editor. The controls, pull-down menus, and panner are defined by or derived from toolkit objects, while other objects are based on Unidraw abstractions.

Drawing creates a single editor instance initially. The user engages the current tool by clicking on the appropriate control along the editor's left edge. Two types of tools are available: graphical component tools for instantiating graphical components and tools for manipulating components already instantiated. Drawing provides graphical component tools for eight different components: text, simple line, compound line, ellipse, rectangle, polygon, open B-spline, and closed B-spline. Other tools include a Select tool for specifying components of interest; Move, Scale, Stretch, and Rotate tools for applying affine

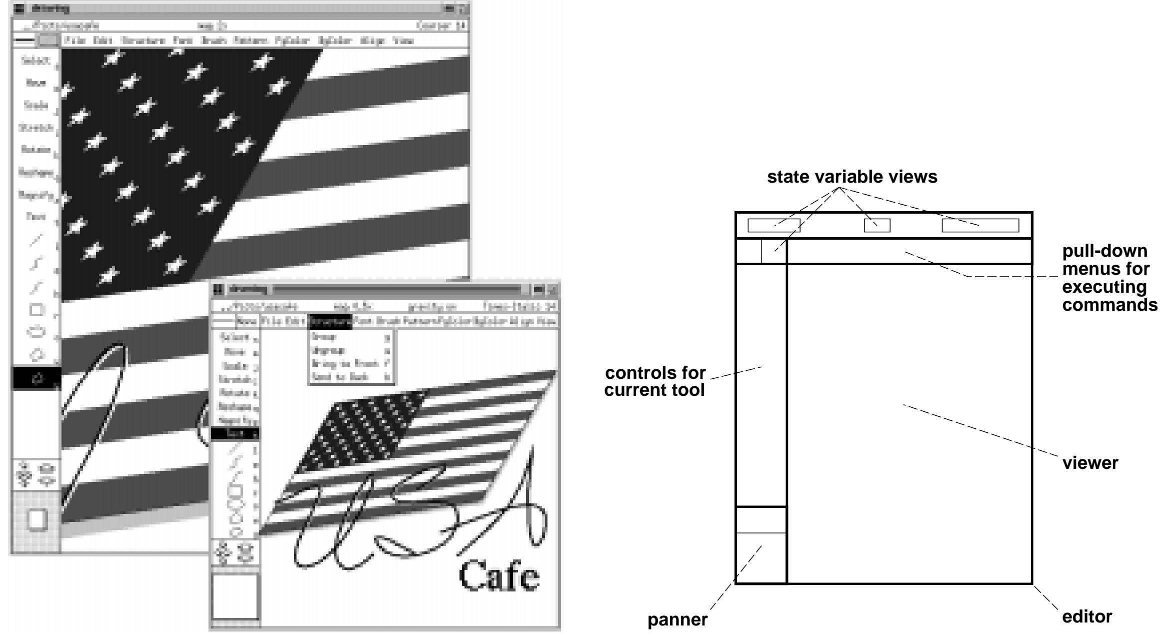


Figure 17: Drawing editor

transformations by direct manipulation; a Reshape tool for repositioning vertices and control points and for editing text components; and a Magnify tool that lets the user zoom in on a particular part of the drawing by dragging a rectangle around the area of interest.

Once a tool is engaged, the user wields it by clicking in the viewer. The tool's behavior thereafter varies with the type of tool and the component(s) on which it is used, if any. All the tools provide dynamic feedback to show the consequence of their use. For example, the graphical component tools obtain a manipulator from their prototype component; the resulting component-specific animation supplies feedback that helps the user specify the new component's size and shape with ease. Transformation-applying tools such as the Rotate and Stretch tools animate the intermediate steps of the transformation before the user commits to the final transformation. The user can change the position of a spline's control point with the Reshape tool and watch the spline's shape change in real time.

Each pull-down menu contains a set of controls, each with an associated command. A control executes its command after the user selects the control through the pull-down menu. Most of Drawing's commands are predefined by the Unidraw library. These include commands for saving and restoring drawings via the catalog; modifying attributes such

as line style and color; cutting, copying, and pasting components to and from a global clipboard; undo and redo commands; and structuring commands for grouping, ungrouping, and otherwise altering the component hierarchy.

To create a new view of the current drawing, the user invokes a command from the View menu that copies the editor, including its commands, tools, viewer, and other interface elements. The copy creates a new view of the component being edited and places the view in its viewer. The user may then edit the component subject through either the original editor or the copy; changes made through one editor are reflected in the other. Any number of editors may be produced in this way. Once instantiated, they can be used to edit the original component subject or to edit other drawings. Each editor maintains its own selection object, allowing the user to select objects in one editor without disturbing the selections in another, even if the editors contain the same subject.

Drawing's interface can be reconfigured at start-up time to include only those features a user requires. Through a command line option, the user can direct Drawing to write out a configuration file containing a list of individual and pairs of strings; the individual strings represent tool names, while the pairs associate a command name with the name of a pull-down menu. When the user starts Drawing with this file as an argument, the editor will incorporate into its interface only those commands and tools whose names it encounters in the file. The order in which the names appear dictates the arrangements of the controls in the tool palette and pull-down menus. By editing the configuration file, then, the user can exercise control over which commands and tools are included and their arrangement.

5.1.2 Implementation

Drawing represents a lower bound of functionality and complexity for practical Unidraw applications. In general, drawing editors are the simplest graphical object editors in that most offer only

- graphical components with no underlying semantics,
- basic commands and tools for creating, manipulating, and modifying those components, and

		code (lines)	
	classes	*interface*	*implem.*
components	0	0	0
commands	1	20	40
tools/manipulators	0	0	0
ext. representations	0	0	0
state vars./transf. fns.	0	0	0
editors	1	80	890
creator	1	20	30
toolkit-derived classes	0	0	0
globals	0	30	110
totals	3	150	1070

Table 22: Drawing code breakdown

- external representations that are straightforward mappings of their component structure onto a page description language such as PostScript.

These elements are useful in many domains other than drawing, and most graphical object editors incorporate at least some drawing editing capabilities in their own interfaces.

For these reasons the prototype Unidraw library predefines all the components, commands, and tools needed for basic drawing editing. Drawing's implementation assembles these predefined elements into a complete application with a minimum of new classes and code. Drawing does not exploit many of the architecture's advanced features, but it does demonstrate useful capabilities, particularly multiple views and static interface customization. Drawing thus constitutes a bare-bones Unidraw application; it serves both as an example of a trivial (but useful) test case and as a tutorial for application writers unfamiliar with Unidraw.

Table 22 presents a breakdown of the Drawing implementation in classes and lines of source code. Since most of Drawing's functionality is inherited from the Unidraw library, the Drawing-specific code is dedicated mainly to defining the application's look and feel.

Drawing defines three new classes, the largest a subclass of editor called **DrawingEditor**. Each window shown in Figure 17 is a DrawingEditor instance providing an independent interface to editing a drawing. Each DrawingEditor has its own component, command, and tool instances and its own selection object. The lone command defined in Drawing, **NewViewCmd**, creates a new DrawingEditor instance by copying the original. Then it creates a new view of the component in the original editor, places it into the copy's viewer, and tells the unidraw object to Open the editor, making it visible on the screen. There is also a **DrawingCreator**, derived from the library's Creator class, that allows the catalog to recreate NewViewCmds from disk.

DrawingEditor uses an EditorInfo object to record the names of the commands and tools in its interface. If the user supplies a configuration file name on the command line, then DrawingEditor asks the catalog to supply an EditorInfo object corresponding to the name. If the catalog returns a valid EditorInfo object, then the editor creates and assembles commands, tools, and controls based on its information. The catalog returns nil if it could not create an EditorInfo object. This would happen if the user has never before supplied a command line argument to produce the configuration file; DrawingEditor assumes this to be the case and instantiates a new EditorInfo object with information reflecting the editor's default appearance. It then tells the catalog to save the EditorInfo object for later use, which will produce a configuration file with the specified name.

5.1.3 Experience

Of the three experimental applications we built with the Unidraw prototype, Drawing gives us the best opportunity for comparison with existing editors. In designing Drawing we tried to match the functionality and look and feel of idraw [49], an object-oriented drawing editor developed and distributed with InterViews. Idraw provides roughly the same functionality as MacDraw, but idraw is more appropriate for comparison purposes because it is a workstation-based application that runs in the X environment. Idraw has a substantial following of users and has established itself as one of the leading object-oriented drawing editors for X. We therefore used idraw as a benchmark in evaluating Drawing's performance.

Idraw is implemented on top of InterViews, and since InterViews abstracts the underlying window system, idraw does not call X directly. Idraw uses InterViews' predefined interactors to compose its look and feel, and it uses the toolkit's structured graphics objects to support its graphical editing capabilities. Even with these assists, however, idraw still contains roughly 15,000 lines of source code. The difference in source code size between idraw and Drawing is considerable, but in fairness we should offer the following caveats:

- Drawing is particularly small, even compared to other Unidraw-based editors, because most of its functionality is provided by the Unidraw library. This follows our assertion that drawing editors represent the lowest-common denominator among graphical object editors.

- Idraw and Drawing are nearly identical in functionality and appearance, but there are notable differences. Drawing provides multiple views and a customizable interface, features absent from idraw. On the other hand, idraw offers arrowheaded lines and operations for specifying graphical transformations numerically; Drawing does not support arrowheads and provides only a direct-manipulation interface for transforming objects.

- Idraw benefits from two years' refinement and user experience. We plan to replace idraw eventually with Drawing for our day-to-day drawing tasks, and doubtless this will yield improvements in both Drawing and the library that will affect their line count.

Despite these qualifications we can still conclude that Unidraw substantially reduced the size of the Drawing implementation, with an attendant decrease in implementation time. We also verified that Drawing is a viable replacement for idraw by measuring the storage requirements and dynamic response of both editors. The measurements were made on an 8-plane DECstation 3100 using InterViews version 2.6 and the X11 Release 4 server and libraries.

To characterize the editors' run-time space requirements, we measured the size of their processes under six conditions:

1. Immediately after startup ("cold").

	process size (MB)		
	idraw	*Drawing*	*Δ%*
cold	1.43	1.58	10.5
warm	1.46	1.65	13.0
1 testobj	1.48	1.66	12.2
10 testobjs	1.52	1.70	11.8
100 testobjs	1.86	2.10	12.9
1000 testobjs	5.55	6.58	18.6

Table 23: Run-time space requirements for idraw and Drawing

2. After having exercised their interfaces (pulling down menus, scrolling their drawing areas, etc.) but before creating any drawing objects (“warm”).
3. After having read in a **testobj**, a group of eight different geometric objects, one for each of the basic geometric objects the editors provide.
4. Displaying 10 copies of testobj.
5. Displaying 100 copies of testobj.
6. Displaying 1000 copies of testobj.

Table 23 shows that the Unidraw-based editor takes slightly more space to represent a drawing because of the overhead for maintaining multiple views. The predefined graphical components require almost twice as much storage as the corresponding idraw objects because the components maintain graphical information in their subject as well as their views. The added cost could be reduced if we treat the single-view case specially, but we have not applied such optimizations.

Another set of measurements relates idraw and Drawing’s run-time performance. We measured how long it took idraw and Drawing to do each of four operations on two drawings, a sports car (**car**) and a circuit diagram (**ckt**). These are representative of two common types of drawings: artistic drawings with complex, overlapping polygons and splines, and

		time (sec)	
	test	*idraw*	*Drawing*
car	zoom #1	0.67	0.66
(53 leaf objects	zoom #2	0.74	0.75
nested 5 deep)	rotation	0.38	0.39
	restruct	0.49	0.64
ckt	zoom #1	0.94	0.70
(440 leaf objects	zoom #2	0.65	0.49
nested 4 deep)	rotation	0.43	0.42
	restruct	0.64	0.98

Table 24: Comparison of idraw and Drawing run-time performance

technical drawings consisting mainly of rectangles, lines, and text with little or no overlap. We timed the following operations:

1. In the "zoom #1" test, the drawing is zoomed from half size to quarter size and back. The drawing is fully visible throughout the test.

2. In "zoom #2," the drawing is zoomed from half size to full size and back. The drawing is clipped at full size so that only half is visible.

3. In "rotation," the top-level object in the drawing is rotated $90°$.

4. In "restruct," the top-level object in the drawing is ungrouped and grouped again.

Table 24 shows the average times for each test based on twenty trials. Drawing performs at least as well as idraw in all but the restruct test. Idraw uses InterViews' structured graphics facility to implement its graphical objects, and the prototype Unidraw library uses the same structured graphics to represent the graphical attributes of components. Since the first three tests involve changes to graphical attributes, then, it is not surprising that the times for these tests are virtually identical. This proves that Unidraw-related overhead such as

subject-view communication and command interpretation and logging does not measurably affect performance.

The Unidraw-based editor's performance does suffer slightly compared to the custom-built editor when a component is restructured. The overhead stems from the grouping operation: the multi-view update algorithm discards the old views of the children that have been grouped, and then it creates a view for the group, which includes views for the children. In contrast, idraw simply reparents the existing graphical objects without destroying or re-creating them. The update overhead amounts to a constant factor, because the destruction and re-creation of the views amounts to two extra traversals of the component hierarchy—both idraw and Drawing must perform similar traversals to draw the graphics following the grouping operations. In practice, it appears that this overhead is inconsequential; at any rate, the comparison is not entirely fair, since the Unidraw-based editor offers additional functionality in the form of a multi-view editing environment in exchange for somewhat slower response in some cases.

From these findings and from hands-on experience with both editors, Drawing easily meets our goals in providing a viable replacement for its custom-built counterpart. Drawing offers essentially the same functionality and performance as idraw at a fraction of idraw's implementation cost, measured in lines of code. Since drawing editors reflect the basic capabilities of graphical object editors, these results imply that the basic operation and graphics performance of Unidraw-based editors will be comparable to conventional implementations—Unidraw incurs no significant space or time penalties.

5.2 User Interface Builder

User interface builders are designed to let the user specify the appearance of a user interface by direct manipulation instead of programming. The goals of such systems are threefold:

1. Shorten design time for a particular interface by eliminating the edit-compile-debug cycle from the design process.

2. Shorten implementation time by generating code from the graphical specification produced through direct manipulation.

3. Allow non-programmers, especially those with graphic design and human factors skills, to design and implement user interfaces.

An effective user interface builder should therefore reduce application development time as it encourages the design of higher quality user interfaces.

Our Unidraw-based user interface builder, called **UI**, provides a direct-manipulation environment for assembling InterViews toolkit objects into a complete interface. UI defines components that correspond to basic interactors such as scroll bars and push buttons, and it provides components that implement the InterViews composition mechanisms. UI components closely match the composition behavior of their toolkit counterparts, supporting the semantics of interactor attributes such as shape and canvas. Thus the interface designer can experiment with an interface without writing and compiling source code. UI generates C++ code from the graphical specification as its external representation. Once the interface designer is satisfied with the interface's appearance and behavior, he can generate the source for incorporation into the application.

5.2.1 User Interface

Figure 18 shows a dialog box being built with UI. Along the top edge are state variable views of internal state and pull-down menus for issuing commands. In the center is a viewer in which the user composes interfaces. Below the viewer are two rows of controls and a panner for scrolling and zooming the viewer's contents. The upper row's controls engage graphical component tools for instantiating user interface components, while the lower row's engage tools that manipulate instantiated components. UI supports multiple views in the same way Drawing does, that is, by copying the editor and editing the original component subject through a new view. UI also incorporates the Select and Move tools from the Unidraw library, and each UI editor maintains its own selection.

UI provides components for a subset of the predefined interactors in InterViews, including scrollers, buttons, borders, glues, message, string editor, and file browser. The graphical component tool controls let the user create these components in the same way that graphical components are created in Drawing. The user engages the graphical component tool of choice by clicking on the corresponding control; he then creates a copy of the prototypical

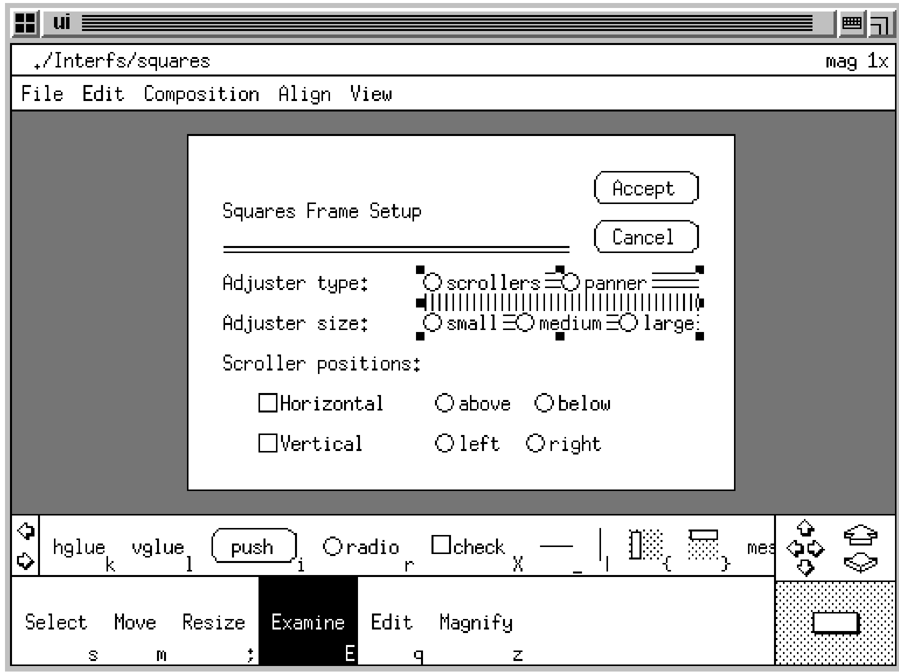


Figure 18: User interface builder

component by clicking in the viewer. Most UI interface components adopt the default behavior of graphical components when they are manipulated by a graphical component tool; that is, a box corresponding to the bounds of the prototypical component appears when the user clicks down in the viewer. The user can drag the bounding box to any position while the button is down, and the prototype is copied and inserted into the interface when the button is released.

UI also defines components that implement InterViews composition mechanisms, including box and tray. These components have no appearance of their own; they compose other interface components in different ways. Composition components are created through commands in the Composition menu. For example, the user can place components into a horizontal box by selecting them first and then choosing the "HBox" entry in the menu. The command inserts components into the box in the same order they were selected. The

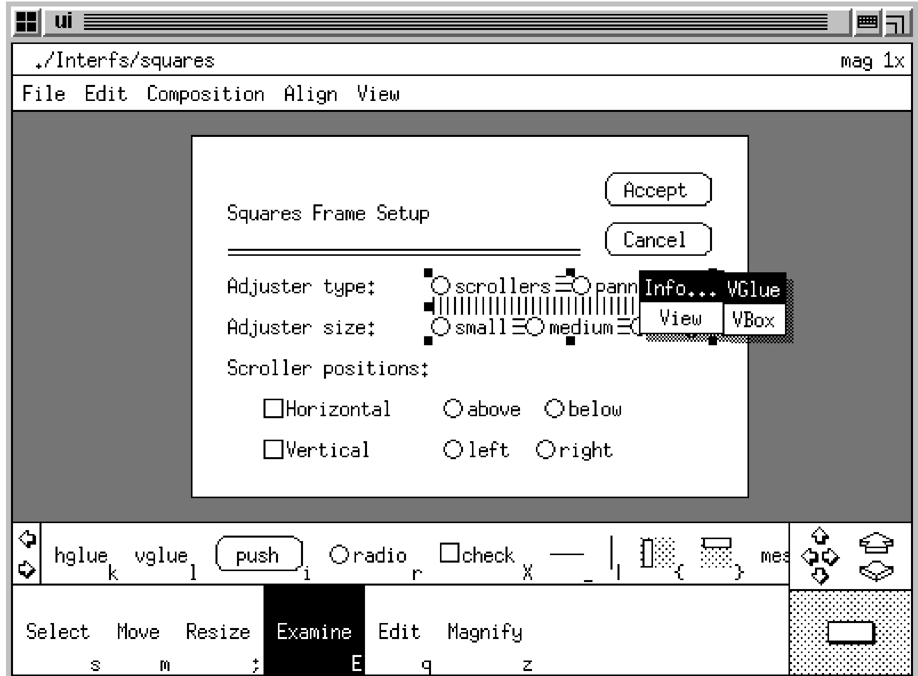


Figure 19: Pop-up from a click on a glue component with the Examine tool

box then tiles the components horizontally to effect the composition semantics of horizontal boxes. The composition can then be treated as a unit and incorporated into larger compositions.

UI provides Resize and Examine tools for manipulating interactor components. The controls for engaging these tools lie alongside the controls for the Select and Move tools. The Resize tool lets the user observe how an interactor (primitive or composite) would be affected by a change in the size of its canvas. This feature can help the user fine-tune the shape (natural size, shrinkability, and stretchability) of glue and other interactors in the interface.

The Examine tool lets the user examine and modify a component's internal state, such as its shape, canvas, or button state (if any). Clicking on a component with the Examine tool engaged produces a pop-up menu containing two submenus: one for displaying and modifying the component's internal state or that of its parents, and one for creating new views on portions of the composition hierarchy in which the component lives.

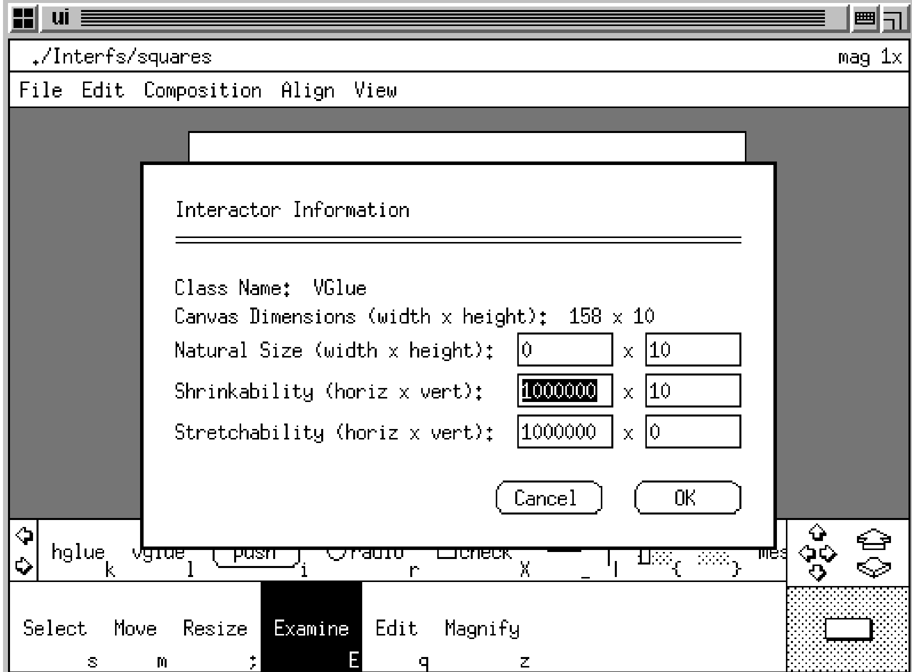


Figure 20: Dialog box containing glue component information

In Figure 19, the user has clicked on a piece of vertical glue (which renders itself with vertical bars by default for easy identification) and selected the “Info...” submenu, whose body lists the names of interactors in the hierarchy from the glue up to its parent. Revealing the hierarchy in this way allows the user to examine internal nodes of the composition without disturbing it. Figure 20 shows the result of choosing to examine the glue component: a dialog box containing information about the glue’s canvas and shape. Since InterViews allows programmers to specify shape parameters of glue in its constructor, the dialog allows the user to change these parameters in the dialog. The user cannot alter these parameters for buttons, for example, since the programmer cannot change these values in InterViews without deriving a new button subclass.

The Examine tool also allows the user to create views of parts of a composition, permitting him to edit otherwise inaccessible parts of the hierarchy. The “View” submenu in the pop-up menu displays a list of the chosen component’s parents; Figure 21 shows this list for a radio button in the composition. The user clicked on a radio button composed

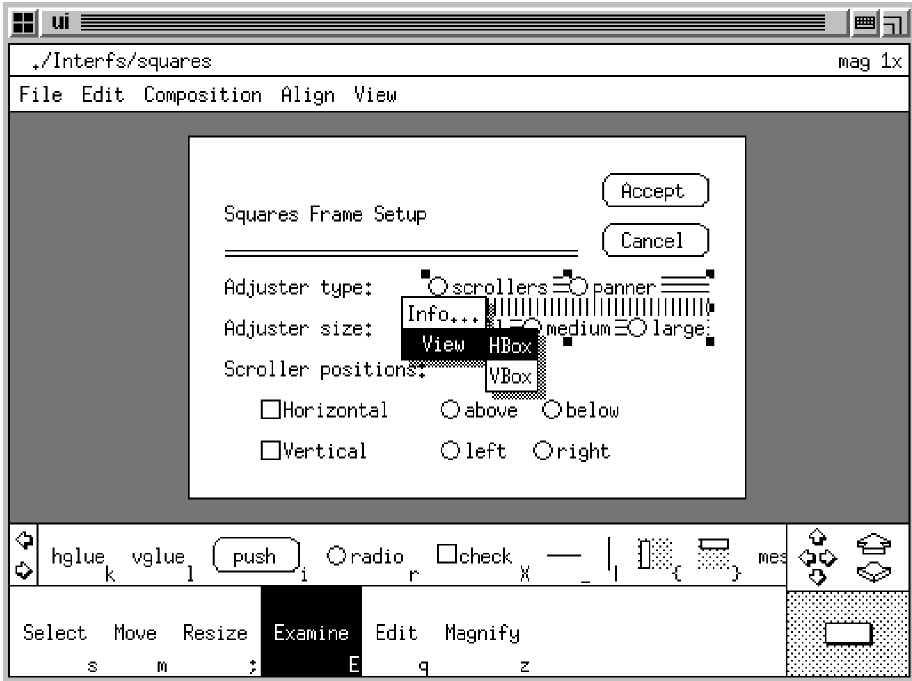


Figure 21: Examining a composition hierarchy

in a horizontal box with two other buttons and some glue. In turn this horizontal box is composed in a vertical box. By selecting the "HBox" entry in the submenu, the user can see the horizontal box composition in a separate view, as shown in Figure 22. The user can now edit the components in the horizontal box without tampering with the rest of the composition. As with all views of the same subject, changes to the composition in either view are reflected immediately in the other.

Multiple views are also instrumental in specifying tray compositions. The user specifies a tray's components by selecting the components and invoking the "Tray" entry in the composition menu. Doing so does not, however, define alignments between the tray's components, which is why one would create a tray in the first place. Such alignments are made by creating a view of the tray, in which its components will be individually selectable, and issuing alignment commands from the Align menu. These commands normally align components by simply moving them, but when the commands are applied to components

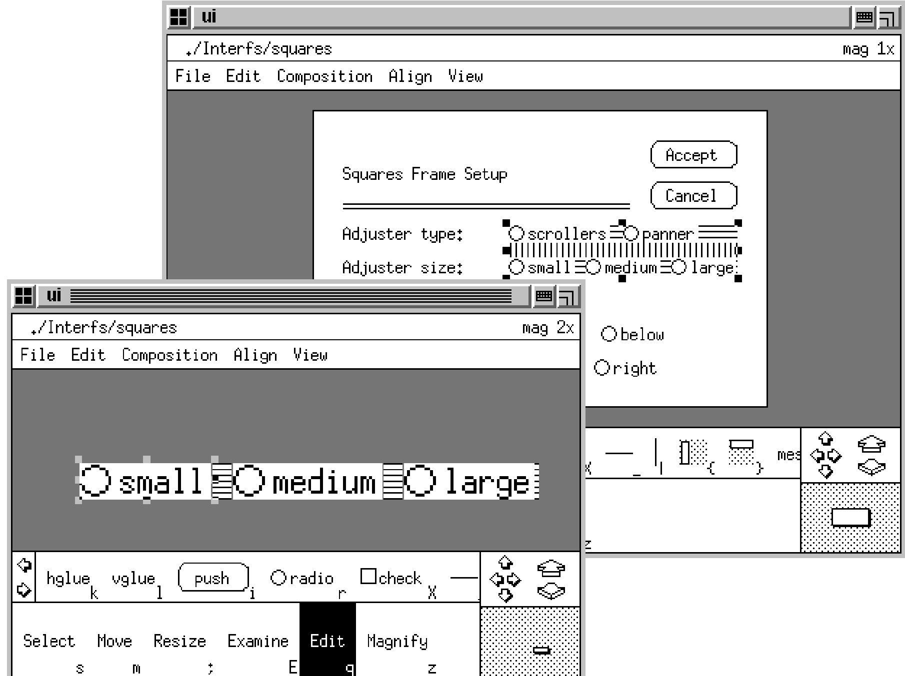


Figure 22: Part of the composition hierarchy presented in a separate view

in a tray, the alignments behave as tray alignments—they persist. Once a tray component is aligned to another, therefore, it stays aligned even if the user attempts to move one of them.

The last tool in the palette, the Edit tool, lets the user change strings as they appear in the interface. It behaves like the Reshape tool in Drawing when reshaping text objects: the user can relabel a button or edit the text in a message by clicking on it and typing the new text.

UI also includes the usual predefined commands for saving and restoring components (**interfaces** in this domain) and editing commands such as cut, copy, paste, delete, and duplicate. In addition, UI defines a command for dissolving compositions (similar to ungrouping in a drawing editor) and for generating the C++ external representation.

5.2.2 Implementation

UI defines many new classes, shown in Table 27. Class names in boldface type are for objects provided by the Unidraw library, while those in italics are defined by InterViews. Table 26 presents a breakdown of the UI implementation in classes and lines of source code.

Like Drawing, UI defines a single editor subclass, **UIEditor**. A new instance of UIEditor is created for each view of an interface. Interfaces are compositions of **InteractorComp**, a subclass of GraphicComps. There is an InteractorComp subclass for each interactor a user can instantiate, each with a corresponding InteractorView.

Interactor Components

InteractorComp adds several operations to the GraphicComps protocol to access state that characterizes the interactor. This state includes state variables for the interactor's class name, canvas, shape, and button state (if any). An InteractorComp also uses a specialized graphic, **UIGraphic**, to define its graphical characteristics, and it provides a GetUIGraphic operation to retrieve this graphic without casting. UIGraphics define their appearance based on information in a canvas state variable, which the interactor component supplies. Most interactor components use a specialized UIGraphic that reflects the interactor's appearance based on the dimensions stored in the canvas variable. By deriving a UIGraphic and redefining its drawing operation, we can specify the interactor's appearance with immediate-mode graphics. This approach has two benefits:

- It is generally easier to render the interactor with immediate-mode graphics. Interactors are simple to draw procedurally, their structure does not change, and the drawing procedure can be parameterized to conform to the canvas.

- Since most interactors use immediate-mode graphics, derived UIGraphic drawing operations can use the corresponding interactor drawing code virtually unchanged.

The interactor component class also defines Reconfig and Resize operations analogous to the corresponding InterViews interactor operations. Conceptually, Reconfig notifies the interactor that the library has computed its final graphics state, upon which its shape may

Editor	UIEditor			
GraphicComps	InteractorComp	HVComp	BorderComp GlueComp ScrollerComp	
		MessageComp	ButtonComp	FBrowserComp StrEditComp
		SceneComp	TrayComp	
			BoxComp	HBoxComp VBoxComp
GraphicViews	InteractorView	HVView	BorderView GlueView ScrollerView	
		MessageView	ButtonView	FBrowserView StrEditView
		SceneView		
Picture	UIGraphic	HVGraphic	BorderGraphic GlueGraphic ScrollerGraphic	
		MessageGraphic	PushButtonGraphic RadioButtonGraphic CheckBoxGraphic StrEditGraphic	
StateVar	**NameVar**	ButtonStateVar		
	CanvasVar ShapeVar			
StateVarView	ButtonStateVarView CanvasVarView ShapeVarView ClassNameVarView			
PreorderView	CodeView	BorderCode ButtonCode GlueCode MessageCode ScrollerCode StrEditCode FBrowserCode BoxCode TrayCode		
Command	CodeCmd InfoCmd NewViewCmd PlaceCmd			
	BrushCmd	GlueVisibilityCmd		
	GroupCmd	SceneCmd		
BasicDialog	InfoDialog			
Tool	ExamineTool			
Manipulator	PopupManip			
Creator	UICreator			

Table 25: UI class hierarchy

		code (lines)	
	classes	*interface*	*implem.*
components	24	500	1870
commands	7	140	360
tools/manipulators	2	60	180
ext. representations	10	170	460
state vars./transf. fns.	7	160	520
editors	1	70	670
creator	1	20	90
toolkit-derived classes	12	230	740
globals	0	100	70
totals	64	1450	4960

Table 26: UI code breakdown

depend. Resize notifies the interactor that its canvas is defined; the interactor may in turn initialize internal state based on the size of the canvas.

Scene Components

The **SceneComp** subclass of InteractorComp provides an abstract base class for scene components. SceneComp subclasses include **BoxComp** and **TrayComp**. Subclasses of BoxComp implement the tiling semantics of horizontal and vertical boxes with the same algorithms that InterViews boxes use, while TrayComp uses connectors to determine the placement of its children. While there is a **SceneView** class for SceneComp, box and tray components have no corresponding SceneView subclass. No view subclasses are necessary because the boxes and trays have no appearance or special direct manipulation semantics of their own, so the SceneView base class can assemble child views for both of them.

Figure 23 shows the connector construction that a tray component creates for each of its children. The tray builds these constructions in its Reconfig operation after it calls Reconfig on its children. The connectors are pins, floating initially, and the glue parameters

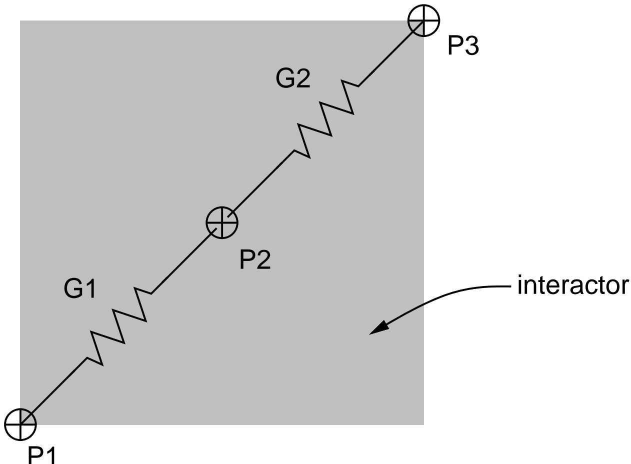


Figure 23: Connector construction for tray children

are based on the child's shape variable, which defines its natural size, stretchability, and shrinkability (in the interactor sense). InterViews makes no distinction between elasticity and deformation limits; they are effectively collapsed into a single parameter per dimension per mode (tension or compression). Therefore, for either G1 or G2,

$$\varepsilon = \lambda = \sigma/2$$

where σ denotes the interactor's horizontal/vertical stretchability/shrinkability, depending on the subscript and superscript we apply to each term.

The tray maintains alignments between its children by establishing connections between the pins in the respective connector constructions. For example, to left-align two children, the tray connects their P1 connectors together with a piece of connector glue having the following parameters:

$$\begin{aligned} h = v = \varepsilon_h = \lambda_h &= 0 \\ \varepsilon_v = \lambda_v &= \infty \end{aligned}$$

(The elasticity and deformation limits are the same in both modes.) These parameters ensure that the children remain left-aligned while remaining able to move freely in the vertical direction.

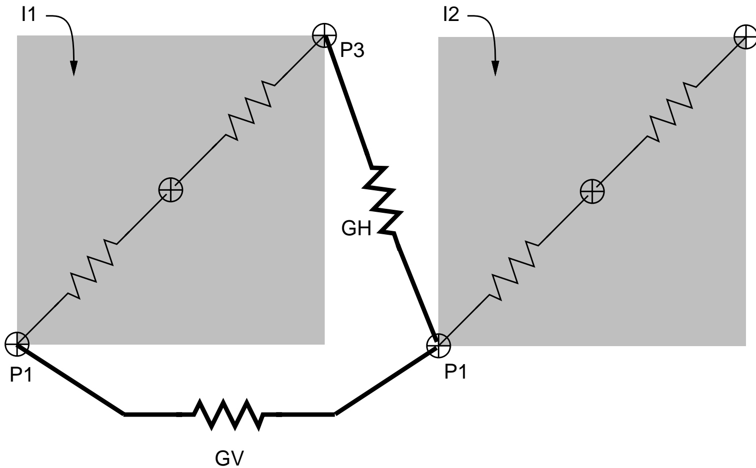


Figure 24: Connections for a two-dimensional tray alignment

Two-dimensional alignments require two connections. For example, a bottom-right to bottom-left alignment between interactors I1 and I2 in Figure 24 requires a connection between their P1 pins and a connection between P3 of I1 and P1 of I2. The P1-P1 connection fixes the relative positions vertically, interposing glue GV that is rigid vertically but infinitely flexible horizontally. The P3-P1 connection fixes the components' relative positions horizontally with glue GH that is rigid horizontally but infinitely flexible vertically.

Once the tray establishes these connections, it must determine its shape. The tray computes its natural size by calling Update on the unidraw object to solve the network and then determining the pins' bounding box. Calculating its stretchability and shrinkability is more difficult, because there is no way to query the Unidraw library for the network equivalent. The tray must ascertain this information from the dynamics of the network.

First the tray fixes the bottom-leftmost pin in the network. To find the shrinkability, the tray connects a fixed pin (call it P^-) to the top-rightmost pin. P^- is centered at $(-\infty, -\infty)$, that is, a large distance below and to the left of the bottom-leftmost pin. Then the tray calls Update again to solve the network. The tray's shrinkability is reflected in the distance the top-rightmost pin moved to maintain its connection with P^-. Similarly, the tray computes its stretchability by connecting the top-rightmost pin to a fixed pin P^+ centered at $(+\infty, +\infty)$ and re-solving the network; the stretchability is the amount the top-rightmost pin moved to remain connected to P^+.

With all shape information calculated, the tray can set its shape state variable and return from Reconfig. When the tray is later resized, it translates the top-rightmost pin so that the bounding box of the tray's pins matches the size of the canvas. The tray then solves the network and translates and resizes its children to reflect the positions of the pins in its constructions.

State Variables

UI defines state variables for interactor attributes that a user can examine, specify, or modify. These include its canvas, shape, class name, and possibly a button state. The views of these state variables are assembled into a dialog box when the user examines the interactor.

CanvasVar stores the width and height of an interactor component's canvas. **CanvasVarView** displays this information in an InterViews **Message** object (an interactor that displays a non-editable string of text).

ShapeVar defines an interactor component's shape. The shape information can be editable or read-only, depending on whether the interactor's programming interface allows the programmer to specify or change the interactor's shape. **ShapeVarView** presents this information in a tabular format incorporating string editors (interactors that display an editable string of text) if the subject is editable; otherwise it incorporates messages exclusively.

ButtonStateVar stores a name, an initial value, and a setting value. All button state variables with a given name share the same initial value. The external representation for an interface will contain an instantiation of a button state object for each unique name among the button state variables. **ButtonStateVarView** provides an interface to changing the subject's name and values.

Finally, interactor components store their class name in a NameVar, a library-defined state variable that stores a string. UI defines a **ClassNameVarView** state variable view that simply presents this name in a message, since it is an unchanging attribute of an interactor.

External Representation

The **CodeView** subclass of PreorderView is the base class for objects that generate UI's external representation. Each interactor component has a corresponding CodeView subclass. Figure 25 lists a sample UI external representation.

CodeView's Emit operation first generates forward declarations for all button states in the interface. The ButtonStateVar class defines a class operation that allows CodeView to iterate through all the ButtonStateVar instances in the interface and produce the ButtonState declarations at the top of Figure 25. CodeView subclasses for leaf interactor components redefine the Definition operation to generate code of the form

`new` *className* `(` *parameters* `)`

where the parameters reflect the component's internal state. **BoxCode**, the CodeView subclass for box components, generates these parameters by calling its children's Definition operations. The tray external representation must be handled specially, since its alignments cannot be specified in-line.

To generate tray code, CodeView's Emit operation performs two traversals of the external view hierarchy. The CodeView class defines a class member variable `boolean _trays`. CodeView subclasses check this value before they produce code. If `_trays` is true, then only TrayCode instances generate output—a function returning an initialized tray instance; otherwise, each CodeView subclass generates code as described above, and TrayCode generates a function call.

A programmer uses the Interior function to produce an instance of the interactor composition. CodeView's Emit operation generates the Interior declaration. The button state initialization code is created by iterating through the ButtonStateVar instances as CodeView did to generate the corresponding declarations. CodeView produces the Interior function's return value by calling its own Definition operation.

Commands and Tools

UI defines several commands. **InfoCmd** creates an instance of an **InfoDialog**, which composes state variable views of an interactor component's state variables. A user invokes

```
static ButtonState* bs0;
static ButtonState* bs1;

static Tray* Tray0 () {
    Tray* tray = new Tray;

    Interactor* i1 = new CheckBox("check", bs0, 1, 0);
    tray->Insert(i1);
    Interactor* i2 = new HGlue(0, 0, hfil);
    tray->Insert(i2);
    Interactor* i3 = new PushButton("push", bs1, 1);
    tray->Insert(i3);

    tray->HBox(tray, i1, i2, i3, tray);
    tray->Align(VertCenter, i1, i2, i3);

    return tray;
}

static Interactor* Interior0 () {
    bs0 = new ButtonState(0);
    bs1 = new ButtonState(0);

    return new Frame(
        new VBox(
            new VGlue(20, 20, vfil),
            new StringEditor(bs0, "sample"),
            new VGlue(20, 20, vfil),
            Tray0(),
            new VGlue(20, 20, vfil)
        )
    );
}
```

Figure 25: Sample UI external representation

this command through the examine tool's pop-up menu. Once the user has changed the state variable values and dismissed the dialog, InfoCmd applies the changes to the selected interactor component. UI's **NewViewCmd** is analogous to Drawing's: it creates a new UIEditor instance for editing the same component as the current editor. The user directs UI to produce an external representation with the **CodeCmd**. This command lets the user specify a file name in which to store the external representation and then creates a CodeView of the editor's interactor component.

The InterViews scene class defines a Place operation that makes an interactor hierarchy visible on the screen. This operation initiates two traversals of the interactor hierarchy: one to determine its shape, and another to allocate screen space. InterViews calls Place when an interactor is inserted into the top-level window; in UI this occurs when an interactor component is created with a graphical component tool. The interactor view base class creates a **PlaceCmd** in response to manipulation by a graphical component tool. The interactor subject initiates the placement traversals when it interprets this command.

The **GlueVisibilityCmd** directs a glue component to make itself visible (by rendering itself with horizontal or vertical bars) or invisible (by erasing the bars). Visible glue is easier to manipulate, while invisible glue is less distracting. Lastly, **SceneCmd** is a subclass of the library-defined **GroupCmd**, which removes selected components from their parent and inserts them into another graphical component (the **destination**), which in UI will be a scene component. SceneCmd extends the behavior of GroupCmd to make the destination interpret a PlaceCmd after reparenting the components, allowing the destination to determine their shape and allocate their screen space.

UI defines only one new tool, **ExamineTool**. Other tools in the interface are taken from the library. In particular, the edit tool is actually an instance of the library's ReshapeTool, which delegates manipulator creation and interpretation to the component(s) being manipulated. Views of interactor components that can be edited with this tool simply redefine their CreateManipulator and InterpretManipulator operations to respond to it appropriately—a new tool subclass is therefore unnecessary. Similarly, UI's ResizeTool is actually an instance of the library's StretchTool.

ExamineTool produces an InterViews pop-up menu with two submenus containing controls that execute InfoCmds and NewViewCmds, respectively. Each of these commands

operates on a different level of the chosen interactor hierarchy. ExamineTool does not delegate manipulator creation to the selected interactor view; instead, it creates a **PopupManip** with the pop-up menu as an argument, and it relies on the pop-up menu's submenus to execute the desired command. The PopupManip does little more than forward events to the pop-up menu to allow it to work within the viewer.

5.2.3 Experience

The user interface building domain is the least mature of the domains represented by our three experimental applications. Graphical object editors for this domain did not arrive until the mid-1980s, and little has been published relating experience with them in production environments. There is a consensus, however, that they do not eliminate the need for conventional programming, and UI is no exception in this regard. UI is best used to describe the layout of an interface and for experimenting with the dynamics of interactor compositions. UI does not help the user specify an interface's input semantics, nor does it let him define new interactors, but neither do current hand-crafted interface builders offer such capabilities.

On the other hand, UI does support layout semantics as sophisticated as any user interface builder. The composition abstractions that underlie existing builders are not as general or powerful as those that InterViews provides, which include non-linear deformation (independent stretchability and shrinkability) with two-way layout constraints (via boxes and tray). UI can offer a better direct manipulation model because it is founded upon more powerful InterViews abstractions.

In performance terms, interfaces as they appear and behave in UI actually outperform real InterViews interfaces. Interactor components consume less memory than real interactors, partly because interactor components do not handle input and partly because each InterViews interactor contains its own X window. The smallest possible InterViews interactor (not including information potentially shared with other interactors) uses about 100 bytes. This figure does not include the X window, which takes up space in both the application and in the window server. In X11 Release 4 the library space is at least 75 bytes, while the server keeps at least an additional 150 bytes. Each InterViews interactor

thus consumes at least 325 bytes. The smallest possible interactor component, including a subject and a view, is 188 bytes.

Interactor components also compare favorably at run-time. Interactor views draw themselves as quickly as real interactors do, and scene components compose and place their children substantially faster than real scenes, since placement does not involve communication with the X server. Tray components are even more efficient in comparison. Tray subjects use connectors to determine the proper placement of their components, so they work in time proportional to $\mathcal{O}(n^2)$ at worst, where n is the number of alignments. InterViews trays use a similar recursive substitution algorithm to compute placement, but they do not incorporate any of the optimizations discussed in Section 4.6.5 to make the search for primitive combinations run in linear time. For example, the search for parallel connections is $\mathcal{O}(n^2)$. Thus InterViews trays take $\mathcal{O}(n^3)$ time at worst.

The upshot, then, is that interfaces in UI perform at least as well as the InterViews-based interfaces they implement. However, the code that UI generates could be better. Usually some modification is required to integrate the code into a larger implementation. The generated interactor instance names are not mnemonic; the user should be able to specify these names at run-time through a state variable view. The programmer will often want the Interior function to be a member of a class rather than a static global. This is true for the button states as well. Still, the external representation does encapsulate the trickiest part of the implementation: specifying the composition hierarchy and interactor attributes correctly.

5.3 Schematic Capture System

Schematic capture systems belong to a class of CAD tools that assist in the design and analysis of electrical circuits. Most schematic capture systems support interactive specification of digital circuits. These systems simplify the design process by letting engineers create, view, and edit a schematic representation of the circuit, from which the system generates a netlist representation. The netlist catalogs the interconnections between the elements in the circuit, information that uniquely defines the circuit's topology. With this netlist information, the circuit can be simulated, analyzed, and ultimately realized in wire

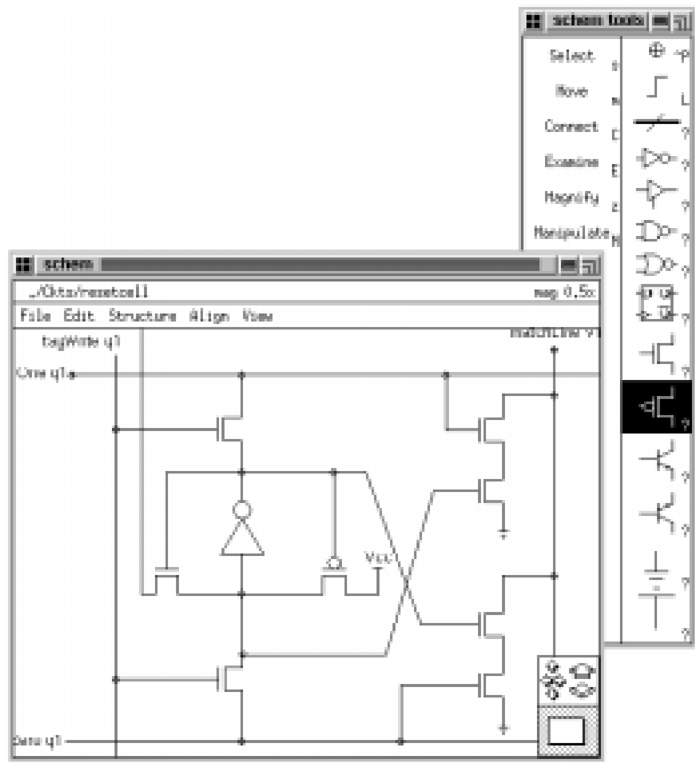


Figure 26: Schematic capture system

and silicon. The schematic capture system is the starting point of the entire circuit-making process.

Our experimental schematic capture system is called **Schem**. It supports hierarchical circuit specification and generates both hierarchical and flattened netlist external representations. In addition to these traditional schematic capture capabilities, Schem takes advantage of Unidraw's advanced features to provide a direct manipulation-oriented interface, connectivity maintenance, full-featured 2-D graphics, and multi-view editing. Schem also uses dataflow to support combinational logic simulation, a feature not normally associated with schematic capture systems.

Schem is also the most extensible of the three experimental domain-specific editors in that the user can extend its repertoire of components at run-time. He can create new **elements**, as we normally refer to components in this domain, define their appearance,

connectivity semantics, and logical functions. Schem thus caters to at least four types of users:

1. System administrators concerned with creating and maintaining a library of elements for others' use.
2. Circuit designers that are constrained to use only predefined elements.
3. Circuit designers that are free to create custom elements for use in their designs.
4. Students interested in learning about logic design in a simulated laboratory.

5.3.1 User Interface

Schem displays two windows initially, as shown in Figure 26. The larger of the two contains a viewer in which to build schematics, while the smaller window contains a palette of tools for creating and manipulating schematic elements. We will refer to these windows as the **Schem editor** and **tool palette**, respectively. Like the other experimental editors, the Schem editor provides a set of state variable views and pull-down menus above its viewer. The tool palette has controls for engaging graphical component tools along its right side and tools for manipulating components along the left. Unlike Drawing and UI, Schem maintains a single, global selection; selecting a component in one editor will unselect a previously selected component in another.

Elements and Nodes

Schem defines two basic types of components: **elements** and **nodes**. An element corresponds to an electronic part, while a node is a point of connection in the circuit. Elements usually have at least one node and may contain subelements, while a node is an atomic entity. Both elements and nodes are named to identify themselves in the netlist. For each element, the netlist records its name and the names of its nodes; for each node in the element the netlist lists the names of nodes connected to it (if any) and the elements to which they belong. A netlist organized in this way is **element-based**; netlists in which the connections are listed by node rather than element are **node-based**.

In addition to a name, nodes have an associated **logic level**: either **zero**, **one**, or **don't care**, and they can serve in an input, output, or bi-directional capacity. These attributes allow nodes to represent the terminals of logic elements. Elements can define a truth table relating the logic levels of their input nodes to those of their output nodes, thus enabling combinational logic simulation.

Schem defines a special element called an **alias** to support hierarchical circuit specification. An alias is an alternate representation for another element, which we call the alias' **definition**. One can use an alias to replace a complex circuit with a "black box" representation, thereby abstracting the circuit. By default, an alias inherits copies of any independent nodes in the definition, and it maintains a link to the definition, but its appearance is defined by the user. Once created, an alias can be used as any other element; it can even serve as part of the definition of another alias. An alias thus represents an interior node in a hierarchically structured circuit.

Tool Palette

Schem incorporates the Select, Move, and Connect tools from the Unidraw library. Select and Move have been discussed previously. The Connect tool provides a direct manipulation interface to connecting connectors. For example, clicking on a pin with the Connect tool engaged allows the user to drag the pin around. If the user drags the pin near another pin (the **target**), the pin being dragged will jump towards the target so that their centers coincide; if the user then releases the mouse button, the Connect tool will connect the two pins. If instead the user continues to drag the pin a certain distance beyond the target, it will snap away from the target and resume following the mouse position. In Schem, the user relies on the Connect tool to connect nodes, which are derived from pins.

Schem defines two other tools: Manipulate and Examine. The Manipulate tool lets the user manipulate components in a life-like manner. Currently only the switch component (described below) responds to this tool. Schem's Examine tool is similar to UI's in that it lets a user examine attributes of a particular component.

Clicking on a component with the Examine tool engaged produces a pop-up menu of options, as shown in Figure 27. The "Create Alias" entry creates a new Schem editor containing an alias of the chosen element. The user may then edit the alias and store it for

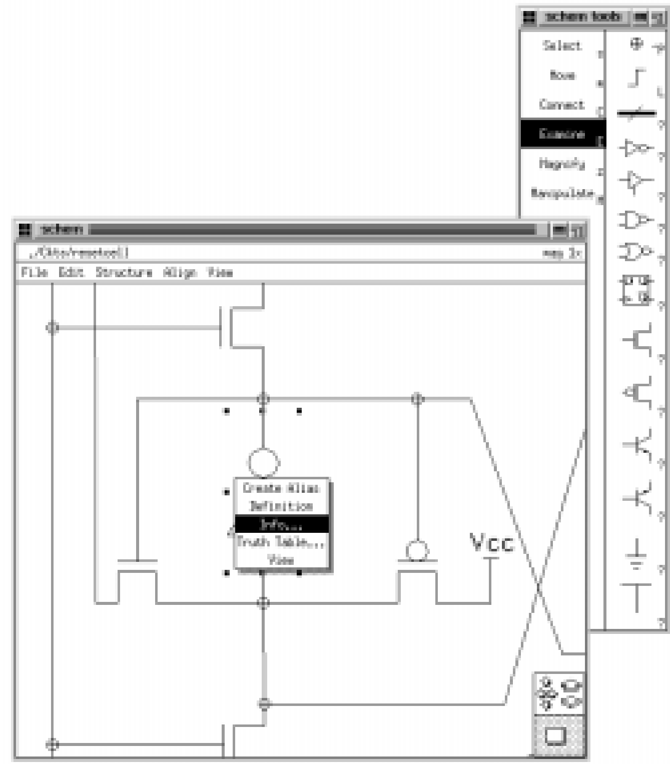


Figure 27: Using the Examine tool on an inverter element

later use. If the chosen component is an alias, then invoking the "Definition" entry creates a new Schem editor containing the alias's definition. The "Info..." entry creates a dialog box for editing the component's internal state. If the component is a node, then the dialog box lets the user change its name and transmission semantics (whether it is an input, output, or bidirectional node); if the component is an element, the dialog lets the user edit its name and the names and transmission semantics of its nodes. The "Truth Table..." entry posts a dialog box that lets the user specify the output logic level values for every combination of input levels. Finally, the "View" entry creates a new Schem editor with a new view of the chosen component, provided it is an element.

Initially, Schem includes graphical component tools for creating four components it predefines:

1. The node component represents an attachment point as described above.

2. The **wire** component connects two nodes visually. Wires can be manhattan or sloped and can have any number of discontinuities.

3. The **bulb** component simulates a light bulb. When connected to a node with logic level zero, the bulb appears dark; when the logic level is one, the bulb lights up.

4. The **switch** component will affect the logic level of any nodes to which it is connected. When the switch is in its down position, it sets the logic level to zero; when it is up, it sets the logic level to one. The user can toggle the switch up and down with the Manipulate tool.

None of these components are elements. The schematic that a user creates in a Schem editor is an element, as are aliases. With these default components and tools for creating them, the user can wire up nodes, bulbs, and switches, but nothing more. To create practical schematics, the user must define new elements.

Defining a New Element

Creating a new element involves three steps:

1. Define its appearance.

2. Define its semantics.

3. Store it for later use.

We should also make it easy for users to insert copies of the element into their semantics. Therefore we need a graphical component tool with the element as its prototype and a control in the tool palette for engaging the tool.

Figures 28 through 33 depict the procedure for creating a prototypical NAND gate component, a corresponding graphical component tool, and a control for engaging the tool. The first step, shown in Figure 28, is to define the element's appearance. Schem provides a full complement of components, commands, and tools for producing graphics. These features are accessed through a window called the **drawing tool palette**, which is not normally visible. To make this palette visible, the user issues the "Drawing Tools..."

command from the Options menu. Schem provides drawing facilities almost as powerful as Drawing's, since the components, commands, and tools for basic drawing are predefined by the Unidraw library. A closed B-spline is used for the AND symbol, a circle for the inversion bubble, and three lines for the gate's input and output leads.

The next step is to define the gate's semantics, which involve connectivity, netlist generation, and combinational logic simulation. We want to connect wires to the ends of the gate's leads and have them stay connected when the gate is moved. To accomplish this we place node components at the ends of each lead (Figure 29). As connectors, the nodes will enforce the connectivity semantics we desire. Next we give the element the name it will use in the netlist, identify its nodes with mutually-unique names, and set the transmission semantics for each node (Figure 30). Schem assigns each element in the netlist a unique integer value and appends it to the user-specified name to form the element's full netlist name. The integer value is unique across element instances, thereby ensuring that all elements in the netlist can be identified even though some may share a user-specified name. Whenever an element references another element's node, the reference appears in the netlist as the concatenation of the name of the enclosing element followed by the node's name. Finally, we define the truth table for the element (Figure 31).

Having specified the NAND gate's semantics, we now store it for later use using the "Save As..." command in the Schem editor's File menu. A dialog box prompts for the name of a file in which to store the gate; we will use the name "nand.proto" in this example. Now if we want to easily incorporate copies of the NAND gate into schematics, we will need a graphical component tool. The "New Tool..." command in the File menu lets us choose an element to use as the prototype for a new graphical component tool (Figure 32). We simply type in the name we used to save the gate ("nand.proto") and press the "OK" button. Schem then creates a graphical component tool with the specified element as its prototype, creates a control for engaging the tool, saves the tool under the name "nand.proto.tool," and installs the control in the tool palette. Now we can create NAND gates just as easily as any other component (Figure 33).

We can also remove and re-install graphical component tools at run-time. The "Tools..." command in the File menu posts a dialog box listing the names of all installed graphical component tools, and it allows us to type in the name of existing but uninstalled tools.

Selecting an installed tool and pressing the "Install/Remove" button removes the control associated with the tool from the tool palette, while typing in the name of a presently uninstalled tool will install a control for engaging it. This level of customizability contrasts with the less dynamic but more flexible capabilities of Drawing. Schem stores the names of the installed tools in a special file so that the same tools are installed when the system is restarted.

5.3.2 Implementation

Table 27 lists the classes that Schem implements. As before, class names in boldface type are defined by the Unidraw library, and names in italics are defined by the InterViews library. Table 28 presents a breakdown of the Schem implementation in classes and lines of source code.

Editors

Schem defines three subclasses of editor: **SchemEditor**, **ToolPalette**, and **DrawingPalette**. SchemEditor implements the Schem editor windows in which the user creates schematics. The SchemEditor incorporates a viewer and commands but no tools. The current tool is defined by the ToolPalette, which is a degenerate editor in that it contains no viewer. The drawing tool palette is implemented by the DrawingPalette editor, which is similarly degenerate. Schem produces a single instance of ToolPalette and DrawingPalette and at least one instance of SchemEditor. The tool palette is created first, for it defines the current tool and the common selection object. The DrawingPalette constructor takes the tool palette instance as an argument, and SchemEditor instances require both the tool and drawing palette instances as arguments. Thus all editors have the same notion of the current tool and selection.

Components and their Semantics

ElementComp and **ElementView** define the subject and view protocols for all netlist-generating and user-defined elements. ElementComps maintains an **ElementVar**, a state variable that identifies the element by name in the netlist. **ElementVarView** defines a

Editor	SchemEditor ToolPalette DrawingPalette		
GraphicComps	ElementComp	AliasComp	
GraphicViews	ElementView	AliasView	
PinComp **PinView**	NodeComp NodeView		
NameVar	ElementVar NodeVar		
StateVarView	ElementVarView NodeVarView		
TruthTable			
TransferFunct *TextEditor*	TF_TruthTable TruthTableEditor		
GraphicComp	WireComp		
	LogicComp	BulbComp SwitchComp	
GraphicView	WireView		
	LogicView	BulbView SwitchView	
PreorderView	NetlistView	NLNode NLGraphicComps	
		NLElement	NLAlias
Command	AliasCmd DefinitionCmd NewViewCmd ToggleDwgPaletteCmd ToolsCmd NetlistCmd InfoCmd TruthTableCmd		
	ViewCompCmd	NewToolCmd	
FileChooser	NetlistDialog		
BasicDialog	InfoDialog TruthTableDialog		
Tool	ExamineTool ManipulateTool		
Manipulator	PopupManip WireManip		
GrowingVertices	RubberWire		
Creator	SchemCreator		

Table 27: Schem class hierarchy

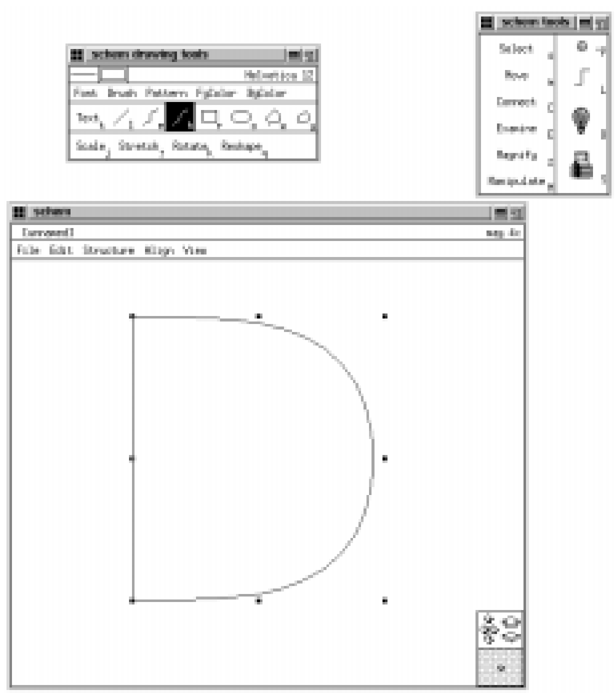


Figure 28: Specifying a NAND gate's appearance

		code (lines)	
	classes	*interface*	*implem.*
components	14	270	1100
commands	9	190	540
tools/manipulators	5	90	230
ext. representations	5	120	380
state vars./transf. fns.	5	140	300
editors	3	170	1140
creator	1	20	60
toolkit-derived classes	6	170	810
globals	0	200	130
totals	48	1370	4690

Table 28: Schem code breakdown

view that displays the subject's unique name. Aliases are instances of the **AliasComp** and **AliasView** classes, which are derived from ElementComp and ElementView, respectively. AliasComps keep a pointer to their definition, which is an ElementComp.

Nodes are represented by instances of **NodeComp** and **NodeView**, which are subclasses of PinComp and PinView, respectively. When instantiated, a NodeComp creates an instance of a **NodeVar** state variable and binds it to itself. The NodeVar stores a name (being derived from NameVar) and a logic level value. A NodeVarView presents an interface for editing the name and for choosing the node's transmission method (see Figure 30).

ElementComps also define a **TF_TruthTable** transfer function that specifies the combinational logic relationships between its nodes. TF_TruthTable stores the truth table itself in a separate **TruthTable** object, and it defines dependencies between the node variables based on the TruthTable object's contents. The user edits an element's truth table using a **TruthTableEditor**, a specialization of the InterViews **TextEditor** object. Any change in the truth table object affects the element's transfer function, since the transfer function relies on the information in the truth table.

Wires, switches, and bulbs are not derived from ElementComp and ElementView because they are not represented in the netlist. The wire component subject **WireComp** and view **WireView** are derived directly from GraphicComp and GraphicView. A WireComp contains two NodeComps to define its endpoints, and it uses a MultiLine structured graphic object to represent the path between them. It relates the two nodes' state variables with a TF_Direct transfer function. The bulb and switch component subjects and views are derived from **LogicComp** and **LogicView**, which are in turn derived from GraphicComp and GraphicView. LogicComps include a NodeComp to define a logic value and connectivity semantics. SwitchComp changes the value of its node's NodeVar to reflect the switch position, while BulbComp bases its appearance on the NodeVar's value.

Schem defines a **NetlistView** that generates its netlist external representation. NetlistViews take a parameter that specifies whether the netlist should be flattened or not. The **NLNode**, **NLElement**, and **NLAlias** subclasses generate the information appropriate for nodes, elements, and aliases, respectively. The **NLGraphicComps** subclass acts as a placeholder for elements that have been grouped to simplify their graphical manipulation; the netlist should not reflect such grouping.

Commands

Schem's **AliasCmd** instantiates an alias for a given ElementComp and creates a new SchemEditor for editing it, while the **DefinitionCmd** creates a new SchemEditor for editing the definition of a given component. The DefinitionCmd reports an error via a dialog box if the component is not an alias and therefore has no definition. **ToggleDwgPaletteCmd** either shows or hides the drawing palette.

ToolsCmd posts a dialog box giving the names of all the installed tools as reported by the catalog, letting the user add new tools or remove existing ones. **NetlistCmd** posts a **NetlistDialog**, which prompts the user for a netlist name for the current schematic and whether the netlist should be flattened or not. It then creates the corresponding netlist view and emits the netlist. Given a component, **InfoCmd** creates an instance of an **InfoDialog**, which provides element and node variable views for changing the corresponding attributes of the subject, if any; otherwise the command posts a dialog box reporting that the component has no schematic information. **TruthTableCmd** posts a **TruthTableDialog** containing a TruthTableEditor for the given component, if it defines a truth table transfer function. Finally, **NewToolCmd** prompts the user for a component name, and it uses the corresponding component to create a graphic component tool. It then inserts a control into the tool palette for engaging the tool.

Tools

The next five classes in the table support Schem's direct manipulation semantics. The **ExamineTool** uses a **PopupManip** in the same way as the corresponding UI classes, except the pop-up menu produced by Schem's ExamineTool does not have submenus for specifying the level in the hierarchy to be examined.

The **ManipulateTool** delegates manipulator creation to the component view. The only component that responds accordingly is the switch, but it does not actually produce a manipulator. Instead it responds by toggling the logic level of its subject's node state variable. Thus the switch throws immediately when the user clicks on it with the manipulate tool, and the operation is not undoable. An alternative interface could have the switch view generate a manipulator that animates the switch's sliding action, from which the view

generates a command that changes the state variable. Apart from making the switch work more realistically, this implementation would also make the action undoable.

Two classes support the special creation semantics of wire components. The user can create wires with any number of discontinuities, and each segment can be either sloped or manhattan depending on whether the Shift key is held down. The rubberbanding effect during the wire's creation is unique as well; the wire deforms dynamically as jogs are added. The **WireManip** implements these manipulation semantics, and it uses a special rubberband, a **RubberWire** to perform the animation.

5.3.3 Experience

By definition, all schematic capture systems support graphical circuit specification and netlist generation. Most allow designers to specify circuits hierarchically. Some systems maintain graphical connectivity automatically; others store connectivity information but do not enforce it in the graphical representation.

Schem provides these features and adds combinational logic simulation, multiple views, and nearly all the features of dedicated drawing editors, while most schematic capture systems have limited drawing capabilities, if any. Some schematic capture systems offer features that we have omitted from Schem, such as iterative circuit specification, element instancing, a bus component abstraction, and the ability to read an externally modified netlist. The first three capabilities are straightforward to add; however, Unidraw does not currently support internalization of external representations. Providing such a capability in general will take further study, but Unidraw does not prevent an application from reading a particular external representation. In fact, we have successfully extended Drawing to read the PostScript external representation it generates.

It is easier to compare functionality than implementation effort, because we cannot examine the source code or development history of commercial systems. But we do know that idraw requires nearly three times the code in Schem, and idraw's code does not include its graphics capabilities, which InterViews supplies. Factoring out the graphics code in this manner, a schematic capture system's capabilities become roughly a superset of a drawing

editor's. Hence we expect that a conventional schematic capture system would require at least as much code as idraw.

Schem's graphics performance is identical to Drawing's, but Schem's schematic components are significantly larger than those of many other schematic capture systems. The premium arises partly out of the subject/view split, where both contain a graphic. InterViews structured graphics have a minimum size of 24 bytes and average around 40, so each schematic component consumes about 80 bytes for graphics alone. This overhead is a consequence of the way we designed Schem and the predefined components, and it could be reduced by optimization or by foregoing multiple views. Another expensive feature is Schem's support for run-time component specification. Each user-defined element has an appearance made up of graphical components, each comprising a subject and a view. Thus even elements with simple graphical designs become bulky. Again, this expense could be mitigated by eliminating the run-time extension feature or by optimization. For example, the graphical component composition that makes up an alias's appearance could be discarded after the alias is defined, retaining only the composite graphic. An instancing component could lessen the impact of expensive components by amortizing the cost of a prototype over many instances. Such techniques would make Schem more space-competitive with conventional systems.

5.4 Summary

Drawing, UI, and Schem demonstrate how Unidraw facilitates the design and implementation of domain-specific editors. Each of these experimental applications reflects a significant reduction in development effort by almost any measure, and all approach or exceed the functionality and performance of their custom-built counterparts.

Each application has features that set it apart from the others, thereby offering different perspectives of Unidraw's capabilities. Drawing is an example of a simple Unidraw-based graphical object editor, demonstrating the power of the predefined library components. Drawing also shows how a Unidraw-based application can give the user considerable control over the application's user interface. UI's components have the most complex

semantics; its tray component in particular shows how Unidraw's connector model can support sophisticated connectivity. UI also takes greatest advantage of multiple views. Schem demonstrates dataflow and shows how new components can be defined dynamically. Together, these applications attest to Unidraw's effectiveness in simplifying the development of practical graphical object editors.

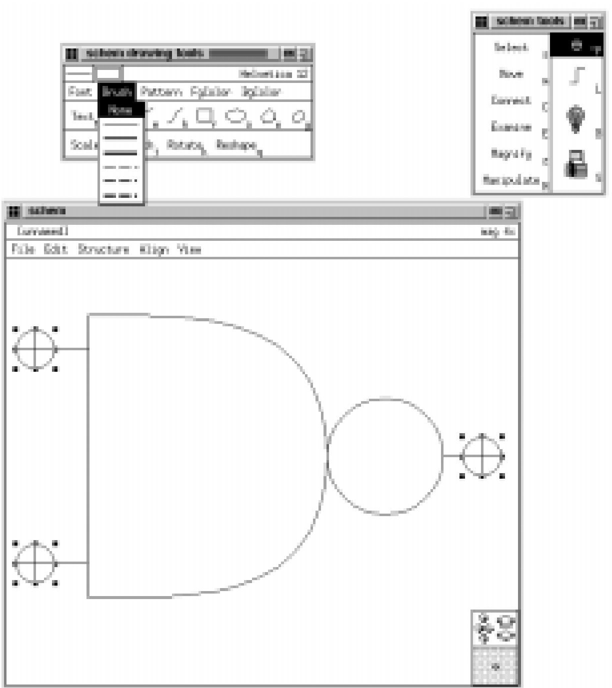


Figure 29: Using nodes to define connectivity, logic simulation, and netlist semantics

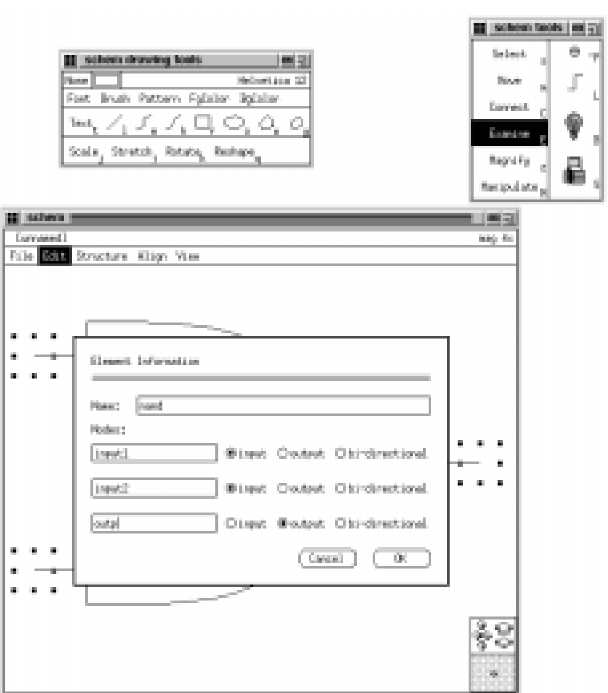


Figure 30: Specifying element and node names and transmission methods

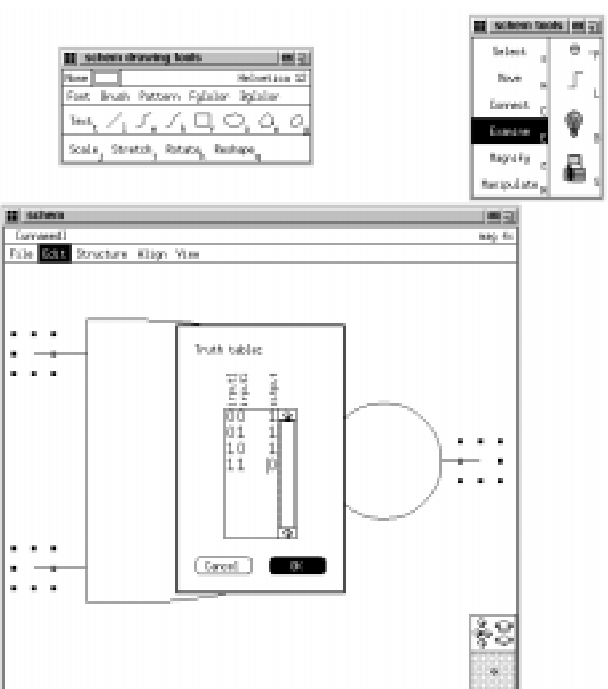


Figure 31: Defining the truth table for the NAND operation

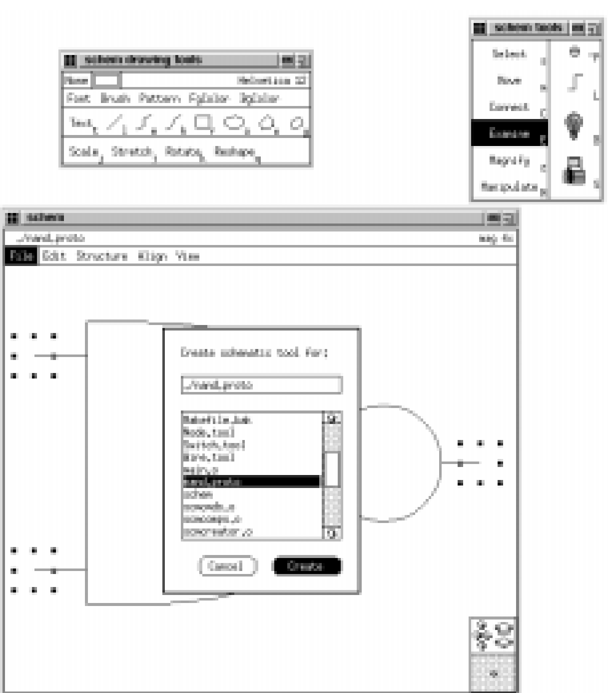


Figure 32: Creating a graphical component tool for instantiating new NAND gates

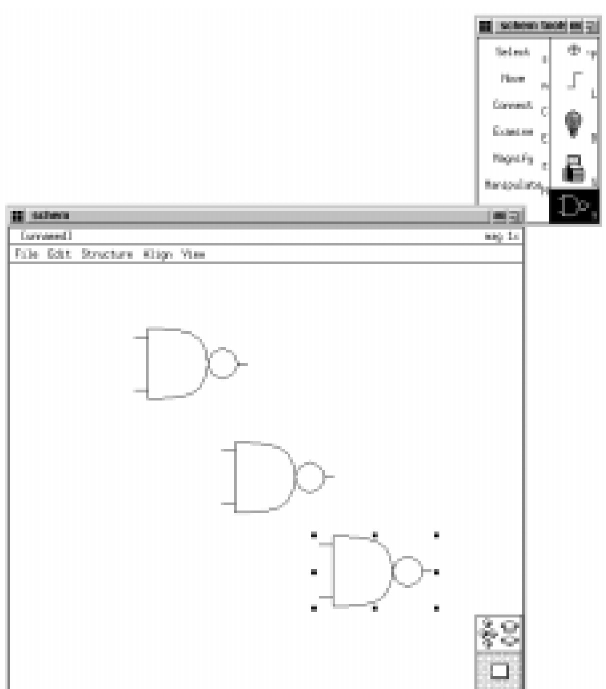


Figure 33: Creating NAND gates with the newly defined tool

Chapter 6

Conclusion

We began this work by identifying an important class of applications, graphical object editors, that had remained difficult to design and build despite advances in user interface technology. This problem prompted our hypothesis that a tool embodying the right abstractions would simplify graphical object editor development. The hypothesis in turn gave rise to an experiment, in which we formulated and specified such a tool, implemented it, and tested it to verify our reasoning.

6.1 Summary of Work and Contributions

Unidraw is first and foremost an architecture for graphical object editing. We designed the architecture to fulfill the needs—unforeseen and anticipated—of graphical object editors. Our design distills the common aspects of these applications into a set of abstractions that give us leverage on the problem. *Components* encapsulate the appearance and semantics of objects the user manipulates. *Commands* define operations on components and other objects. *Tools* support direct manipulation of components. *External representations* define the mapping between components and the information an editor generates for external consumption. The architecture also defines abstractions for specifying common component semantics: *connectors* support connectivity and confinement between components and, together with *state variables* and *transfer functions*, support dataflow between them. Finally,

the architecture defines a framework for assembling these elements into a complete application. This architecture is grounded on prior experience with real applications, systematic analysis of the problem, and extensive experimentation.

Unidraw is also a prototype implementation of the architecture that demonstrates the efficacy of the design. With few exceptions, the prototype implements the architecture as specified, and it furnishes a variety of predefined elements with which programmers can build applications. The prototype has established that the architecture is realizable.

Finally, Unidraw is a test of the concept of generalized graphical object editing. We used the prototype implementation to build three experimental domain-specific editors, applications complex and different enough to demonstrate the architecture's applicability, flexibility, and power. All seemed daunting to implement at the outset of this work; in the end their development seemed simple and almost routine. The drawing editor provided the most dramatic example of how much code is saved when one exploits the predefined components. The user interface builder and the schematic capture system showed that more specialized editors require less code by virtue of Unidraw's architectural abstractions, even when few of their features are predefined.

In summary, the contributions of this work are:

1. Identification and characterization of graphical object editors.

2. An original architecture for graphical object editing.

3. A prototype implementation that demonstrates the architecture's realizability.

4. Experimental evidence that the architecture simplifies the implementation of practical graphical object editors.

6.2 Observations

One problem with object-oriented design is that it takes considerable experience, forethought, and experimentation to factor a system into a good set of objects and protocols. Programmers new to object-oriented programming in particular have a difficult time just

knowing where to start in a design task. And even with such experience, building a graphical object editor still involves a major design and implementation effort.

The Unidraw architecture simplifies the design task by reducing the number of decisions a programmer must make. The architecture has settled many design issues before the programmer starts. A potential drawback of such an architecture is that it may place restrictions on what the programmer or application can do; however, we did not find this to be a problem as we built our experimental applications. The architecture let us do what we wanted to do in these applications, and we never found ourselves changing the architecture to suit an application. This result has reassured us that the basic Unidraw model is sound.

Moreover, the prototype Unidraw library simplified the implementation task by providing reusable functionality. Basic geometric components and their PostScript external representations, commands for carrying out catalog operations and graphical transformations, tools for creating and selecting components—these and many of the other predefined objects proved useful in all three applications. A welcome side-effect of predefined objects is that debugging time is reduced, both because we wrote less new code and because the library was robust enough that we could limit bug searches to application code.

6.3 Future Work

Several aspects of this work merit further investigation.

6.3.1 Stricter Adherence to the Architectural Specification

The prototype implementation does not support two aspects of the architectural specification: non-orthonormal slots and pads and truly non-linear glue. Both limitations stem from the csolver model, in which horizontal and vertical placement constraints are solved independently and connection networks are solved by recursive substitution, which presumes a linear system. While these simplifications have been adequate, exploring how to efficiently support the more general capabilities should prove worthwhile.

6.3.2 Library Optimizations

Two library optimizations could improve the performance of applications in which users routinely edit several thousand components or more. The first is to reduce the size of graphical components. A substantial savings would accrue just by shrinking InterViews structured graphics objects. The second optimization would reduce csolver's running time to expedite connector network solution in applications that produce few disjoint networks. Reducing the algorithm from $\mathcal{O}(n^2)$ to $\mathcal{O}(n \log n)$ could yield a noticeable speedup in networks with hundreds of mutually-dependent connections.

6.3.3 Architectural Extensions

The architecture only addresses external representation *generation*. We would like to go beyond the current predefined external view traversals to develop a more powerful model that supports external representation *interpretation*. While Unidraw currently does not preclude such interpretation, neither does it aid its implementation. This capability would let a domain-specific editor read existing representations, including those not generated by the editor itself. For example, a schematic editor could read an existing netlist, allow the user to edit it graphically, and generate a new netlist. A logic simulator could then give the user feedback about the modified circuit's behavior, which might prompt him to edit the circuit again. Reading and writing external representations permits iterative design by closing the loop between specification and analysis.

Another useful architectural extension would support automatic component layout. Often in applications such as tree or graph editors the user is not interested in arranging components by hand; instead he would rather specify rules for their placement and let the system enforce them. Later he might tidy up the system's layout via direct manipulation, but the bulk of the work will have already been done. The architecture could include an object that, like csolver, positions components according to a specification. Such an object could use established layout algorithms and heuristics to produce pleasing component layouts.

6.3.4 Additional Experiments

Metrics based on the experiences of other Unidraw programmers would shed more light on the system's benefits. We envision an experiment in which experienced programmers with no background in user interface development are assigned to implement a graphical object editor chosen at random with different user interface tools, one of which is Unidraw. The results of this experiment would be interpreted from the time required to produce the editors and their quality. Other experiments could pit experienced user interface developers and their favorite tools against experienced Unidraw developers; explore Unidraw's limits by building editors for many more domains; or benchmark the prototype's performance in different applications.

6.3.5 Graphical Object Editor Builders

Just as the concept of assembling user interface components by direct manipulation came into its own when toolkits furnished the underlying abstractions for user interface builders, so too does Unidraw provide the foundation for a direct manipulation approach to building graphical object editors. A graphical object editor builder would offer direct manipulation analogs of Unidraw architectural features and generate Unidraw code to implement them. We see the beginnings of such capabilities in our experimental schematic capture system, where new components can be defined at run-time. A graphical object editor builder (itself a graphical object editor) would take the metaphor a step further to let a user specify commands, tools, and external representations dynamically and assemble them into domain-specific editors.

Bibliography

[1] Apple Computer, Inc. *MacDraw Manual*, 1984.

[2] Apple Computer, Inc. *Inside Macintosh, Volume I*, 1985. Published by Addison-Wesley, Reading, MA.

[3] P.S. Barth. An object-oriented approach to graphical interfaces. *ACM Transactions on Graphics*, 5(2):142–172, April 1986.

[4] E. Bier and M. Stone. Snap-dragging. In *ACM SIGGRAPH '86 Conference Proceedings*, pages 233–240, Dallas, TX, August 1986.

[5] P. Bono et al. GKS: The first graphics standard. *IEEE Computer Graphics & Applications*, 2(5):9–23, July 1982.

[6] Alan H. Borning. *ThingLab: A Constraint-Oriented Simulation Laboratory*. PhD thesis, Stanford University, 1979.

[7] Alan H. Borning. The programming language aspects of ThingLab, a constraint-oriented simulation laboratory. *ACM Transactions on Programming Languages and Systems*, 3(4):353–387, October 1981.

[8] Cadence Design Systems, Inc. *EDGE Schematic Editor Manual*, 1988.

[9] Luca Cardelli. Building user interfaces by direct manipulation. Technical Report 22, Digital Equipment Corp. Systems Research Center, October 1987.

[10] Status report of the graphics standards planning committee of ACM/SIGGRAPH. *Computer Graphics*, 13(3), Fall 1979.

[11] Robert W. Floyd. Computer Science Department, Stanford University, February 1990. Private communication.

[12] E.P. Glinert and S.L. Tanimoto. Pict: An interactive graphical programming environment. *Computer*, 17(11):7–25, November 1984.

[13] Adele J. Goldberg. *Smalltalk-80: The Interactive Programming Environment*. Addison-Wesley, Reading, MA, 1984.

[14] Adele J. Goldberg and David Robson. *Smalltalk-80: The Language and its Implementation*. Addison-Wesley, Reading, MA, 1983.

[15] Laura Gould and William Finzer. Programming by rehearsal. Technical Report SCL-84-1, Xerox Palo Alto Research Center, May 1984.

[16] Steven H. Gutfreund. ManiplIcons in ThinkerToy. In *ACM OOPSLA '87 Conference Proceedings*, pages 307–317, Orlando, FL, October 1987.

[17] D. Halbert. *Programming by Example*. PhD thesis, University of California at Berkeley, December 1984.

[18] W.T. Hewitt. Programmers Hierarchical Interactive Graphics System (PHIGS). In G. Enderle et al., editors, *Advances in Computer Graphics I*. Springer-Verlag, 1986.

[19] R.J.K. Jacob. A state transition diagram language for visual programming. *Computer*, 18(8):51–59, August 1985.

[20] Y.E. Kalay. Worldview: An integrated geometric-modeling/drafting system. *IEEE Computer Graphics & Applications*, 7(2):36–46, February 1987.

[21] Keith A. Lantz and William Nowicki. Structured graphics for distributed systems. *ACM Transactions on Graphics*, 3(1):23–51, January 1984.

[22] Mark A. Linton, Paul R. Calder, and John M. Vlissides. InterViews: A C++ graphical interface toolkit. Technical Report CSL–TR–88–358, Stanford University, July 1988.

[23] Mark A. Linton, John M. Vlissides, and Paul R. Calder. Composing user interfaces with InterViews. *Computer*, 22(2):8–22, February 1989.

[24] Joel McCormack, Paul Asente, and Ralph R. Swick. *X Toolkit Intrinsics—C Language Interface*. Digital Equipment Corporation, March 1988. Part of the documentation provided with the X Window System.

[25] M.K. Molloy. A CAD tool for stochastic petri nets. In *Proceedings of the 1986 Fall Joint Computer Conference*, pages 1082–1091, Dallas, TX, November 1986.

[26] B.A. Myers. Visual programming, programming by example, and program visualization: A taxonomy. In *ACM CHI '86 Conference Proceedings*, pages 59–66, Boston, MA, April 1986.

[27] B.A. Myers. *Creating User Interfaces by Demonstration*. PhD thesis, University of Toronto, 1987.

[28] National Instruments Corp. *LabVIEW Demonstration Package*, 1987.

[29] National Instruments Corp. *LabVIEW Manual*, 1987.

[30] G. Nelson. Juno, a constraint-based graphics system. In *ACM SIGGRAPH '85 Conference Proceedings*, pages 235–243, San Fransisco, CA, July 1985.

[31] Thomas Neuendorffer. GLO—a tool for developing window-based programs. In *Proceedings of the 1986 Winter USENIX Technical Conference*, pages 34–44, Denver, Colorado, January 1986.

[32] John K. Ousterhout et al. Magic: A VLSI layout system. In *Proc. ACM/IEEE 21st Design Automation Conference*, pages 152–159, Albuquerque, NM, June 1984.

[33] Andrew J. Palay et al. The Andrew Toolkit: An overview. In *Proceedings of the 1988 Winter USENIX Technical Conference*, pages 9–21, Dallas, Texas, February 1988.

[34] James L. Peterson. Petri nets. *Computing Surveys*, 9(3):223–252, September 1977.

[35] G. Pfaff, editor. *User Interface Management Systems*. Springer-Verlag, Berlin, 1985.

[36] G. Raeder. A survey of current graphical programming techniques. *Computer*, 18(8):11–25, August 1985.

[37] Steven P. Reiss. Working in the Garden environment for conceptual programming. *IEEE Software*, 4(6):16–27, November 1987.

[38] Steven P. Reiss, E. Golin, and R. Rubin. Prototyping visual languages with the Garden system. In *IEEE 1986 Workshop on Visual Languages*, pages 81–90, Dallas, TX, June 1986.

[39] R. Rhodes, P. Haeberli, and K. Hickman. Mex—a window manager for the IRIS. In *Proceedings of the 1985 Summer USENIX Technical Conference*, pages 381–392, Portland, Oregon, June 1985.

[40] R. Rubin, E. Golin, and Steven P. Reiss. ThinkPad: A graphical system for programming-by-demonstration. *IEEE Software*, 2(2):73–78, March 1985.

[41] Robert W. Scheifler and Jim Gettys. The X window system. *ACM Transactions on Graphics*, 5(2):79–109, April 1986.

[42] Paul G. Schreier. Groundbreaking icon-driven language handles data analysis, 488 interfacing. *Personal Engineering & Instrumentation News*, March 1987.

[43] Nan C. Shu. *Visual Programming*. Van Nostrand Reinhold, New York, 1988.

[44] Silicon Graphics, Inc. *IRIS User's Guide*, 1984.

[45] SmethersBarnes. *Prototyper Manual*, 1987.

[46] D.C. Smith. *Pygmalion: A Creative Programming Environment*. PhD thesis, Stanford University, 1977.

[47] Randall B. Smith. Experiences with the alternate reality kit: An example of the tension between literalism and magic. *IEEE Computer Graphics & Applications*, 7(9):42–50, September 1987.

[48] Reid G. Smith. Impulse-86: A substrate for object-oriented interface design. In *ACM OOPSLA '86 Conference Proceedings*, pages 167–176, Portland, OR, September 1986.

[49] Stanford University. *InterViews Reference Manual, Version 2.6*, 1989.

[50] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley, Reading, MA, 1986.

[51] Sun Microsystems, Inc. *Programmer's Reference Manual for SunWindows*, 1985.

[52] Sun Microsystems, Inc. *NeWS Preliminary Technical Overview*, 1986.

[53] I.E. Sutherland. *Sketchpad: A Man-Machine Graphical Communication System*. PhD thesis, MIT, 1963.

[54] Valid Logic Systems, Inc. *SCALDsystem Reference Manual*, 1985.

[55] Jeffrey S. Vitter. US&R: A new framework for redoing. In *Proceedings of the SIGSOFT/SIGPLAN Symposium on Practical Software Development Environments*, pages 169–176, Pittsburg, PA, April 1984. Published as SIGPLAN Notices, Volume 19, Number 5.

[56] John M. Vlissides. Dissertation proposal: Generalized graphical object editing. Stanford University, August 1987.

[57] John M. Vlissides and Mark A. Linton. Applying object-oriented design to structured graphics. In *Proceedings of the 1988 USENIX C++ Conference*, pages 81–94, Denver, CO, October 1988.

[58] John M. Vlissides and Mark A. Linton. Unidraw: A framework for building domain-specific graphical editors. In *Proceedings of the ACM SIGGRAPH Symposium on User Interface Software and Technology*, pages 158–167, Williamsburg, VA, November 1989.

[59] Bruce F. Webster. *The NeXT Book*. Addison-Wesley, Reading, MA, 1989.