COMPUTER SYSTEMS LABORATORY

STANFORD UNIVERSITY . STANFORD, CA 94305-4055

Unidraw: A Framework for Building Domain-Specific Graphical Editors

John M. Vlissides and Mark A. Linton

Technical Report: CSL-TR-89-380

July 1989

Research supported by the NASA CASIS project under Contract NAGW 419 and by the Quantum project through **a** gift from Digital Equipment Corporation.

Unidraw: A Framework for Building Domain-Specific Graphical Editors

John M. Vlissides and Mark A. Linton

Technical Report: CSL-TR-89-380

July 1989

Computer Systems Laboratory
Departments of Electrical Engineering and Computer Science
Stanford University
Stanford, CA 94305

Abstract

Unidraw is a framework for creating object-oriented graphical editors in domains such as technical and artistic drawing, music composition, and CAD. 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 behavior of objects, **tools** support direct manipulation of components, **commands** define operations on components, and **external representations** define the mapping between components and a file or database. Unidraw also supports multiple views, graphical connectivity, and dataflow between components. This paper presents Unidraw and three prototype domain-specific editors we have developed with it: a schematic capture system, a user interface builder, and a drawing editor. Experience indicates a substantial reduction in implementation time and effort compared with existing tools.

Key Words and Phrases: Object-oriented graphical editors, direct manipulation user interfaces, graphical constraints.

Unidraw: A Framework for Building Domain-Specific Graphical Editors

John M. Vlissides and Mark A. Linton
Stanford University

Abstract

Unidraw is a framework for creating object-oriented graphical editors in domains such as technical and artistic drawing, music composition, and CAD. 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 behavior of objects, **tools** support direct manipulation of components, **commands** define operations on components, and **external representations** define the mapping between components and a Ele or database. Unidraw also supports multiple views, graphical connectivity, and dataflow between components. This paper presents Unidraw and three prototype domain-specific editors we have developed with it: a schematic capture system, a user interface builder, and a drawing editor. Experience indicates a substantial reduction in implementation time and effort compared with existing tools.

Keywords: object-oriented graphical editors, direct manipulation user interfaces, graphical constraints

1 Introduction

Graphical editors represent familiar objects visually and allow a user to manipulate the representations directly. Unfortunately, these editors are difficult to build with general user interface tools because of the special requirements of graphical editors. For example, user interface toolkits provide buttons, scroll bars, and ways to assemble them into a specific interface, but they do not offer primitives for building drawing editors that produce PostScript or schematic capture systems that produce netlists. Higher-level abstractions are required to make such editors easier to implement.

We use **the term graphical object editor** for an application that lets users manipulate graphical representations of domain-specific objects and generates one or more static representations. Most graphical object editors also feature

- non-interactive operations, usually invoked from menus, that affect objects' state;
- structuring techniques for building hierarchies of objects;
- mechanisms for propagating information and maintaining graphical constraints between objects;
- and a persistent representation to store objects in non-volatile form.

Unidraw is a collection of programming abstractions that simplifies the construction of graphical object editors. **Unidraw** reduces the time it takes to produce an editor for a domain by providing functionality characteristic of graphical object editors; it does not offer toolkit features (it is used in conjunction with a toolkit), nor does it assume the role of a program development environment (it provides objects that are used within an existing environment). In this paper we present the Unidraw architecture and discuss our prototype implementation, including a brief description of three editors we have built with the Unidraw prototype: a schematic capture system, a direct-manipulation user interface builder, and an object-oriented drawing editor similar to MacDraw.

2 Related Work

We can divide current systems that support graphical object editing into three categories: domain-specific editors, multi-domain systems, and graphical programming environments. Domain-specific editors are stand-alone applications designed for editing in **a** particular domain. Object-oriented drawing editors such as MacDraw are the most common example. **Other** examples are

- computer-aided design (CAD) tools that provide a direct manipulation metaphor for producing design specifications, such as VLSI layout editors

This research has been supported by the NASA CASIS project under Contract NAGW 419 and by the Quantum project through a gift from Digital Equipment Corporation.

To appear in the Proceedings of the ACM SIGGRAPH/SIGCHI User Interface Software and Technologies '89 Conference, Williamsburg, Virginia, November 1989.

for creating chip masks and schematic capture systems for generating netlists;

- diagram editors [5, 7] that specify, model, and document physical or mathematical processes with graphical notations such as finite-state diagrams and petri nets;
- and user interface editors that let nonprogrammers assemble a user interface by direct manipulation and then generate the source code for the interface.

"Multi-domain" is a catch-all term for systems that are neither tied to a particular domain nor designed to supplant traditional programming languages, as graphical programming environments are. Examples of multi-domain systems include user interface toolkits such as Interviews [6] and GROW [2] that support definition and manipulation of an editor's graphical data; simulation systems [4, 1] that provide a direct-manipulation metaphor for representing and analyzing real-world processes in areas such as data acquisition, manufacturing, and decision support; and general constraint-based graphical editors such as Sketchpad [9] and ThingLab [3], which can theoretically support graphical editing in any domain given enough constraints.

Graphical programming environments [8] 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.

Our work with Unidraw focuses on production-quality domain-specific editors for a broad range of domains. Each of the approaches described above falls short of this goal. Domain-specific editors are designed to support a single domain only. Existing multi-domain systems have at least one of the following shortcomings:

- They provide relatively few abstractions for building domain-specific editors. For example, one toolkit might offer structured graphics but not graphical constraints, while another supports constraints but not static representations, and so on.

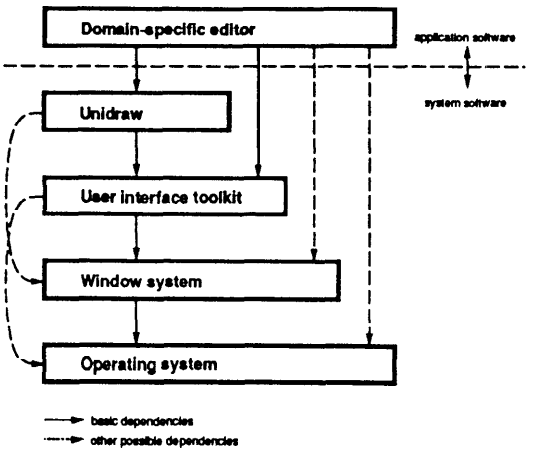

Figure 1: Layers of software underlying a domain-specific editor

- They require a large run-time environment or are embedded in a larger system. Thus they cannot be used to create stand-alone editors.
- The extension mechanism for multiple domains is not efficient enough to be practical. For example, a fast general constraint solver is hard to implement. Developing a domain-specific editor that has acceptable performance is therefore difficult when constraints are the only extension mechanism.

Graphical programming environments have proven inadequate as well. Though many such environments have been developed, none has succeeding in supplanting textual programming. Graphical 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. Moreover, most graphical programming systems are interpretive and must deal with considerable overhead associated with pictorial representations. Thus, performance is acceptable only for simple programs.

3 Unidraw Architecture

Unidraw is designed to span the 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 the "look and feel" of the user interface, and on the window and operating systems for managing workstation resources.

Figure 1 depicts the dependencies between the layers of software that underlie a domain-specific editor based on Unidraw. At the lowest levels are the operating and window systems. Above the window system level are the abstractions provided by the toolkit, including buttons, scroll bars, menus, and techniques for composing them into generic interfaces. Unidraw stands at the highest level of system software, providing abstractions that are closely matched to the requirements of graphical object editors. In theory, a domain-specific editor can access any of these layers; in practice, minimizing the number of software interfaces the programmer must use dramatically reduces complexity.

3.1 Overview

Unidraw partitions the common attributes of domain-specific editors into an object-oriented architecture having four class hierarchies:

1. **Components** are graphical representations of elements in a domain. Examples include geometric shapes in technical drawing, electronic parts in circuit layout, and notes in written music. Components encapsulate the appearance and behavior of these elements. A domain-specific editor's purpose 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, except they are stateful and can be executed as well as interpreted by objects, Commands can also be reverse-executed, allowing rollback to a previous state. Examples include commands that change component attributes, duplicate components, and group several components into a composite component.

4. **External representations are used to** 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.

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.

3.1.1 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. A subject defines the context-independent state and operations of a component. A view supports a context-dependent presentation of the subject. A component subject may have one or more 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.

3.1.2 Application Framework

Figure 2 shows the general structure of a domain-specific editor based on Unidraw. At the bottom level in the diagram are two component subjects, the leftmost containing subcomponent subjects. An entire domain-specific drawing is represented by a composite component subject that can be incorporated into a larger work. At the second level from the bottom are the corresponding views of the subjects. Note that the right-hand subject has two views attached. Each component view is placed in a **viewer at the** third level.

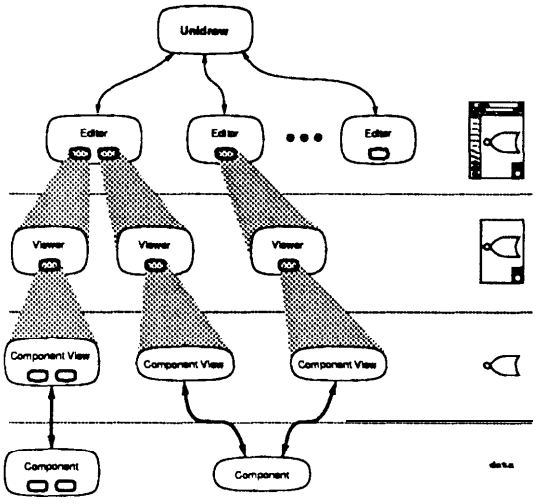

Figure 2: General structure of a domain-specific editor based on Unidraw

return values	*operation*	*arguments*
	Attach	comp. view
	Detach	comp. view
	Notify	
	Update	
	Interpret	command
	Uninterpret	command
transfer fn.	GetTransferFunction *(child iteration and manipulation operations}*	
graphic	GetGraphic	
	SetMobility	mobility
mobility	GetMobility	

Table 1: Component subject protocol

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.

An **editor** associates tools and user-accessible commands with one or more viewers and combines them into a coherent user interface. An editor also maintains a **selection** object that manages component views in which the user has expressed interest. A Unidraw-based application can create any number of editor objects, allowing the user to work on multiple views of components. Operations requiring inter-editor communication or coordination access the **unidraw** object, a one-of-a-kind object maintained by the application. For example, commands that allow the user to 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. 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 unused 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 viewers. The architecture only specifies how the editor mediates communication between components and the commands, tools, and viewers that affect them.

3.2 Components

A component defines the appearance and behavior of a domain-specific object A component's behavior has three aspects: (1) how it responds to commands and tools, (2) **its** connectivity, and (3) how it communicates with other components. This section describes the protocols and abstractions that support component semantics.

3.2.1 Subject and View Protocols

The Unidraw architecture defines separate protocols for component subjects and views. Tables 1 and 2 list the protocols' basic operations. Component subjects define Attach and Detach operations to establish or destroy a connection with a component view. 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 is used to notify 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 to interpret via the Interpret operation. The semantics of this operation are component-specific; the subject

return values	*operation*	*arguments*
	Update	
	Interpret	command
	Uninterpret	command
	{child iteration and manipulation operations}	
graphic	GetGraphic	
	Highlight	
	Unhighlight	
manipulator command	CreateManipulator	tool, event
	InterpretManipulator	manipulator

Table 2: Component view protocol

typically retrieves information from the command for internal use or executes the command. The Uninterpret operation allows the component to negate the effects of a command; the subject might undo internal state changes based on information in the command, or it might simply reverse-execute the command. Components **can** also define a **transfer function,** described in Section 3.2.4, that can be accessed via the GetTransferFunction operation. Finally, a component subject can contain other component subjects, allowing hierarchies of domain-specific components. Component subjects therefore define a family of operations for iterating through their child subjects (if any) and for reordering them.

The component view protocol duplicates some of the subject protocol's operations, namely Update, Interpret, Uninterpret, and those for child iteration and manipulation. A subject's Notify operation usually calls Update on each of its views. Interpret and Uninterpret are defined on views because some objects manipulate component views rather than their subjects. Thus it may be 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.

Graphical components are specialized components that use **graphic** objects in both their subjects and views to define their appearance. A graphic contains graphics state and geometric information and uses this information to draw itself and to perform hit detection. By definition, graphical component subjects store their geometric and graphics state in a graphic, providing a standard interface for retrieving this information. The GetGraphic operation returns the information in the subject's graphic. Graphical component subjects can **also have a mobility attribute and define** operations for assigning **and retrieving it.** Later we show how mobility is used to define the component's connectivity semantics.

Several operations augment the basic component view protocol to support graphical component views. These views maintain a graphic that defines their appearance, so they provide a GetGraphic operation. Highlight and Unhighlight operations let views distinguish themselves graphically, for example, when they are selected. CreateManipulator and InterpretManipulator define how a graphical component view reacts when it is manipulated by a tool and how the tool affects the component after manipulation. Both operations **use a manipulator to** characterize the manipulation. Manipulators abstract and encapsulate the mechanics of direct manipulation; they are discussed further in Section 3.4.

3.2.2 Connectors

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 connector subjects to define their own connectivity semantics but do not incorporate the corresponding connector views in their own views.

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 sub classes support different connection semantics. 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. **Slots and pads** provide one and two degrees of freedom within certain bounds, respectively.

Figure 3 shows how different connectors behave in several connections, using the connectors' default graphical representations. The centers of two connected pins must always coincide (Figure 3a). A pin connected to a slot (Figure 3b) 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 3c) can move relative to each other as long as the center lines of their major axes share a point. Finally, Figure 3d shows how a pad-pin connection constrains the pin to stay within the confines of the pad

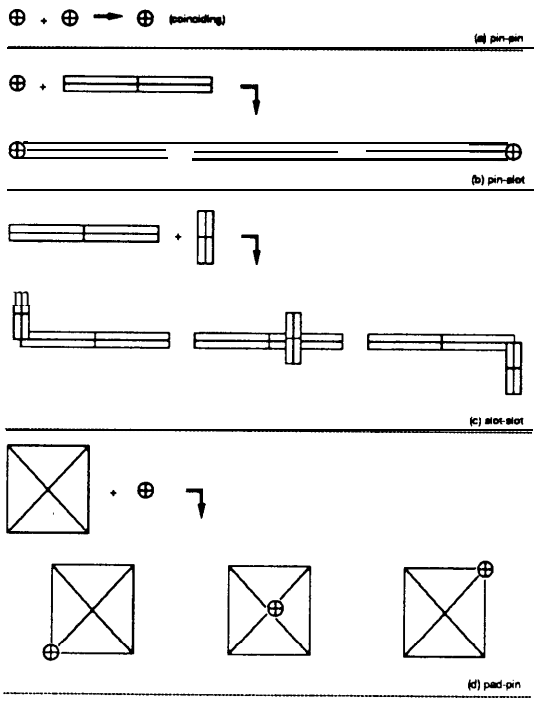

Figure 3: Several connections and their semantics

The connectors' mobilities characterize how each connector moves to satisfy the connection constraints. A mobility 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 connector 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 3b, 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 move 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.

3.2.3 Domain-Specific Connectivity

Domain-specific components use connectors to define their connectivity semantics. For example, consider

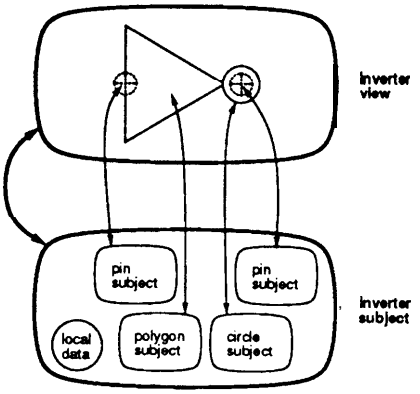

Figure 4: Possible composition of an inverter component

an inverter schematic component whose wires remain connected when the user moves it. Figure 4 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 gives its pins fixed mobility so that their positions are unaffected by any connections to them. When the inverter is moved, it moves all its components. 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 as the connection permits. One such floating connector might be a pin subcomponent of a wire component whose shape is determined by the position of the pin; the wire therefore deforms to maintain the connection. Graphical components can thus extend the connectivity behavior of primitive connectors to define their own connectivity semantics.

3.2.4 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.

Component subjects often maintain state on which other subjects depend or state that must be accessible to the user. Unidraw defines a set of objects called **state variables** that 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. **A** component can make its state variables available externally by providing access operations as necessary.

A state variable can be bound to a connector like an actual parameter is bound to a formal parameter in a procedure call. Connectors define "parameter passing" semantics for any bound state variable, one of in, **out,** or **inout**. When connected, two connectors with bound state variables will pass their values accordingly; for example, an in connector's state variable will receive the value of an out connector's variable. Passing a value between incompatible connectors (such as two out connectors) is an error; such connections should be disallowed by the tool or command making the connection.

Transfer functions complete the dataflow model by participating in the propagation of state variable values. A transfer function defines a relationship between state variables, modifying one set of variables based on the values of another set. For example, the inverter could use an *Invert* transfer function to establish a dependency between the logic level state variables bound to its in and out pins: Invert assigns the inverse value of the input variable to the output variable. Thus transfer functions describe how values change as they flow from one component to another.

3.3 Commands

Commands are analogous to messages because they can be interpreted by components. Commands are also like methods in that they are stateful and can be executed, and they resemble transactions because they can be reverseexecuted to a previous state. Some commands may be directly accessible to the user as menu operations, while others are only used by the editor internally. In general, any undoable operations should be carried out by command objects.

Table 3 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 reverseexecute. 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 indicates whether or not the command is unexecutable and uninterpretable. If the command is not reversible, then it can be ignored during the undo process.

return values	*operation*	*arguments*
	Execute	
	Unexecute	
boolean	Reversible	
	Store	**comp.** subj., any
any	**Recall**	**comp.** subj.
	SetClipboard	clipboard
clipboard	GetClipboard	
	(child iteration and manipulation operations}	

Table 3: Command protocol

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 later use 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. Furthermore, commands that operate on selected or otherwise distinguished components must maintain a record of the component subjects they affected and the order in which 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 children, if any. By default, (un)executing or (un)interpreting a macro command is semantically identical to performing the corresponding operations on each of its children.

return values	*operation*	*arguments*
manipulator	CreateManipulator	event
command	InterpretManipulator	manipulator
component view	GetPrototype	

Table 4: Tool protocol

return values	*operation*	*arguments*
boolean	Grasp	event
	Manipulating	event
	Effect	event
	(childiteration and manipulation operations)	

Table 5: Manipulator protocol

3.4 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 are viewed, or there may be no effect at all (if the tool is used in an inappropriate context, for example). Tools often use animated graphical effects as they are wielded to suggest how they will affect their environment.

3.4.1 Tool Protocol

The basic tool protocol is shown in Table 4. Conceptually, tools work within viewers, in which graphical component views are displayed and manipulated. Whenever a viewer receives an event, 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 component views (usually among those in the editor's selection object) to allow component-specific interaction. A tool's InterpretManipulator operation analyzes information in the manipulator that characterizes the manipulation and then creates a command that carries out the desired effect. If the tool delegated manipulator creation to a graphical component view, then it must delegate its interpretation to the same view.

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 consists of both a graphical component subject and a view. The tool copies the prototype, modifies it to conform to the direct manipulation, and inserts it into the component hierarchy using an appropriate command.

3.4.2 Manipulator Protocol

The manipulator protocol (Table 5) is designed to reflect the grasp-wield-effect behavior of tools. The Grasp operation takes a window system event (such as a mouse click or key press) 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.

Some kinds 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 can therefore have children, and the manipulator protocol includes operations for iterating through and manipulating them.

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. Unidraw implementations can predefine manipulators for the most common kinds of manipulation. Since manipulators must maintain information that characterizes the final 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.

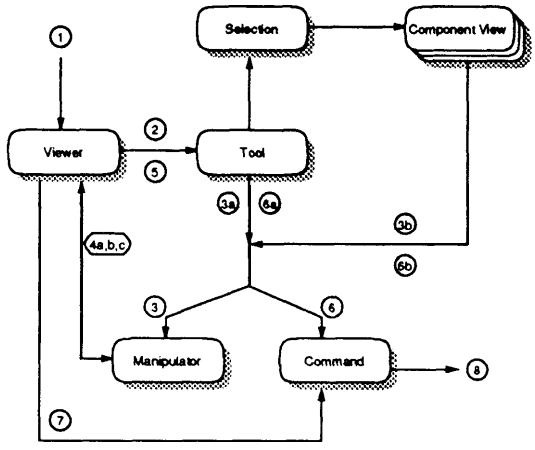

Figure 5: Communication between objects during direct manipulation

3.4.3 Object Communication during Direct Manipulation

Figure 5 diagrams the communication between objects during direct manipulation. 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 ap propriate 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 the 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 (*Rectangle View*) in the drawing area (viewer) with the *MoveTool. The* viewer receives a "mouse-button-wentdown" 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 CreateManipulator 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 parameter&es 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 External Representations

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

The external view protocol defines two operations, Emit and Definition, that generate a stream of bytes constituting the external representation. Emit initiates external representation generation, and Definition is called recursively by Emit. 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 might define a *PostScript View* 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.

4 Prototype Implementation

Our Unidraw prototype is **a** library of C++ classes containing about 20,000 lines of source. It runs on top of **Interviews and the X** Window System. The prototype uses Interviews' object-oriented structured graphics [10] to support graphical components and its persistent object facility for implementing catalog semantics.

C++ is an attractive language for our purposes because of its efficiency and true object-oriented semantics, but it does not allow sending arbitrary messages to objects. Messages are sent via strongly-typed procedure calls, so a class must declare all acceptable messages at compile-time. Thus, 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. We solved this problem in our implementation by using the IsA operation defined for InterViews persistent objects, but ideally the language would provide either run-time class resolution or untyped (or dynamically-typed) method lookup.

The remainder of this section discusses two significant aspects of the implementation followed by a brief description of the three domain-specific editors we have implemented with the Unidraw prototype.

4.1 View Consistency

A component view must reconcile its internal state with its subject's when Update is called. This is usually trivial for leaf components, but components with children must be prepared to restructure themselves to conform to their subject's structure. To accomplish this, the view could assume that all its children are inconsistent with their subjects' and just rebuild them from scratch based on the subject's structure. This 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 is identical to the subject's, Components with differing subject and view structures must implement their own update algorithm.

Figure 6 shows the algorithm. On Update, any children that no longer have a subject are destroyed. Then the list of child views is compared to the list of child subjects. If there is a subject-view mismatch, the UpdateCurrent operation is called. UpdateCurrent per-

```
void UpdateViewStructure(){
  iterator subj = SubjectsFirstChild();
  iterator view = ViewsFirstChild();
  DeleteSubjectlessViews();
  do {
    if (*subj != *view) {
      UpdateCurrent(subj, view);
    }
    Advance(subj);
    Advance(view);
  } until (Done(subj) || Done(view));
  UpdateExcessSubjects(subj);
  DeleteExcessViews(view);
}
```

Figure 6: View structure update algorithm

forms two searches through the remaining views and subjects. If it does not find a view corresponding to the current subject, it creates one and inserts it before the current view; otherwise, it moves the corresponding view to its proper position. If it does not find the current view's subject among the remaining subjects, it deletes the view; otherwise, if the corresponding subject does not follow the current subject, it moves the view to the end of the list of child views for repositioning in subsequent iterations. The main iteration loop continues until either the subject or the view runs out of children. Finally, views are created for any remaining subjects, and unused views are destroyed

4.2 Connector Implementation

Connectivity semantics are enforced by a **csolver** object that manages ***connection networks,*** or disjoint sets of connections. A connection consists of two connectors and a piece of **connector glue.** Connector glue is characterized by a natural size, elasticity, and deformation limits. Elasticity is specified in terms of independent shrinkability and stretchability parameters. Deformation limits are expressed as independent limits on the total amount the glue can stretch and shrink. A connection uses connector glue to define the relationship between connectors' centers, thus defining their connectivity semantics. For example, connector glue of zero natural size and elasticity is used to implement pin-pin connection semantics. Pin-pad semantics are modeled with a piece of glue of infinite elasticity within limits that keep the pin inside the pad.

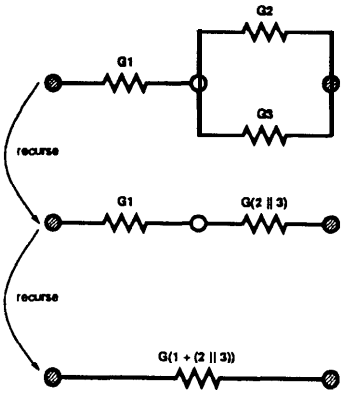

Figure 7: Recursive solution of connection network

The csolver is responsible for solving constraint 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 each network by recursively identifying primitive combinations of connections and replacing them with equivalent connections. The two most common primitive combinations are series and parallel connections. Figure 7 depicts the process of recursive substitution on a network having three connections. Connectors are shown as circles, and connector glue is represented by resistor symbols. 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. Recursion terminates whenever a single connection remains or 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 move commands to the affected connectors.

4.3 Three Domain-Specific Editor Prototypes

We have built three domain-specific editors with our prototype Unidraw library: a schematic capture system (see Figure 8), a user interface builder (Figure 9), and a drawing editor (Figure 10). All provide a direct-manipulation, multi-view editing environment. The schematic capture system lets the user wire-up circuit elements (such as gates, latches, and pass transistors)

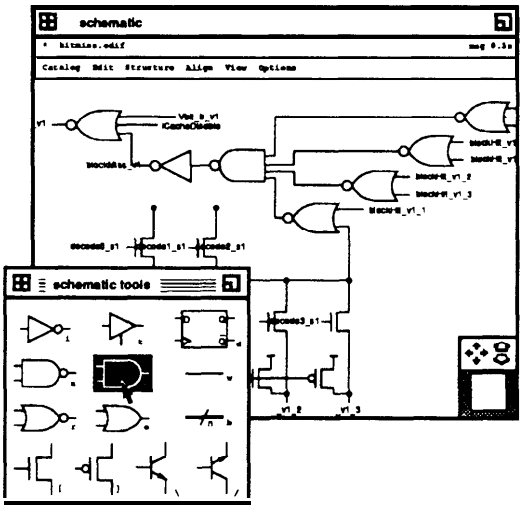

Figure 8: Schematic capture system prototype

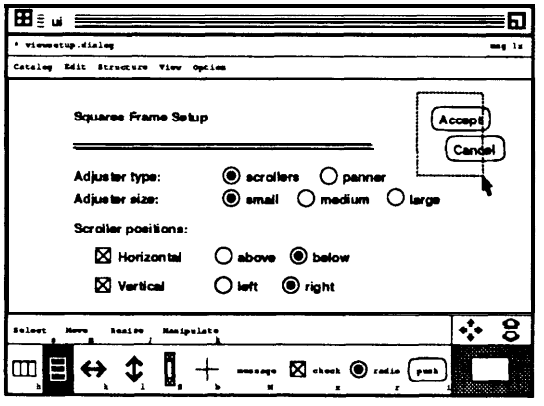

Figure 9: User interface builder prototype

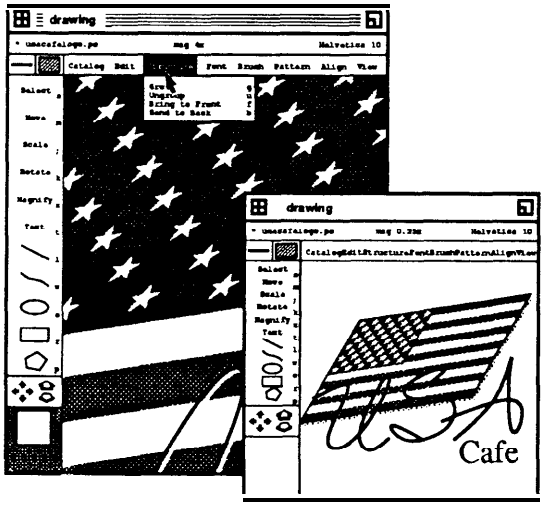

Figure 10: Drawing editor prototype

and generates a **netlist** of the resulting circuit. The system supports hierarchical composition of circuit elements and maintains graphical connectivity between them. The user interface builder lets the user compose a user interface in terms of Interviews toolkit abstractions and generates C++ source code to be incorporated into the target application. Finally, the drawing editor provides MacDraw-like functionality (with the added benefits of multiple views) and generates PostScript. The prototype schematic capture system and user interface builder are less than 5000 lines each, while the drawing editor is less than 2500 lines.

5 Conclusion

Unidraw greatly facilitated the implementation of our three prototype domain-specific editors. Though these editors do not yet represent production-quality systems, they have proven to be useful tools for their intended purposes. Unidraw narrowed the design space for each editor significantly, obviating basic design decisions that are independent of the domain. The prototype library provided reusable functionality in the form of predefined components, commands, and tools. Debugging time was reduced because much less code was written. Our experience is that developing domain-specific editors with Unidraw is mainly a matter of choosing, designing, and implementing the required domain-specific components. Significantly less effort is spent defining new commands, while specialized tools are needed the least often.

The architecture is undergoing continuous refinement as we experiment with the prototype. Fertile ground for future research involves additional support for external representations, which is a difficult problem in general. We would like to go beyond the current predefined external view traversals to develop a more powerful model that includes support for ***interpreting*** external representations. This capability would let a domain-specific editor read in existing representations, including those not generated by the editor itself. For example, a schematic editor could read in 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. The ability to read as well as write external representations permits iterative design by closing the loop between specification and analysis, making Unidraw-based tools even more useful.

References

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

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

[3] Alan H. Boming. *ThingLab — A Constraint-Oriented Simulation Laboratory.* Technical Report SSL-79-3, Xerox Palo Alto Research Center, July 1979.

[4] Steven H. Gutfreund. ManiplIcons in Thinker-Toy. In ***ACM OOPSLA '87 Conference Proceedings, pages*** 307-317, Orlando, FL, October 1987.

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

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

[7] 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.

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

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

[10] 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, October 1988.