Unidraw: A Framework for Building Domain-Specific Graphical Editors

JOHN M. VLISSIDES and MARK A. LINTON
Stanford University

Unidraw is a framework for creating 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 behavior of objects, *tools* support direct manipulation of components, *commands* define operations on components, and *external representations* define the mapping between components and the file format generated by the editor. Unidraw also supports multiple views, graphical connectivity, and dataflow between components. This paper 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.

Categories and Subject Descriptors: D.2.2 [**Software Engineering**]: Tools and Techniques—*software libraries; user interfaces*; I.3.4 [**Computer Graphics**]: Graphics Utilities—*application packages*; J.6 [**Computer Applications**]: Computer-Aided Engineering—*computer-aided design (CAD)*

General Terms: Design, Human Factors

Additional Key Words and Phrases: Direct manipulation user interfaces, graphical constraints, object-oriented graphical editors

1. INTRODUCTION

Graphical editors represent familiar objects visually and let a user manipulate the representations directly. Unfortunately, graphical editors are difficult to build with general user interface tools because such editors have special requirements. 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

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. A preliminary version of this paper was presented at the ACM SIGGRAPH Symposium on User Interface Software and Technology, Williamsburg, Va., November 1989.

Authors' address: Center for Integrated Systems, Stanford University, Stanford, CA 94305.

that produce netlists. Such applications provide direct-manipulation interfaces to graphical objects having diverse attributes, behaviors, and interdependencies.

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

—noninteractive 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 way to store objects in nonvolatile form.

Unidraw is a framework of programming abstractions that simplifies the construction of graphical object editors. Our 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 and effort needed 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 have designed and implemented Unidraw to achieve these goals. 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 and does not assume the role of a program development environment. In this paper we present the Unidraw architecture and discuss our prototype implementation, including descriptions of three editors we have built with the Unidraw prototype: an object-oriented drawing editor similar to MacDraw, a direct manipulation user interface builder, and a schematic capture system.

2. RELATED WORK

We can divide current systems that support graphical object editing into three categories: domain-specific editors, multidomain 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 examples. Other examples are

—computer-aided design (CAD) tools that provide a direct manipulation metaphor for producing design specifications, such as VLSI layout editors for creating chip masks and schematic capture systems for generating netlists;

—diagram editors [6, 10] that specify, model, and document physical or mathematical processes with graphical notations such as finite-state diagrams and petri nets; and

—user interface builders that let nonprogrammers design a user interface by direct manipulation and automatically generate the source code for the interface.

Multidomain systems include user interface toolkits, such as InterViews [8] and GROW [2], and user interface management systems, such as Higgens [5], that support definition and manipulation of an editor's graphical data; simulation systems [4, 13] 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 [16] and ThingLab [3, 9], which in principle can support graphical editing in any domain characterized by a set of constraints.

Graphical programming environments [15] let users specify arbitrary computation graphically. They are meant to replace conventional textual programming by exploiting graphics and direct manipulation. One can view graphical programming environments as extensible domain-specific editors where the domain is programming. Creating an editor for a particular domain, such as schematic capture, might therefore be a natural application of a graphical programming environment.

In our work with Unidraw we have concentrated on supporting production quality domain-specific editors for a broad range of domains. Each of the approaches described above falls short in this capacity. Domain-specific editors are designed to support one domain only. Existing multidomain 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.

—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 been unsuccessful at replacing 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

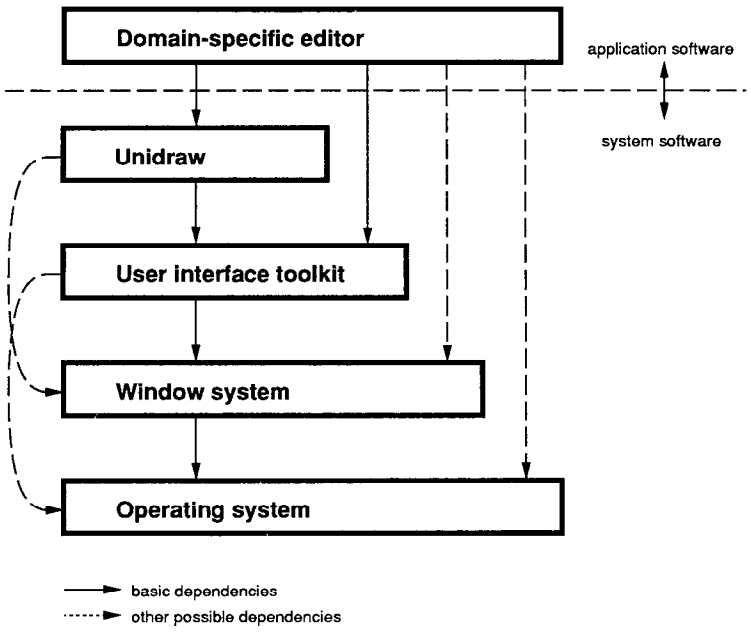


Fig. 1. Layers of software underlying a domain-specific editor.

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.

3.1 Overview

In designing the architecture we focused on the common attributes of domain-specific editors. The attributes we identified are reflected in an object-oriented organization comprising 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. Components, for example, 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.

3.1.1 *Subjects and Views.* A well-established user interface concept is the distinction between the state and operations that characterize objects and 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. Many systems employ some form of subject-view partitioning [1, 2, 7, 8, 14].

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 Organization.* 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

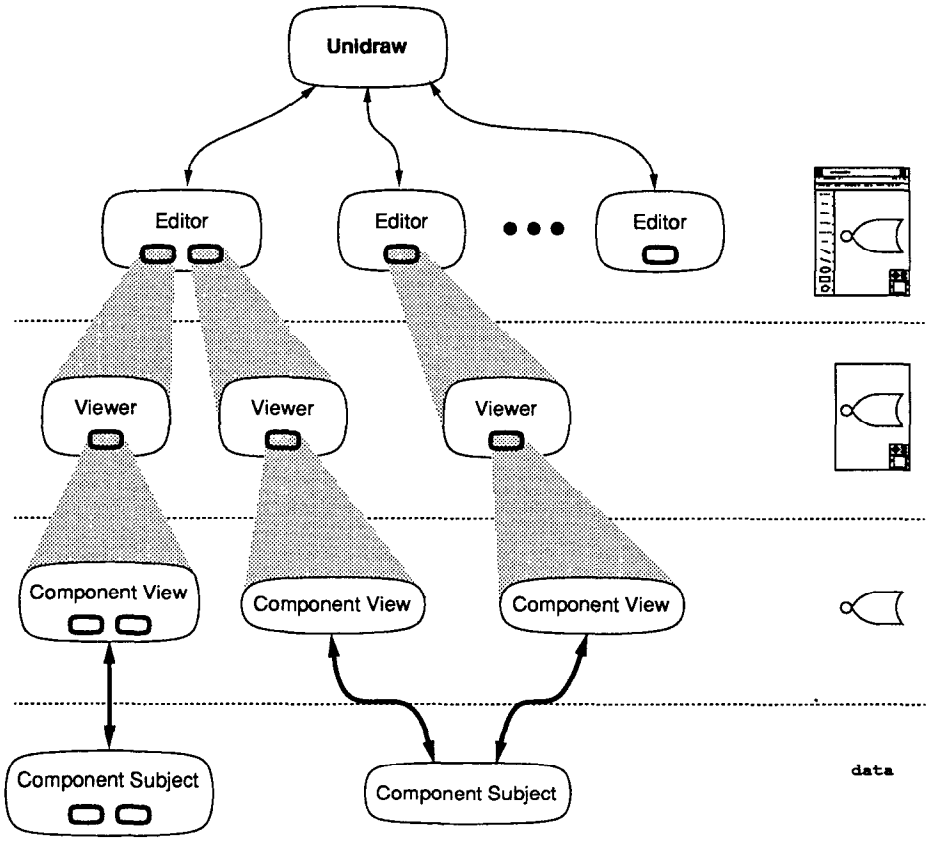


Fig. 2. General structure of a domain-specific editor based on Unidraw.

two views attached. Each component view is placed in a *viewer* at the third level. The viewer displays a graphical component view, most often the root view in a hierarchy. The viewer provides a framework for displaying the view, supporting such "nonsemantic" manipulations as scrolling and zooming. Viewers also take raw window system or toolkit events and translate them into Unidraw protocol requests.

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 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 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) its response to commands and tools, (2) its connectivity, and (3) its communication 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 I and II 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 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.

Table I. Component Subject Protocol

Return values	*Operation*	*Arguments*
	Attach	Component view
	Detach	Component view
	Notify	
	Update	
	Interpret	Command
	Uninterpret	Command
Transfer function	GetTransferFunction	
	{*child iteration and manipulation operations*}	
Graphic	GetGraphic	
	SetMobility	Mobility
Mobility	GetMobility	

Table II. Component View Protocol

Return values	*Operation*	*Arguments*
	Update	
	Interpret	Command
	Uninterpret	Command
	{*child iteration and manipulation operations*}	
Graphic	GetGraphic	
	Highlight	
	Unhighlight	
Manipulator	CreateManipulator	Tool, event
Command	InterpretManipulator	Manipulator

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 affects 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. Thus connectors and other components can be treated uniformly.

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. 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, and can undergo affine transformations.

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.

A connector's mobility characterizes how the connector moves to satisfy connection constraints. The mobility attribute can have one of three values: *fixed*, *floating*, or *undefined*. In general, a fixed connector's position cannot be affected by a connection regardless of the connection's semantics, while a floating connector will move to satisfy the connection's semantics. Components other than connectors often have undefined mobility.

Mobility specifications help determine the placement of connectors. 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.

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 theirs accordingly. The composite component might compute its own mobility from the mobilities of its components, or it may return an undefined mobility.

Connections involving pins, slots, and pads furnish a finite number of connectivity semantics. Additional semantics can be supported through subclassing. As an alternative to subclassing, however, the architecture allows connections with a piece of *connector glue* interposed. Connector glue is characterized by a natural size, elasticity, and deformation limits (see Figure 4). 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

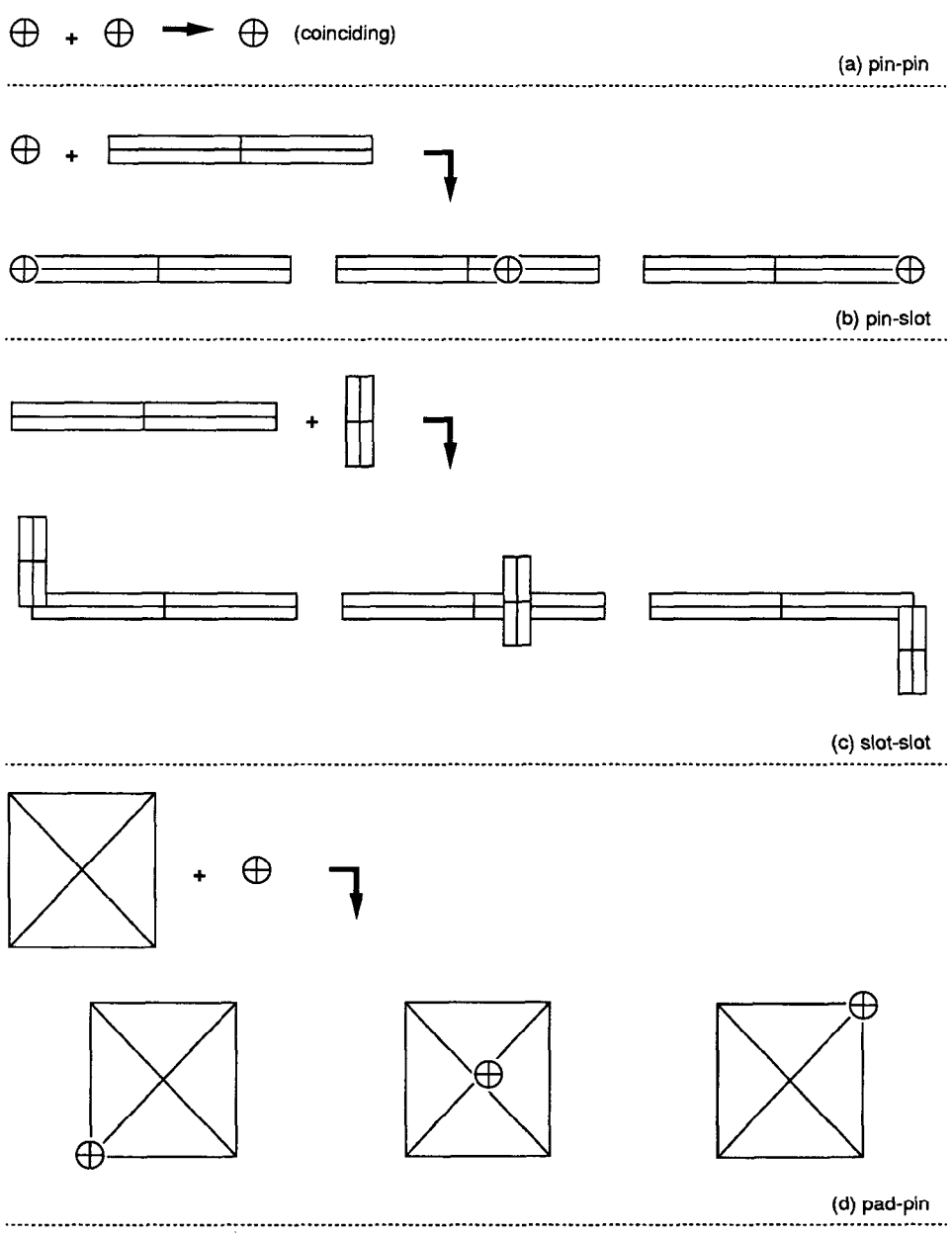
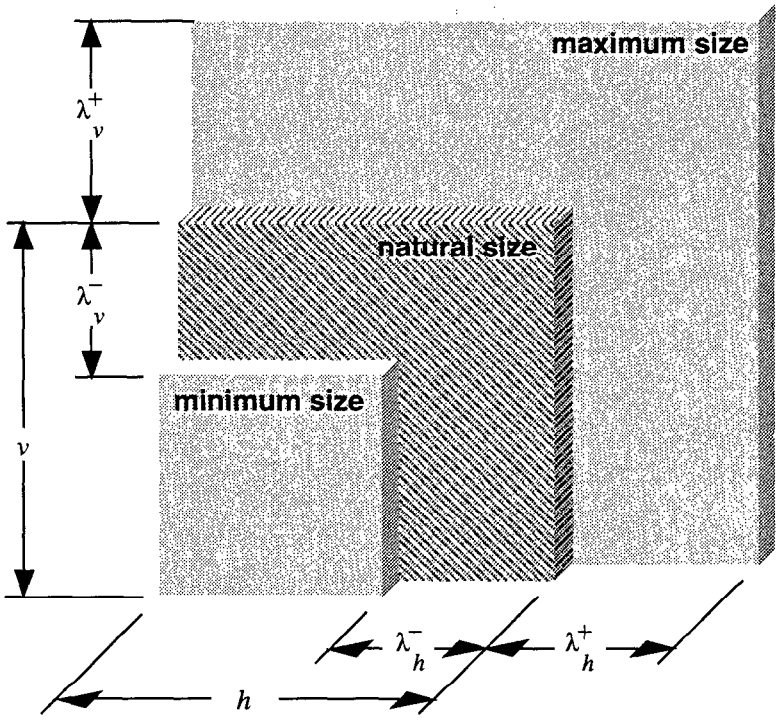


Fig. 3. Several connections and their semantics.

determines how deformation is apportioned to a collection of mutually-dependent connections.

For example, Figure 5 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



Connector glue parameters:

h = natural width	v = natural height
ε^+_h = stretchability, horizontal	ε^+_v = stretchability, vertical
ε^-_h = shrinkability, horizontal	ε^-_v = shrinkability, vertical
λ^+_h = stretch limit, horizontal	λ^+_v = stretch limit, vertical
λ^-_h = shrink limit, horizontal	λ^-_v = shrink limit, vertical

Fig. 4. Connector glue characteristics.

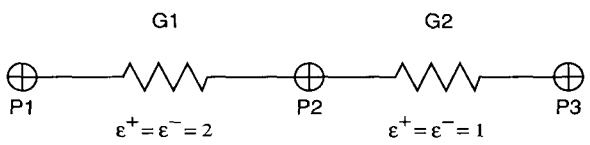


Figure 5: Three pins connected in series

Fig. 5. Three pins connected in series.

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 nonlinear springs, one horizontal and the other vertical, each having independent spring constants and travel limits in tension (ϵ^+, λ^+) and compression (ϵ^-, λ^-).

3.2.3 *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 6 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 connector glue reflecting the wire's 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.

3.2.4 *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. The model is designed to support these capabilities without incurring conceptual or implementation overhead if dataflow is not needed in a particular application.

Component subjects often maintain state that must be accessible externally, either to other subjects or to a user through some interface. Unidraw defines objects called *state variables* that represent and allow uniform access to such 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

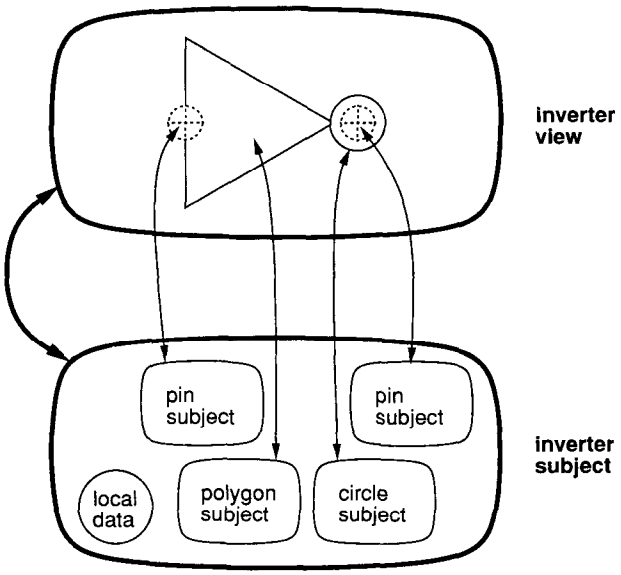


Fig. 6. Possible composition of an inverter component.

generally represents typed data, 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[1] 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.

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

[1] The term "transfer function" is a misnomer because it refers to an object.

variables. Components that require a combination of existing transfer functions must use a transfer function subclass that composes other transfer functions.

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 a connector's Transmit operation is called (e.g., following modification of the bound state variable), and the rate at which propagation proceeds is nondeterministic. If a state variable can be affected through two connection paths, for example, 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" the values of input state variables onto outputs through a control input. Transfer functions can thus support synchronization.

Unidraw's dataflow model unites components, connectors, state variables, and transfer functions to make dataflow semantics straightforward to implement. The approach differs from more general attribute evaluation semantics, such as Higgens', that propagate values automatically as soon as they are modified. In Unidraw the programmer retains control over what values are propagated and when, independent of modification time or order, and problems arising from circularities are disallowed explicitly.

3.3 Commands

Commands are analogous to messages because they can be interpreted by components. Commands are also like methods in that they are executable, but they can also be reverse executed 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 will be carried out by command objects.

Table III shows the basic operations defined by the command protocol. Execute performs computations 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 indicates whether or not the command is unexecutable and uninterpretable. A command that is not reversible implies that it can be ignored during the undo process.

Insofar as execution and reverse execution are concerned, Unidraw commands are similar to other operations-as-objects approaches characterized by Myer [11]. Unidraw commands differ from other approaches primarily in their interpreted nature, in which components interpret them like messages and uninterpret them to undo prior interpretation. 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.

Table III. Command Protocol

Return values	*Operation*	*Arguments*
	Execute	
	Unexecute	
Boolean	Reversible	
	Store	Component subject, any
Any	Recall	Component subject
	SetClipboard	Clipboard
Clipboard	GetClipboard	
	{*child iteration and manipulation operations*}	

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.

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 effects as they are wielded to suggest how they will affect their environment. Making tools distinct from components lets programmers define direct manipulation semantics and component semantics independently, enhancing modularity and reusability. For example, general manipulation behavior can be defined by one tool but can apply to any number of components.

3.4.1 *Tool Protocol.* The basic tool protocol is shown in Table IV. Conceptually, tools are used on component views as they appear in viewers. 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

Table IV. Tool Protocol

Return values	*Operation*	*Arguments*
Manipulator	CreateManipulator	Event
Command	InterpretManipulator	Manipulator
Component subject	GetPrototype	

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 is a graphical component 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.

3.4.2 *Manipulator Protocol.* The manipulator protocol (Table V) 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 submanipulations 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.

Table V. Manipulator Protocol

Return values	*Operation*	*Arguments*
	Grasp	Event
Boolean	Manipulating	Event
	Effect	Event
	{*child iteration and manipulation operations*}	

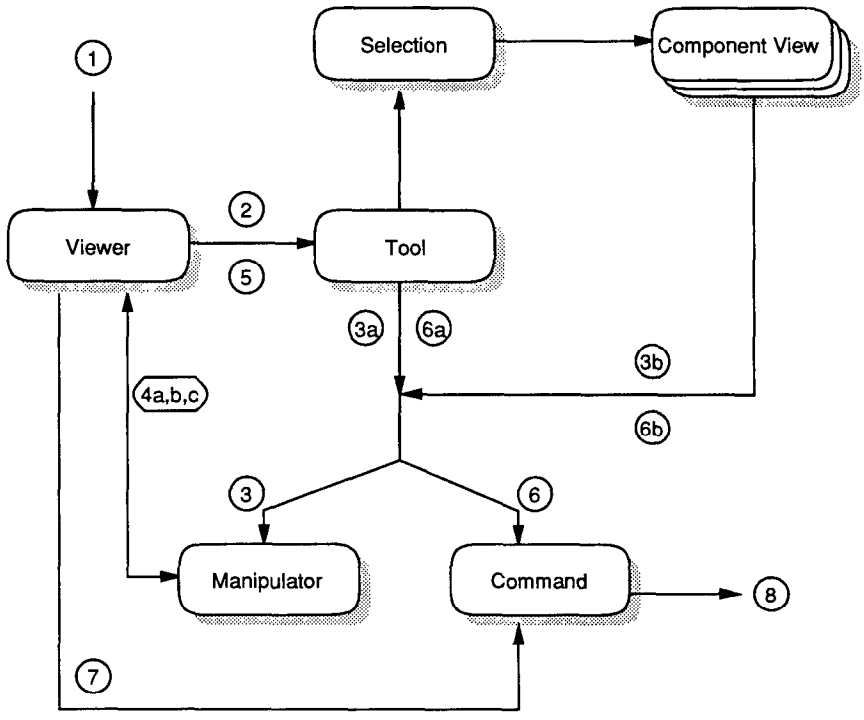


Fig. 7. Communication between objects during direct manipulation.

3.4.3 *Object Communication During Direct Manipulation.* Figure 7 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 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 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 a 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 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 a 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 lets the manipulator 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.

3.4.4 *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 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.

Unidraw's approach to direct manipulation is similar in some ways to that used in Garnet [12], which defines objects called *interactors* to encapsulate the related machinery. A Garnet interactor defines a particular style of manipulation with parameters for specifying initialization, termination, and other, interactor-specific conditions. Garnet interactors differ from tools in that all interactors use the same highly parameterized state machine to define the mechanics of the manipulation, and there is no mechanism analogous to manipulator composition. Moreover, Garnet interactors are not tightly associated with a particular set of objects, whereas tools and components are more closely coupled. Thus objects that interactors ultimately affect have a limited say in how the manipulation will progress: they can influence the dynamics of the manipulation only as far as the interactor's parameterization will allow. Finally, Garnet interactors have a fixed set of actions they perform in response to the manipulation, though it is possible to specify arbitrary Lisp procedures as well; tools, on the other hand, generate commands, thereby incorporating support for undo.

3.5 External Representations

An external representation of a component is generated by a nongraphical view of the corresponding component subject. Domain-specific external representation objects are derived from the *external view* subclass of component view. This

approach has several benefits:

—Since a component can have any number of views, it can have any number of external representations, even simultaneously.
—Existing mechanisms for keeping a component view consistent with its subject can be used to keep the external representation consistent.
—Like other component views, external views can be composed to simplify generation of the external representation.

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 *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 three common traversals of the external view hierarchy.

4. PROTOTYPE IMPLEMENTATION

Our Unidraw prototype is a library of C++ classes containing about 25,000 lines of source. It runs on top of InterViews and the X Window System. We begin this section by explaining our choice of C++ and relating some of its shortcomings with respect to this work. Then we briefly describe key aspects of the InterViews structured graphics library. The remainder of the section discusses our approach to two implementation issues: maintaining subject-view consistency and enforcing connectivity semantics.

4.1 C++

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. Our implementation defines an IsA operation based on programmer-managed class identifiers to solve this problem, but ideally the language would provide a way to determine an object's type at run-time.

4.2 Structured Graphics

InterViews provides a comprehensive user interface toolkit with object-oriented structured graphics [18]. Graphical components in the prototype use structured graphics objects to define their appearance. These objects support hit detection and incremental screen update, thereby removing much low-level graphics code from the Unidraw library. The classes supporting structured graphics collectively form the InterViews Graphic library.

The approach to incremental update is key to understanding the subsequent discussion of subject-view consistency. The Graphic library includes a *damage* base class that defines a protocol for objects that 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.

Upon creation, a damage object receives a graphic for which it is responsible, usually a complex, composite graphic. The damage class defines an Incur operation that the client program calls whenever the graphic is *damaged*. The programmer must supply the damaged region to the Incur operation, either by passing the graphic that contributed the region or by specifying a rectangular area explicitly. For simplicity we sometimes refer to this operation as "damaging" the supplied graphic or rectangular area, though strictly speaking we are *incurring* damage on a damage object *with* the graphic or rectangular area. The graphic is incrementally updated when a Repair operation is called. Often, damage is incurred several times before it is repaired. This interface is convenient for the programmer because he can choose to incur damage at any point, not necessarily when a graphic is modified. He can also choose to ignore the incremental approach altogether if it is simpler to just redraw a graphic explicitly.

4.3 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 is novel in that it automatically handles the common case where the view's child structure is identical to the subject's. For components with differing subject and view structures, the programmer must

```
UpdateViewStructure (subject s, view v){
  let triples L = list of triples (child of s, child of v, child of v's original index in v);
  Initialize(s, v, L);
  Rearrange(s, v, L);
  Damage(s, v, L);
}
```

Fig. 8. View structure update algorithm.

implement a custom update algorithm as he would in existing subject-view systems.

The algorithm consists of three routines, as shown in Figure 8: an initialization routine, a view rearrangement routine, and a damage routine.

4.3.1 *Initialization and View Rearrangement.* Figure 9 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 (through an InterViews damage object) in the process. This routine also initializes a list of triples, one triple for each subject. Each triple associates a child subject with its view in the parent and the index of that view prior to rearrangement (for example, the ith child). The information in the triples is used in the remaining routines to expedite view rearrangement and to determine which graphics to damage.

Rearrange repositions child views that are out of order with respect to the order of the child subjects. It simply iterates through the child subjects and views in tandem, comparing each subject with the corresponding view's subject. If there is a mismatch, it finds the proper view for the subject in the list of triples and inserts it into its rightful place.

4.3.2 *Damage Incursion.* The Damage routine, shown in Figure 10, builds on the InterViews damage object's capabilities to support incremental screen update of Unidraw composite component views. It determines a set of graphics to damage so that the parent view's appearance will reflect any restructuring. We assume in the algorithm that minimizing the number of times damage is incurred 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 minimal 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 index. As it scans, it records the largest

```
Initialize (subject s, view v, triples L) {
  for each child of v {
    if the child's subject is not a child of s {
      discard the child of v, damaging its graphic;
    } else {
      create a triple T recording the child view, its subject, and its index in v;
      enter T into L;
    }
  }
  for each child of s {
    find the triple in L containing the child;
    if no triple containing the child exists {
      create a new view of the child subject, making it the last child of v and damaging its graphic:
      create a triple T recording the child view, its subject, and its index in v;
      enter T into L;
    }
  }
}
Rearrange (subject s, view v, triples L) {
  for each child of s {
    find the triple T in L containing the child;
    let integer i = child's index in s;
    if the child view in T is not the ith child of v {
      make T's child view the ith child of v;
    }
  }
}
```

Fig. 9. Initialization and view rearrangement.

```
Damage (subject s, view v, triples L) {
  let sets D_f, D_r = sets of views, initially empty;
  for each child of s {
    find the triple T in L containing the child;
    if the child view in T is out of order in the forward direction {
      add the child view to D_f;
    }
    if the child view in T is out of order in the reverse direction {
      add the child view to D_r;
    }
    damage all graphics from views in either D_f or D_r, whichever set is larger;
  }
}
```

Fig. 10. Damage incursion.

original index encountered so far in the forward direction and the smallest in the reverse direction. Any views whose original index is less than the largest in the forward direction are out of order with respect to those already encountered; these views form a set whose graphics may require damaging. Similarly, any

views whose original index is greater than the smallest in the reverse direction are out of order with respect to those already encountered; these form another damage set. Once these two sets have been accumulated, the algorithm damages the graphics from the views in the smaller set.

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 (e.g., bring-to-front and send-to-back operations). In these cases, either the forward or the backward scan will identify exactly those views as being out of order. 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.

4.4 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. 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.

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 represents 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. Csolver identifies three primitive connection combinations: *series*, *parallel*, and *Y*.

Figure 11 depicts the process of recursive substitution on a network having 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. The equivalent's glue parameters (natural size, elasticity, and deformation limits) are calculated from the original glue parameters based on simple, fixed formulas.[2] For example, the natural size for the equivalent of a series combination is simply the sum of the original glues' natural sizes. Next, the csolver 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

[2] The relationships between primitive combinations of glue and their equivalents are described in more depth elsewhere [17].

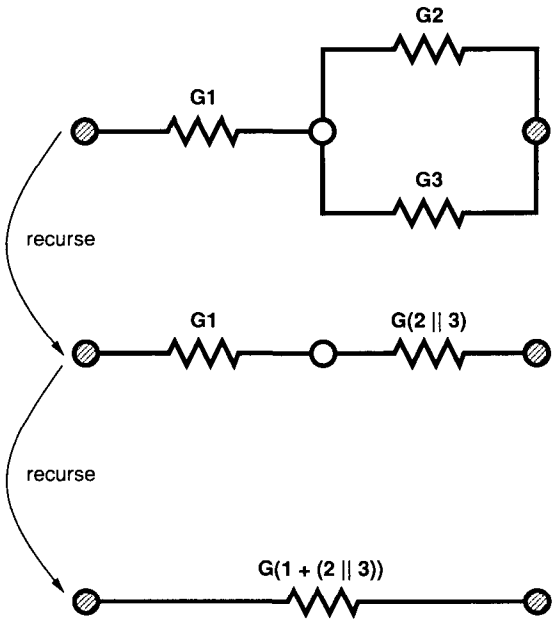


Fig. 11. Recursive solution of connection network.

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.

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 nonlinear connector glue. A rigorous solution would involve solution of nonlinear simultaneous equations, which generally requires iterative techniques and thus would be much more expensive. Also, because csolver constructs independent networks for constraints in the horizontal and vertical dimensions, it cannot enforce connectivity semantics that involve dependencies between them. Our implementation therefore restricts slots and pads to orthonormal orientations. In practice, however, this approach is a reasonable compromise between efficiency and accuracy, at least for the experimental editors we have built. The implementation models the most common connections (pin-pin, pin-slot, and pin-pad) accurately and exhibits predictable behavior for more complex connections.

5. EXPERIMENTAL APPLICATIONS

We built three domain-specific editors with the prototype Unidraw library: 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

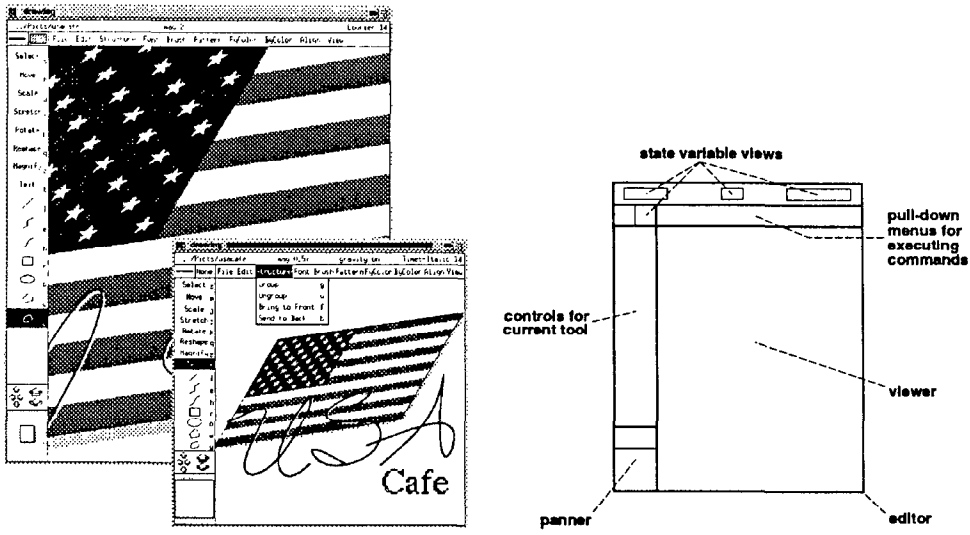


Fig. 12. Drawing editor.

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.

5.1 Drawing Editor

The drawing editor, called *Drawing* (Figure 12), 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.

Figure 12 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

Table VI. Drawing Code Breakdown

		Code (lines)	
	Classes	*Interface*	*Implementation*
Components	0	0	0
Commands	1	20	40
Tools/manipulators	0	0	0
External representations	0	0	0
State variables/transfer functions	0	0	0
Editors	1	80	900
Creator	1	20	30
Toolkit-derived classes	0	0	0
Globals	0	30	100
Totals	3	150	1070

are defined by or derived from toolkit objects, while other objects are based on Unidraw abstractions. Table VI presents a breakdown of the Drawing implementation in classes and lines of source code.

5.2 User Interface Builder

The user interface builder, called UI (Figure 13), provides a direct-manipulation environment for assembling toolkit objects into a complete interface. UI defines components that correspond to basic InterViews interactors (widgets) such as scroll bars and push buttons, and it provides components that implement InterViews composition mechanisms, including boxes and tray.

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 takes advantage of multiple views to allow editing parts of an interactor composition without disturbing the overall hierarchy. The user can select an interactor component buried deep in the composition and create a view of any of its ancestors up to the root. Once the user is satisfied with the interface's appearance and behavior, UI can generate C++ code that implements the interface.

Table VII presents a breakdown of UI's implementation. Each InterViews interactor is represented by a Unidraw component, and state variables represent interactor attributes such as canvas, shape, and button state. The user can examine and potentially change these attributes through dialog boxes comprising the corresponding state variable views. UI supports scenes (interactors that compose other interactors) using composite components. In particular, UI's tray component models the composition semantics of InterViews trays, which allow the user to specify arbitrary interactor layouts. For example, a tray enforces placement constraints requiring that the right edge of one interactor abut with the left edge of another, perhaps with a piece of elastic whitespace in between. UI's tray component enforces such constraints using connectors, both to keep the interactors properly aligned and to model their elastic properties [17].

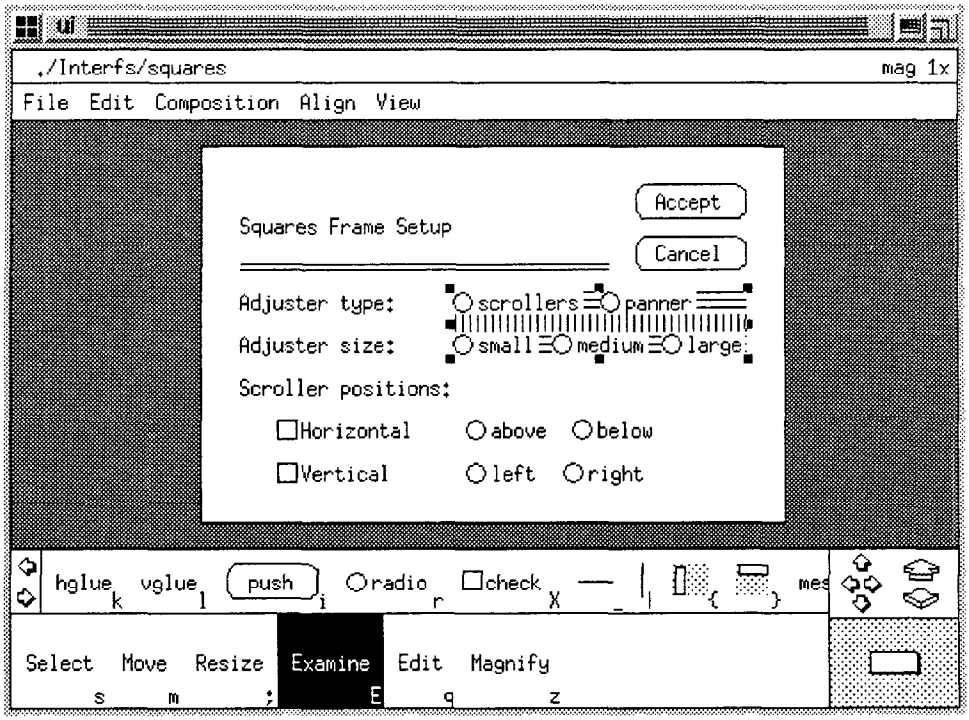


Fig. 13. User interface builder.

Table VII. UI Code Breakdown

		Code (lines)	
	Classes	*Interface*	*Implementation*
Components	24	500	1870
Commands	7	140	360
Tools/manipulators	2	60	180
External representations	10	170	460
State variabless/transfer functions	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

5.3 Schematic Capture System

The schematic capture application is called *Schem* (Figure 14). It supports hierarchical circuit specification and generates both hierarchical and flattened netlist external representations. Schem also provides connectivity maintenance and multiview editing, and it uses dataflow to support combinational logic simulation. Moreover, Schem is the most extensible of the three experimental applications in that a user can extend its repertoire of components at run-time. He can create new schematic components and define their appearance,

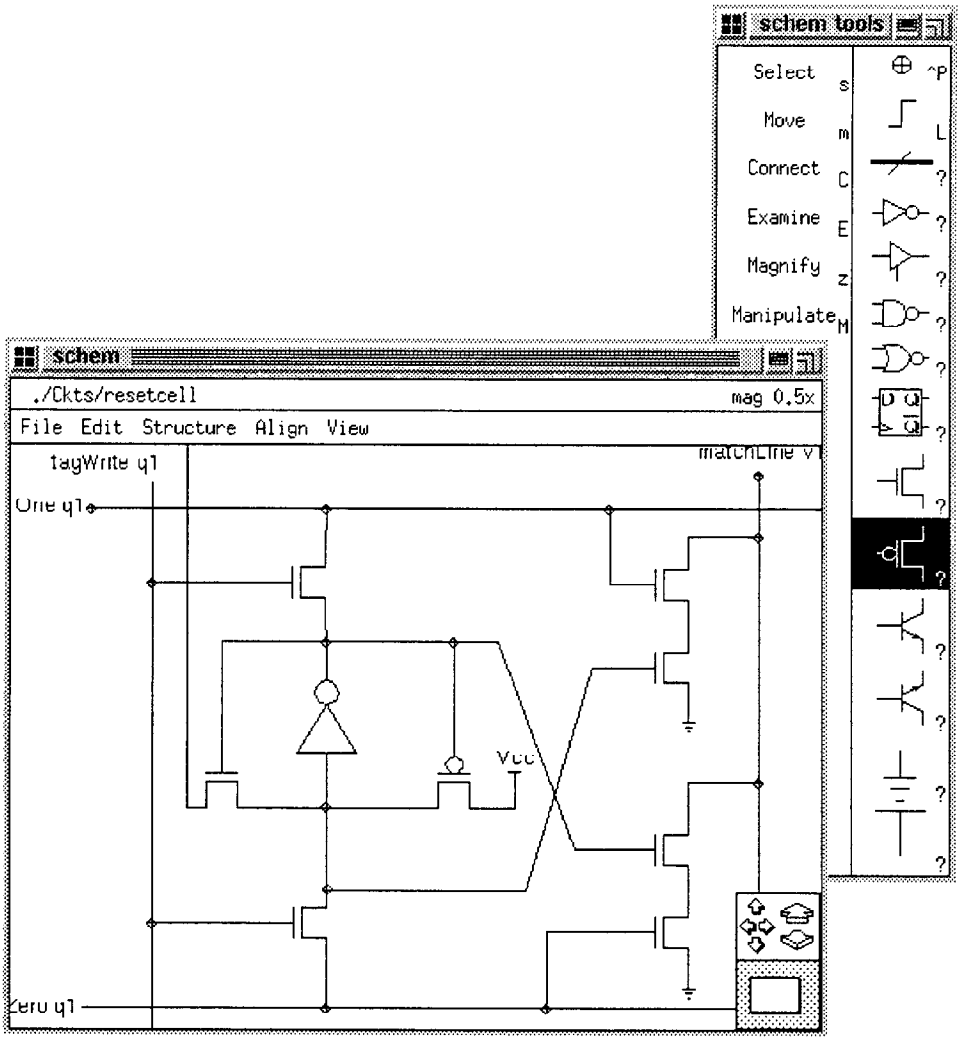


Fig. 14. Schematic capture system.

connectivity and netlist generation semantics, and logical functions. The class and source code breakdown for Schem is shown in Table VIII.

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.

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 bidirectional capacity. These

Table VIII. Schem Code Breakdown

		Code (lines)	
	Classes	*Interface*	*Implementation*
Components	14	300	1100
Commands	9	200	500
Tools/manipulators	5	100	200
External representations	5	100	400
State variables/transfer functions	5	100	300
Editors	3	200	1100
Creator	1	20	60
Toolkit-derived classes	6	200	800
Globals	0	200	100
Totals	48	1420	4560

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, thereby enabling combinational logic simulation. Element and node names and logic levels are represented with state variables, while transfer functions implement the truth table relationships.

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 affects 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.

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. Creating a new element involves three steps: (1) define its appearance, (2) define its semantics, and (3) store it for later use. The user defines the element's appearance using drawing tools similar to those provided in Drawing. The element's connectivity semantics are defined by composing nodes in a manner similar to that described in Section 3.2.3. The user can specify netlist semantics through state variable views for editing an element's name and those of its nodes. The user can also modify an element's combinational logic semantics through dialogs for changing node directionalities and the transfer function's logic-level dependencies.

6. CONCLUSION

Unidraw greatly simplified the implementation of our three experimental domain-specific editors. Though these editors are not polished systems, they demonstrate

that Unidraw is a viable way to build practical applications. 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.

Fertile ground for future research involves additional support for external representations. We would like to go beyond the current predefined external view traversals to develop a more powerful model that supports the *internalization* of one or more different external representations. While Unidraw currently does not preclude this capability, neither does it aid its implementation. Internalization would let a domain-specific editor read representations produced by other applications. For example, a schematic editor could read in a 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 the user 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.

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.

Finally, Unidraw provides a foundation for research into *graphical object editor-building* applications. 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 would 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.

REFERENCES

1. Apple Programmer's & Developer's Association. *MacApp: The Expandable Macintosh Application*, 1987.
2. Barth, P. S. An object-oriented approach to graphical interfaces. *ACM Trans. G. 5*, 2 (April 1986), 142–172.

3. BORNING, A. H. ThingLab: A constraint-oriented simulation laboratory. Ph.D. dissertation. Stanford Univ., 1979.
4. GUTFREUND, S. H. ManiplIcons in ThinkerToy. In *ACM OOPSLA '87 Conference Proceedings*, (Orlando, Fl., Oct. 1987), 307–317.
5. HUDSON, S. E. AND KING, R. Semantic feedback in the Higgens UIMS. *IEEE Trans. Softw. Eng., 14,* 8 (Aug. 1988), 1188–1206.
6. JACOB, R. J. K. A state transition diagram language for visual programming. *Computer 18,* 8 (Aug. 1985), 51–59.
7. KRASNER, G. E., AND POPE, S. T. A cookbook for using the model-view-controller user interface paradigm in Smalltalk-80. *J. Object-Oriented Program. 1,* 3 (Aug./Sept. 1988), 26–49.
8. LINTON, M. A., VLISSIDES, J. M., AND CALDER, P. R. Composing user interfaces with InterViews. *Computer 22,* 2 (Feb. 1989), 8–22.
9. MALONEY, J. H., BORNING, A. H., AND FREEMAN-BENSON, B. N. Constraint technology for user interface construction in ThingLab II. In *ACM OOPSLA '89 Conference Proceedings*, (New Orleans, La., Oct. 1989), 381–388.
10. MOLLOY, M. K. A CAD tool for stochastic petri nets. In *Proceedings of the 1986 Fall Joint Computer Conference.* (Dallas, Tex., Nov. 1986), 1082–1091.
11. MYER, B. *Object-Oriented Software Construction.* Prentice Hall, New York, N.Y., 1988.
12. MYERS, B. A. Encapsulating interactive behaviors. In *ACM CHI '89 Conference Proceedings*, (Austin, Tex., May 1989), 319–342.
13. NATIONAL INSTRUMENTS CORP. *LabVIEW Manual*, 1987.
14. PALAY, A. T., HANSEN, W. J., KAZAR, M. L., SHERMAN, M., WADLOW, M. G., NEUENDORFFER, T. P., STERN, Z., BADER, M., AND PETERS, T. The Andrew toolkit: An overview. In *Proceedings of the 1988 Winter USENIX Technical Conference.* (Dallas, Tex., Feb. 1988), 9–21.
15. SHU, N. C. *Visual Programming.* Van Nostrand Reinhold, New York, 1988.
16. SUTHERLAND, I. E. Sketchpad: A man-machine graphical communication system. Ph.D. dissertation, MIT, 1963.
17. VLISSIDES, J. M. Generalized graphical object editing. Ph.D. dissertation, Stanford Univ., 1990.
18. VLISSIDES, J. M., AND LINTON, M. A. Applying object-oriented design to structured graphics. In *Proceedings of the 1988 USENIX C++ Conference.* (Denver, Colo., Oct. 1988), 81–94.