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 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.

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.

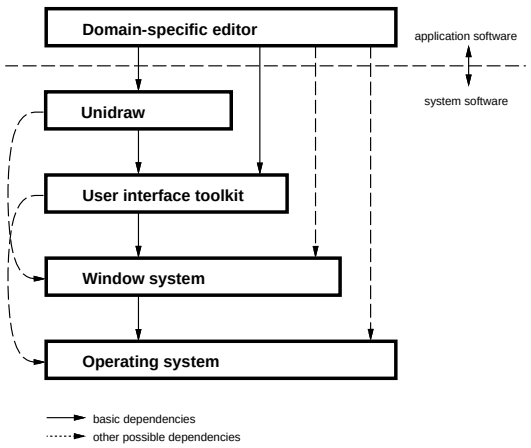


Figure 1: Layers of software underlying a domain-specific editor

- 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.

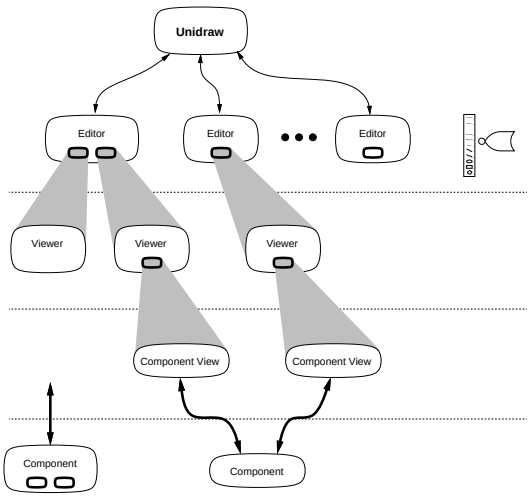
3.1.1 Subjects and Views

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.

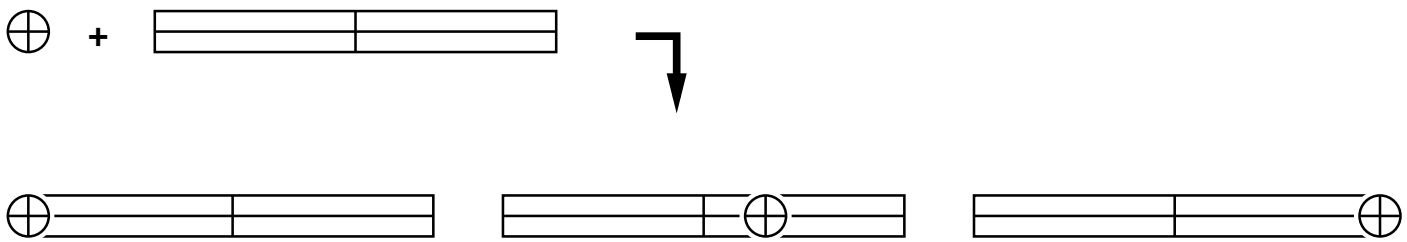
A component subject maintains information that characterizes the component; in the case of a logic gate component, for example, the subject might contain information about what is connected to the gate and its current input values. Different views of the subject can reflect this information in distinctive 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

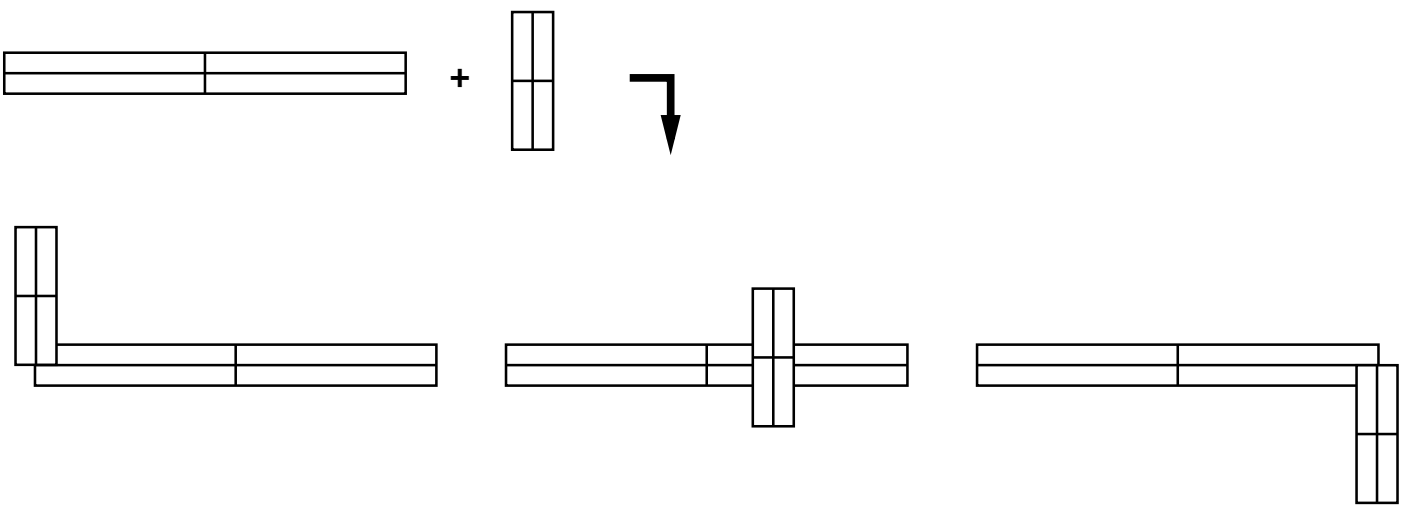
Figure 2 shows the general structure of a domain-specific editor based on Unidraw. At the bottom level in the diagram are two component subjects, the leftmost containing subcomponent subjects. An entire domain-specific drawing is represented by a composite component subject that can be incorporated into 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.


Unidraw
Editor
Editor
Editor
Viewer
Viewer
Viewer
Component View
Component View
Component
Component

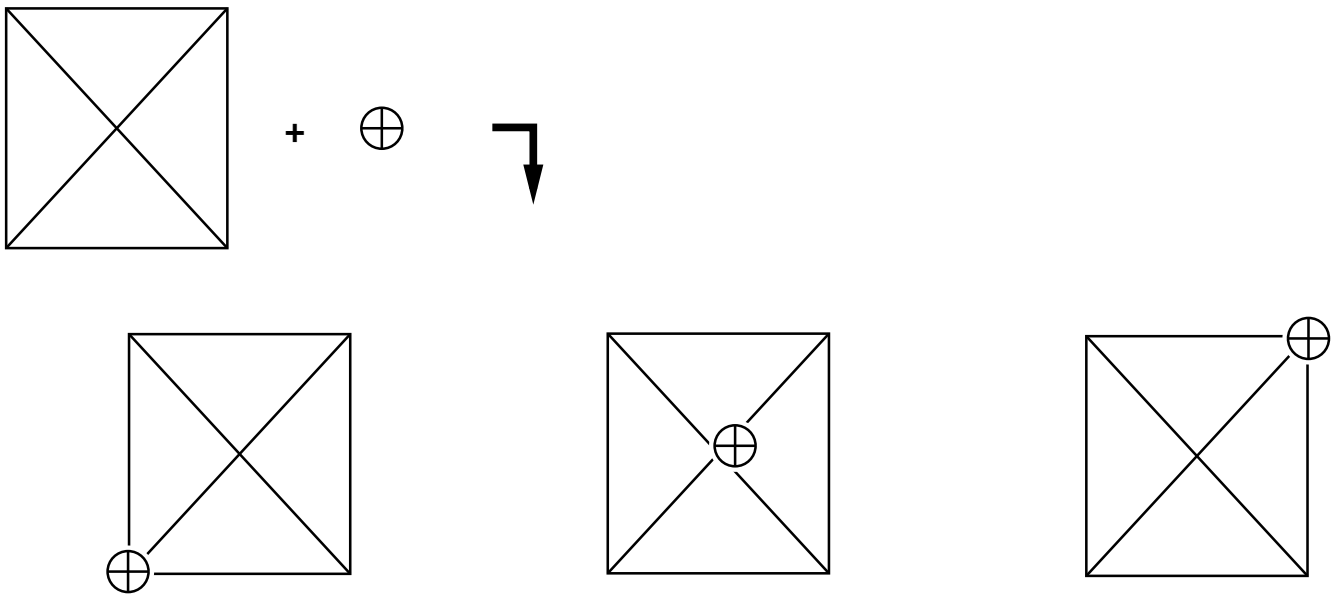
(a) pin-pin



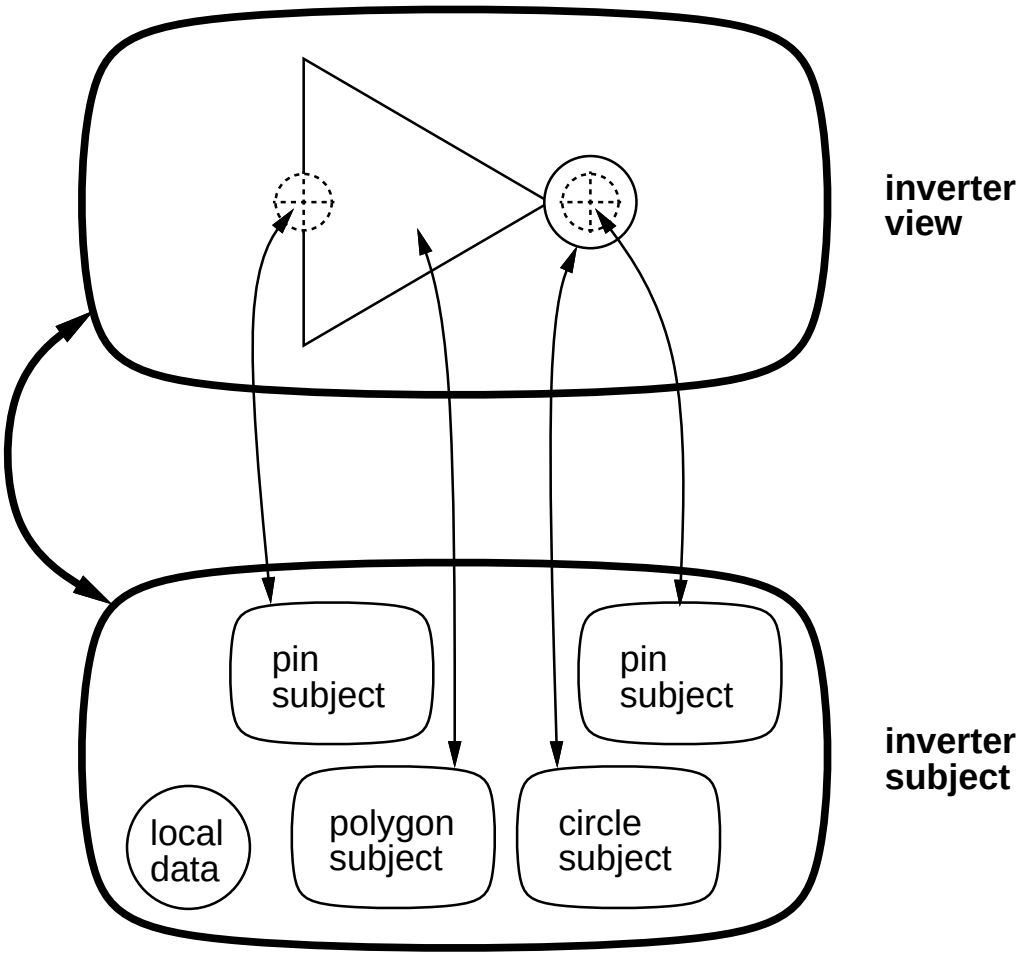
(b) pin-slot

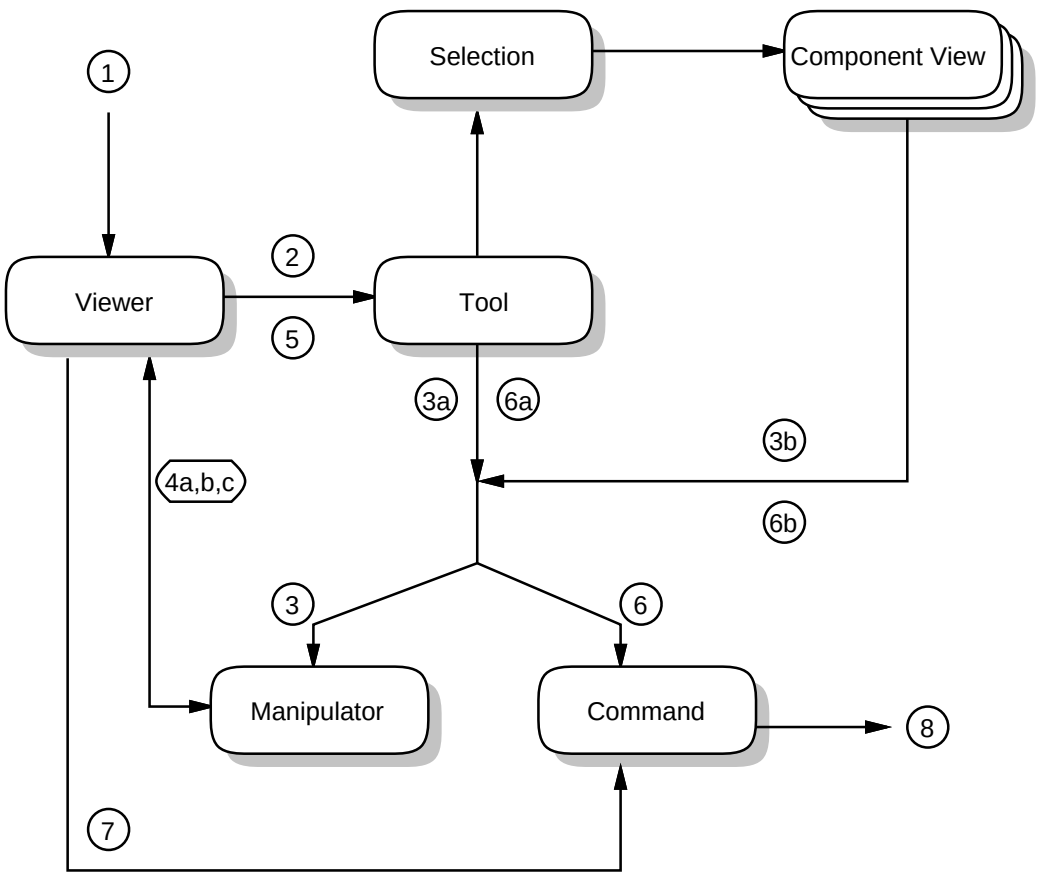


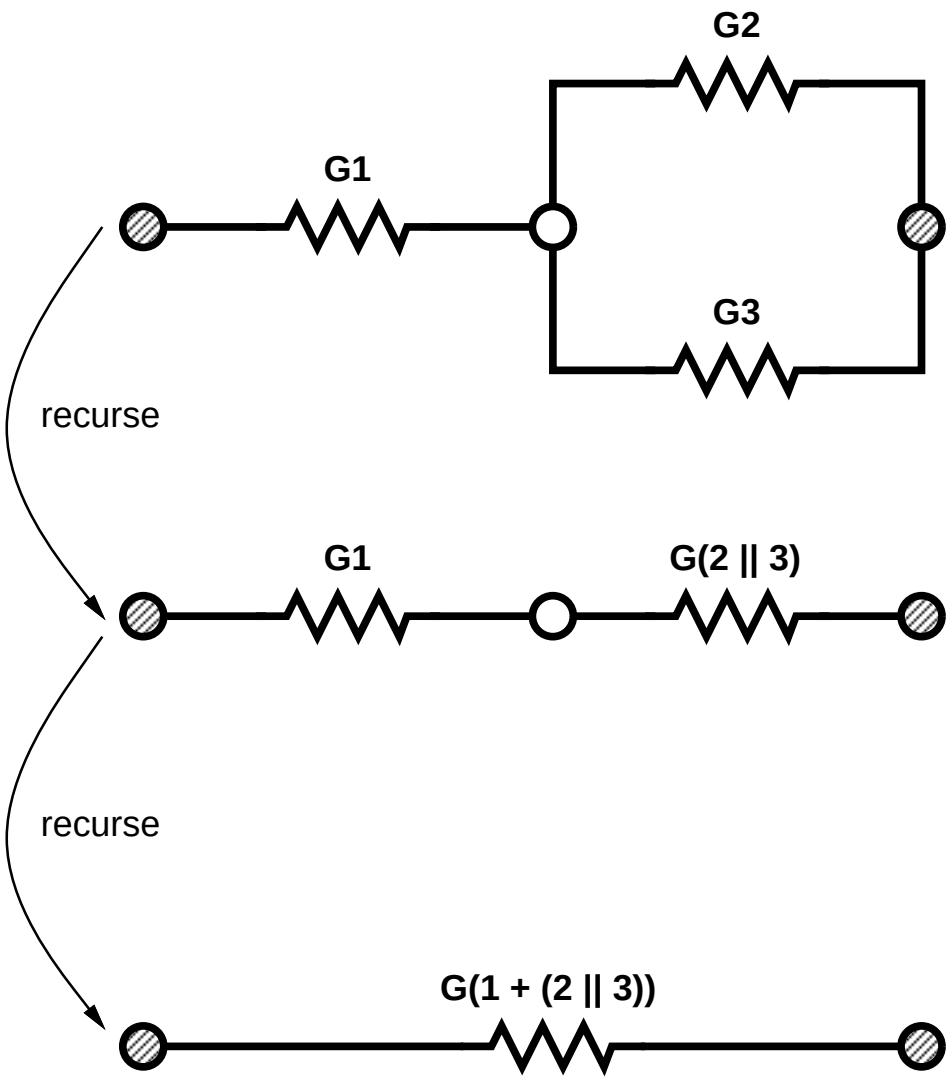
(c) slot-slot

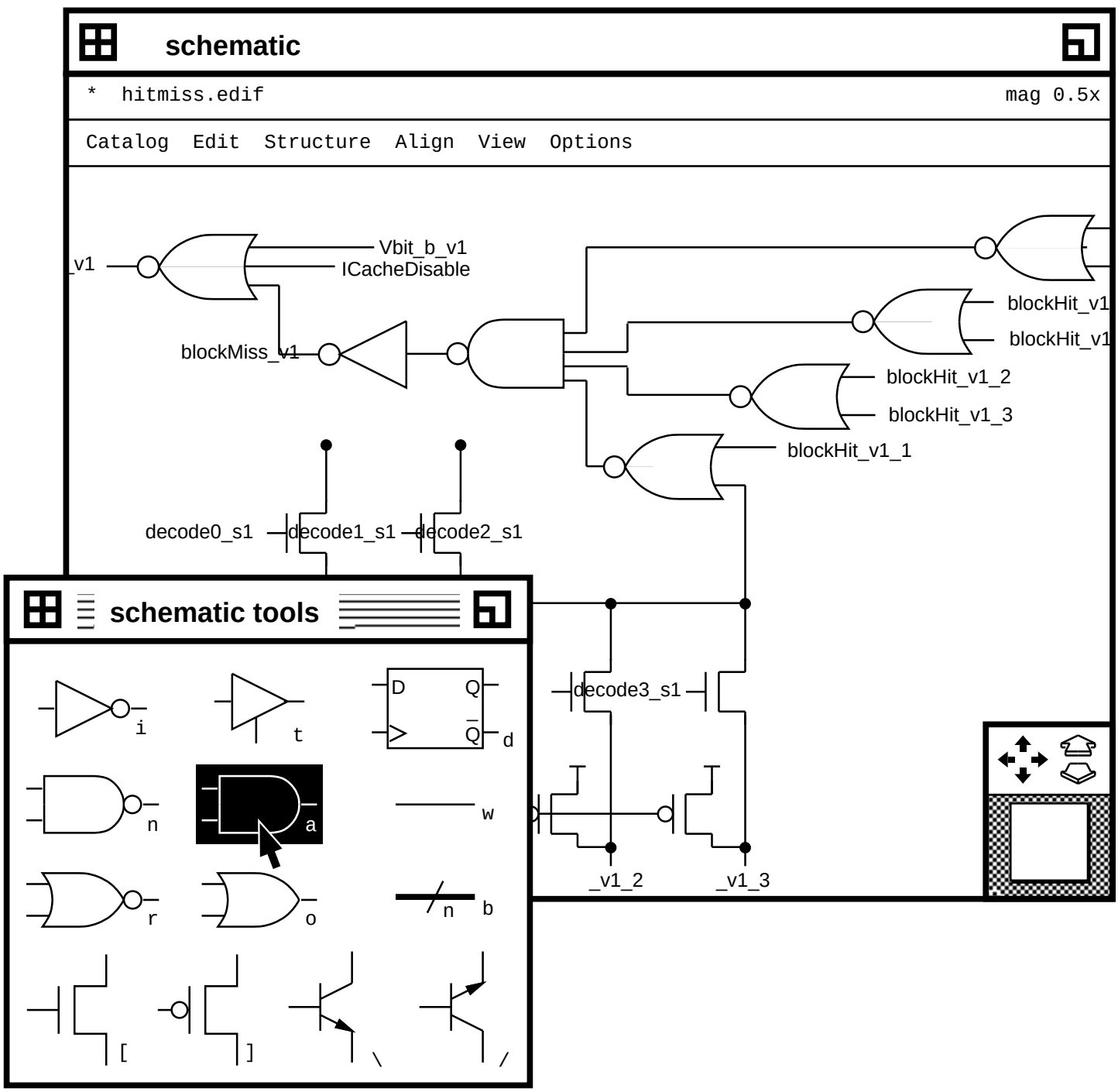


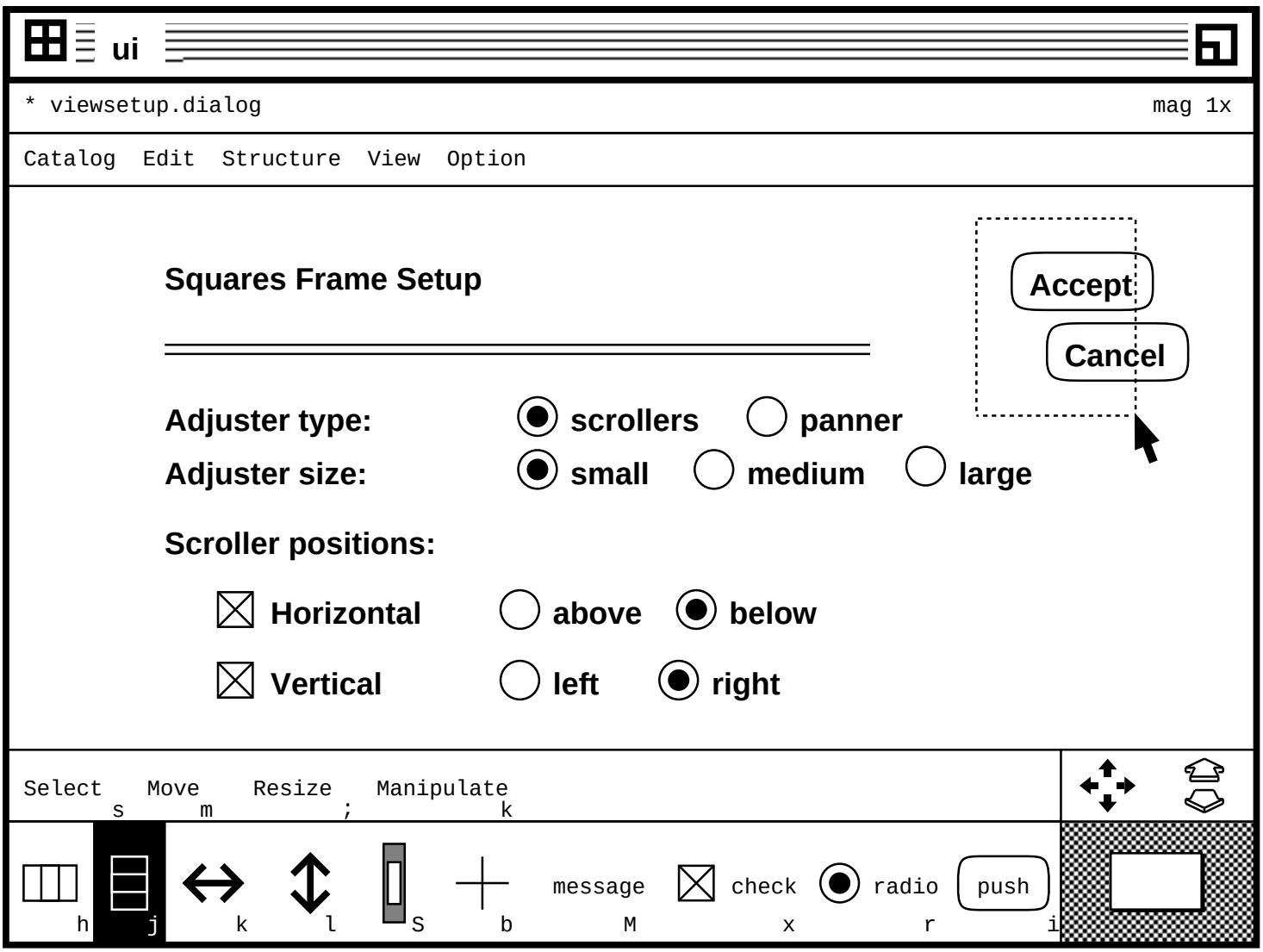
(d) pad-pin

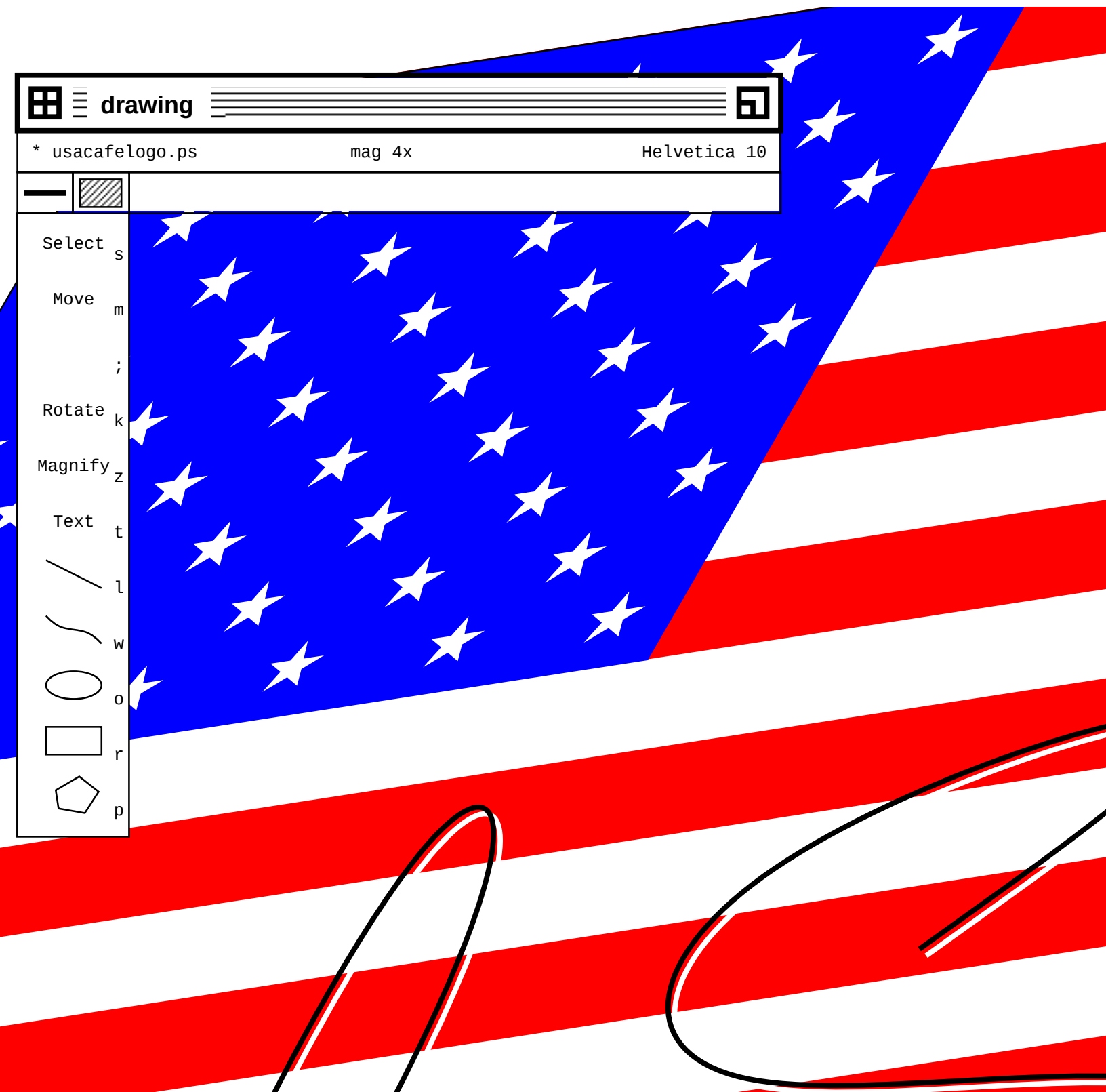

inverter
view
pin
subject
pin
subject
inverter
subject
local
data
polygon
subject
circle
subject


Selection
Component View
1
Viewer
2
5
Tool
3a
6a
3b
6b
4a,b,c
3
6
Manipulator
Command
8
7


G2
G1
G3
recurse
G1
G(2 || 3)
recurse
G(1 + (2 || 3))


schematic
* hitmiss.edif
mag 0.5x
Catalog Edit Structure Align View Options
_v1
Vbit_b_v1
ICacheDisable
blockMiss_v1
blockHit_v1
blockHit_v1
blockHit_v1_2
blockHit_v1_3
blockHit_v1_1
decode0_s1
decode1_s1
decode2_s1
decode3_s1
_v1_2
_v1_3
schematic tools
i
t
D
Q
Q
d
n
a
w
r
o
n
b
[
]
\
/


ui
* viewsetup.dialog
mag 1x
Catalog Edit Structure View Option
Squares Frame Setup
Accept
Cancel
Adjuster type:
scrollers
panner
Adjuster size:
small
medium
large
Scroller positions:
Horizontal
above
below
Vertical
left
right
Select
Move
Resize
Manipulate
s
m
;
k
h
j
k
l
S
b
message
check
radio
push
M
x
r
i


drawing
* usacafelogo.ps
mag 4x
Helvetica 10
Select s
Move m
;
Rotate k
Magnify z
Text t
l
w
o
r
p