An Object-Oriented Architecture for Constraint-Based Graphical Editing

Richard Helm, Tien Huynh, Kim Marriott, and John Vlissides
IBM T.J. Watson Research Center
Yorktown Heights, NY 10598

Abstract

Direct-manipulation graphics editors are useful tools for a wide variety of domains such as technical drawing, computer-aided design, application building, and music composition. Constraints can be a powerful mechanism for specifying complex semantics declaratively in these domains. To date, few domain-specific graphics editors have provided constraint-based specification and manipulation facilities. Part of the reason is that graphical editors are hard enough to develop without implementing a constraint system as well. Even though graphical editing frameworks can reduce the difficulty of constructing domain specific graphical editors, a fundamental problem remains: there do not exist general constraint solving architectures that are efficient enough to support highly interactive editing, yet suitably flexible and extensible to adapt to different editing domains.

Addressing this problem, we present an object-oriented architecture that integrates the graphical editing framework Unidraw with QOCA, a powerful new constraint solving toolkit. QOCA leverages recent advances in symbolic computation and geometry to support efficient incremental solving of simultaneous equations and inequations, while optimizing convex quadratic objective functions. QOCA also supports new kinds of constraint manipulation that have novel applications to graphical editing. QOCA exploits the implementation language to provide a convenient, object-oriented syntax for expressing constraints in the framework. The result is a generic and easily extended architecture for constraint-based, direct-manipulation graphical editing.

Keywords: graphical editing, simultaneous linear constraints, quadratic optimization, object-oriented frameworks

1 Introduction

Constraints are a powerful formalism in graphical user interfaces, both as an aid in interface development and as an interaction paradigm. Constraints can specify spatial and semantic relationships declaratively between objects in a user interface, while an underlying constraint solver will ensure that interface meets the specification. Previous work [2, 3, 5, 16, 20, 13, 21, 22] has established that constraint systems need at least the following capabilities to be effective in graphical user interfaces:

An earlier version of this paper appears in *Proceedings of the Third Eurographics Workshop on Object-Oriented Graphics*, Champéry, Switzerland, October 1992.

To appear in *Advances in Object-Oriented Graphics II*, Springer Verlag, 1993.

- multi-way constraints that can express at least simultaneous linear equations and inequations [7, 10]
- low latency and high-bandwidth feedback during direct manipulation [16]
- incremental addition and deletion of constraints [10, 17]
- the ability to detect causes of unsatisfiability for debugging inconsistent systems of constraints [10].
- semantic feedback during direct manipulation to indicate valid ranges for variables and movements of objects [12]
- graceful handling of underconstrained systems [16, 6]

Drawing packages, CAD systems, application builders, and diagrammatic editors are representative of a class of applications that could benefit particularly from constraints. These direct-manipulation graphics editors let a user manipulate visual manifestations of familiar objects to convey information in a domain, and they are usually responsible for maintaining spatial and semantic relationships between objects. Constraints are a natural way to specify these relationships and to ensure their maintenance. Responsibility can thus be transferred from the user to the constraint system, freeing the user to focus on more creative aspects of his task.

Yet few graphical editing systems employ constraints to any degree; those that do are research prototypes [1, 20, 2, 21]. Perhaps one reason is that graphical editors are notoriously difficult to implement, even with conventional user interface toolkits. Several frameworks for building graphical editors have been reported recently [23, 24, 27] that address this problem. These frameworks provide a generic software architecture that typically supports the following:

- the definition of domain-specific graphical components and their semantics
- mechanisms for composing and structuring components
- (reversible) operations on components
- specialized direct manipulation techniques
- persistence and externalization of application data

Experience with graphical editing frameworks [25] has shown that they simplify editor development for different domains compared with traditional user interface toolkits, which support only the controlling elements of an application (e.g., buttons, scroll bars, and menus). Unfortunately, current frameworks take little or no advantage of the power of constraints. This deficiency reflects the fact that constraint capabilities are absent from most hand-built graphics editors. Therefore combining the capabilities of a graphical editing framework with a general-purpose constraint system can make domain-specific, constraint-based graphical editing systems far simpler to develop.

Integrating graphical editing frameworks and constraint systems raises new issues and challenges. Some problems stem from the nature of constraint-based editing in a highly interactive environment: every component may be constrained, and the entire constraint system may need to be re-solved on every input event (e.g., mouse motion). Other problems concern the integration itself: constraints can be so basic to the operation of framework objects but so closely coupled with the constraint system that integrating them requires a rewrite of the framework, the constraint system, or both. Consequently, the integration strategy requires a careful design and implementation effort to minimize modifications to the existing systems.

This paper presents an architecture for constraint-based, direct-manipulation graphical editing that addresses these issues. The architecture integrates Unidraw [27], a graphical editing framework developed at Stanford University, and QOCA[1], a new object-oriented constraint-solving toolkit developed at IBM Research. Unidraw is an established graphical editing framework that already has limited constraint-solving capabilities. QOCA leverages recent results in symbolic computation and geometry to support efficient incremental and interactive constraint manipulation. Our goal is to combine these systems to provide a generic and easily extended architecture for constraint-based, direct-manipulation graphical editing.

This paper offers an overview of the integrated architecture and its subsystems. We begin by presenting examples of constraint-based editing that demonstrate the power and desirability of this paradigm in general and the advanced capabilities of QOCA in particular. Then we describe the Unidraw framework and how we integrated it with QOCA toolkit objects to allow constraint specification. Next we provide details of the QOCA implementation and the algorithms on which it is based. We conclude the paper with a summary of the architecture and discussion of future directions for this work.

2 Sample Applications

QOCA is an extensible constraint solving toolkit that supports incremental solving of simultaneous (in)equations and optimizes convex quadratic objective functions. QOCA also supports new kinds of constraint manipulation that have novel applications to graphical editing. The following examples illustrate how graphical user interfaces can benefit from this technology, both in implementing commonplace functionality and in providing new, constraint-based capabilities.

2.1 Graphical Connectivity

A simple application of constraints in user interfaces is to maintain connectivity between graphical objects. The top of Figure 1 depicts rectangle objects A and C and an arrow-headed line B. We wish to link the rectangles with the line so that the arrows and rectangles abut regardless of their relative positions, as shown at the bottom of the figure.

To ensure that the endpoints of the arrows remain inside the rectangles, we begin by specifying the inequality constraints

$$l_A \leq l_B \leq r_A,\ b_A \leq b_B \leq t_A,\ l_C \leq r_B \leq r_C,\ b_C \leq t_B \leq t_C$$

[1]Quadratic Optimization Constraint Architecture

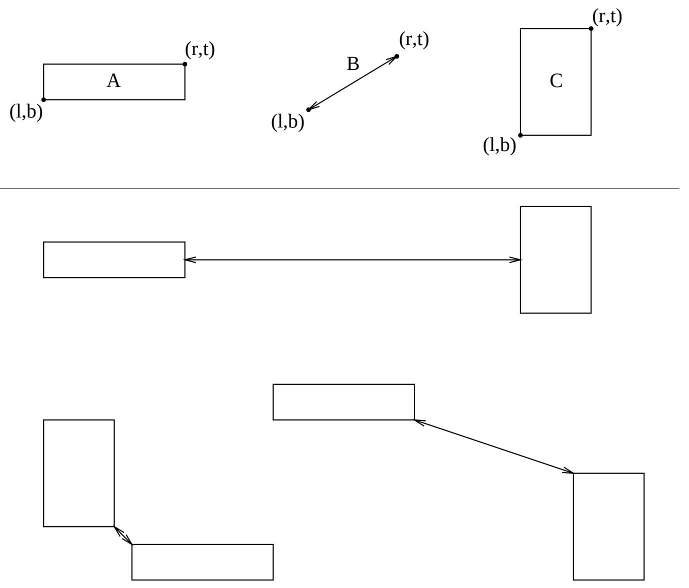


Figure 1: Boxes-and-arrows connectivity

over the variables defining the rectangles and line. These constraints are insufficient, however, because they do not guarantee that the arrowheads and rectangles abut properly. We can express these semantics as an optimization problem that minimizes the arrowheaded line's length:

$$\mathit{minimize}((l_B - r_B)^2 + (b_B - t_B)^2)$$

This expression, called an **objective function**, ensures that the line assumes the shortest distance between the rectangles. Objective functions are distinct from constraints: an objective function can only affect an underconstrained system. During constraint solving, therefore, QOCA will assign values to variables that minimize the objective functions. As the line's endpoints are constrained by the inequalities and governed by this objective function, the line will reorient and deform to accommodate the desired optimization.

2.2 Underconstrained Systems

So far we have used an objective function to specify an explicit design criterion: that the arrowheads and rectangles should abut. Less obvious is the need to clarify what happens when the user moves a rectangle, say rectangle A. Rectangle C may remain stationary and the line may stretch, for example, or the line may stay a fixed size while C moves the same distance as A. Without specifying a preference, either scenario is plausible; the system is underconstrained.

Handling underconstrained systems is a classic problem in constraint satisfaction. Stated generally, a constraint system must have a way to determine values for variables that are not constrained to take unique values. Requiring precisely constrained systems—that is, neither over- nor underconstrained—places too much responsibility on the user to create potentially complex yet error-free constraint specifications.

One way to deal with this problem is with constraint hierarchies [4], in which lower priority constraints express default behavior. The constraint solver selects (either arbitrarily or via comparators) non-required constraints to include in the solution. The primary difficulty with constraint hierarchies is in defining appropriate hierarchies (and comparators) so that, as constraints and defaults from different parts of the hierarchy are selected, the resultant solutions are continuous with respect to each other.

In contrast, the process of minimizing objective functions effectively selects values for underconstrained variables. The key idea is to continually refine the objective functions during direct manipulation so that new solutions are always as close as possible to the old. The objective functions provides a declarative way to express exactly what "closeness" means.

Returning to our connectivity example, we can make the system behave predictably when a rectangle is moved by introducing additional objective functions. Suppose we require that the rectangles deform and move as little as possible during direct manipulation. This requirement is captured via the objective functions

$$\mathit{minimize}(\ (l_A - l_{0A})^2 + (b_A - b_{0A})^2 + (r_A - r_{0A})^2 + (t_A - t_{0A})^2\ )$$

and

$$\mathit{minimize}(\ (l_C - l_{0C})^2 + (b_C - b_{0C})^2 + (r_C - r_{0C})^2 + (t_C - t_{0C})^2\ ),$$

which state that the new values for the variables defining the rectangles $(l_A, r_A, ..., l_C, r_C, ...)$ should remain as close as possible to their current values given by the constants $(l_{0A}, r_{0A}, ..., l_{0C}, r_{0C}, ...)$. By updating these constants at the start of each direct manipulation, we ensure that the rectangles will be deformed no more than necessary (and typically not at all).

Through objective functions, QOCA supports the "Principle of Least Astonishment": it guarantees that the rectangles will move as little as possible should the system ever be underconstrained. Moreover, because solutions are selected via objective functions, which are continuous, solutions generated by the solver during direct manipulation are likewise continuous with respect to each other. The system will not generate sudden discontinuous jumps between solutions.

2.3 Constrained Layout

Simultaneous linear constraints are a convenient way to express graphical layout. For example, Figure 2 graphically depicts constraints that left-align three small boxes and center the topmost small box horizontally in the surrounding box. Vertical constraints provide whitespace between the boxes and the surrounding box, ensuring that it is large enough to contain the smaller boxes.

Given that the dimensions of the surrounding box are $W \times H$ and the dimensions of the top most box are $w \times h$, the constraints that capture this layout are

$$3h + 2Y_1 + 2Y_2 = H,$$

$$2X_1 + w = W,$$

$$X_1 = Y_1.$$

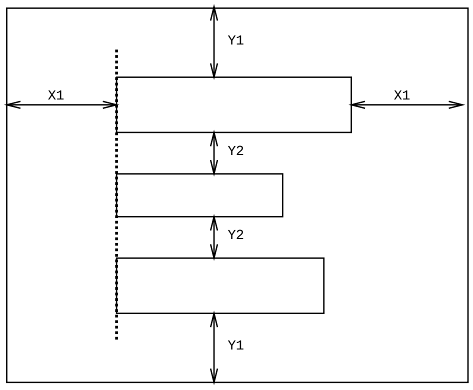


Figure 2: Example layout

These constraints form a system of simultaneous linear equations in three unknowns, thereby demonstrating the need for a constraint solver that can solve simultaneous linear equations. In general, a constraint solver based on local propagation is not adequate whenever constraints express a dependency between the x and y dimensions.

Objective functions add even more power as they can express layout in terms of a "spring" metaphor, in which layouts can deform in precise and intuitive ways. The objective function measures the potential energy of a particular configuration, and the best layout is the one that minimizes this potential.

More formally, a spring S is specified by its minimum length L_{min}, its rest length L_{rest}, its maximal length L_{max}, and its energy coefficients when compressed E_{comp} and stretched E_{str}. Letting x be the extent of S, x must obey the constraints $L_{min} \leq x \leq L_{max}$, and the energy of S is:

$$e(x) = \begin{cases} E_{comp}(L_{rest} - x) & \text{if } L_{min} \leq x \leq L_{rest}; \\ E_{str}(x - L_{rest}) & \text{if } L_{rest} \leq x \leq L_{max}. \end{cases}$$

As the acronym suggests, QOCA is designed to solve quadratic optimization problems. At first glance it is not clear that minimization of e can be handled by our system, because it is piecewise-linear rather than quadratic. However, we can transform this into a quadratic optimization problem (actually a linear optimization problem) by introducing two new variables: x_{comp}, the amount the spring is compressed, and x_{str}, the amount the spring is stretched. The associated constraints are

$$L_{min} \leq x \leq L_{max}, \qquad x_{comp} \geq 0, x_{str} \geq 0, \qquad x = L_{rest} - x_{comp} + x_{str}$$

and the energy of S is given by

$$e'(x, x_{comp}, x_{str}) = E_{comp}x_{comp} + E_{str}x_{str}.$$

Now for all $L_{min} \leq x \leq L_{max}$, the minimum value of $e'(x, x_{comp}, x_{str})$ is the same as that of $e(x)$—the minimum value of e' occurs when both x_{comp} or x_{str} is zero. Thus the two problems have the same solution.

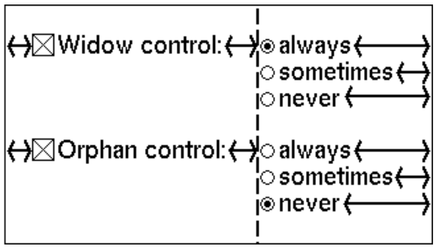


Figure 3: Dialog box with horizontal spring constraints superimposed

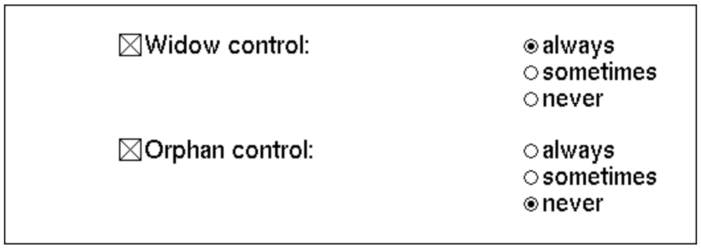


Figure 4: Resized dialog box with incorrect layout behavior

2.4 Diagnosing Anomalies

One of the problems with declarative specifications in general and constraints in particular is that it can be difficult to ascertain the cause of unexpected behavior. The larger the set of constraints, the more likely it is that the system is either over- or underconstrained, inconsistent, or otherwise at odds with desired semantics. Any system that supports nontrivial constraint specifications should also offer mechanisms for diagnosing anomalous behavior.

For example, consider interactive layout in a user interface builder. The dialog box in Figure 3 consists of check boxes and radio buttons aligned with spring constraints (arrows) and an alignment constraint (dashed vertical line). The builder is displaying only horizontal constraints for simplicity.

Now the interface designer would like the whitespace in between and around the buttons to grow and shrink equally as the dialog is resized. When the designer resizes the dialog, however, there radio buttons stay a fixed distance away from the right edge (Figure 4). To diagnose this problem, the builder can use the constraint system to determine the causes of unsatisfiability.

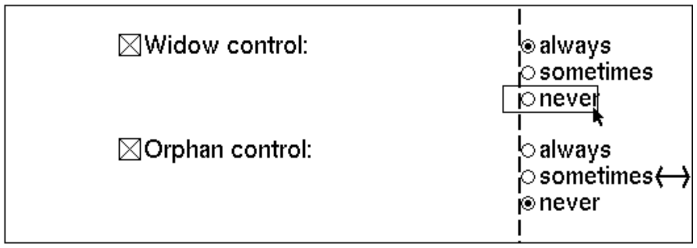
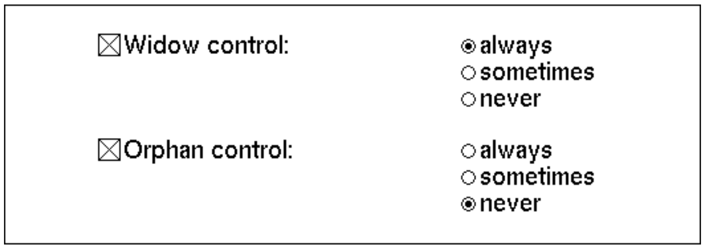


Figure 5: Diagnosing incorrect layout behavior



☒Widow control: [✓ always] sometimes never

☒Orphan control: always sometimes [✓ never]

Figure 6: Corrected layout behavior

A natural interface to this functionality would let the user try to move a misplaced object. Then the system can provide feedback to help explain why the object cannot occupy its proper place. QOCA supports this diagnosis by providing primitive operations for testing the satisfiability of constraints and detecting the causes of their unsatisfiability.

In Figure 5 the user is trying to move one of the misplaced radio buttons. The system responds by displaying graphically the constraints that keep the button from moving: the left-alignment constraint and the spring to the right of the second “sometimes” radio button. This suggests to the user that a spring constant is incorrect. When he examines the spring’s attributes he discovers that its spring constant is zero when it should be identical to that of the other springs. The user can then modify this constant in the builder, and the resulting interface exhibits the proper resize semantics (Figure 6).

3 Unidraw Framework Integration

Unidraw is an object-oriented framework for building direct-manipulation graphical editors. It is a part of InterViews [15], a comprehensive set of programming abstractions and tools for the design and implementation of workstation applications. Unidraw partitions the common functionality of graphical editors into four major class hierarchies:

1. **Components** represent the elements in a graphical editing 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. The user arranges 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. 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** define a one-way mapping between components and their representation in an outside format. 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.

Partitioning editor functionality into components, commands, tools, and external representations is the foundation of the Unidraw architecture. We will introduce additional Unidraw classes as they become relevant.

3.1 Basic Integration

The obvious application of QOCA in Unidraw was as a replacement for Unidraw's special-purpose geometric constraint solver, which enforces connectivity semantics between components. However, making QOCA's full power available to the Unidraw programmer adds a new dimension to the framework's capabilities—support for constraint-based graphical editing. In this section we discuss several key aspects of the integration of these two systems.

Unidraw can leverage constraints in two ways: (1) constraints can define attributes of new user-defined components, for example, to define the center point of a rectangle in terms of its corners; and (2) constraints can appear as graphical components to be manipulated in their own right. Before describing how this is done in Unidraw, we must first consider how to specify constraints in QOCA.

3.1.1 Expressing Constraints in QOCA

QOCA makes constraints, objective functions, and variables first class objects, and it provides a natural syntax to define these objects directly in the programming language, in this case C++. QOCA defines constraints and variables using the arithmetic and relational operators of C++. This requires heavily overloading these operators, but the result is a natural syntax for declaring constraints.

The following example, written in C++, captures the relationship between temperature scales in Fahrenheit, Celsius, and Kelvin as constraints over variables representing these quantities. It makes use of three classes, **CVariable**, **Constant**, and **Constraint**.

```
CVariable fahr, cent, kelv;
Constant Freezing = 32.0;
Constant AbsoluteZero = 273.13;

Constraint c1 =  fahr - Freezing == cent * 1.8;
Constraint c2 =  cent == kelv - AbsoluteZero;
```

Constraints are added to the system merely by instantiating constraint objects. QOCA ensures that the values of CVariable objects adhere to the constraint specification. Through operator overloading, QOCA evaluates the expressions in the constraints and returns instances of class **Expression**. Expressions are objects that capture the abstract syntax tree of the expressions in the constraint. These structures can be then assigned as in the case above or can be manipulated symbolically by other objects.

Objective functions define expressions to be minimized or maximized. In this example, suppose we want to minimize the difference between the variable representing Fahrenheit and freezing. We can express this requirement with an instance of class **Objective**:

```
Objective o = Minimize(fahr*fahr - Freezing);
```

Minimize is a function that takes an Expression as an argument and returns an instance of class Objective. The Objective object o establishes an objective function that QOCA must consider in solving the constraint system.

It is often necessary to assign values to the variables and then have these values automatically propagate to the constrained variables via the constraint solving class **ConstraintSolver**. But the allowed values of CVariables are governed by Constraint and Objective objects. Consequently, assigning a value to a variable is not a direct assignment—the assigned value may be inconsistent with some constraints or may not satisfy some objective. Instead QOCA treats an assignment to a variable as a *request* that the CVariable take that value. Only when the constraint system is solved are the requested values considered. Then the solver propagates computed values back to the variables, notifying them that they have changed. In solving the constraints, the requested values act as parameters to the system, and all other variables will depend on them. Thus we can write

```
cent = 95;
```

and the solver will assign the correct values to `fahr` and `kelv` whenever the Solve method (i.e., `ConstraintSolver::Solve`) is called.

The classes CVariable, Constant, Constraint, Objective, and ConstraintSolver are the primary base classes visible to users in QOCA, and they do not depend on Unidraw in any way. Additional classes integrate QOCA and Unidraw without compromising their independence, as we demonstrate in subsequent sections.

3.1.2 Constraint State Variables

CVariable objects play a central role in the specification of constraints. Clearly if Unidraw is to support general constraint specification, it must surface CVariable to the users of the framework. Complicating the issue is Unidraw's notion of a **state variable**. State variables are persistent values that can define a graphical user interface for viewing and modification, and they can change automatically through Unidraw's support for dataflow. Components commonly have one or more state variables that store user-accessible state. For example, an inverter component in a schematic capture system may use state variables to define the logic levels at its input and output terminals.

State variables thus play some of the same roles as constraint variables, and vice versa. To avoid introducing dependencies between Unidraw and QOCA, we derive a new class, **constrained state variable**, or **CSVar**, from both the StateVar state variable base class

```
class ConstrainedRectComp : public Component {
    CSVar _left, _right, _centerx;
    CSVar _top, _bottom, _centery;
    Constraint _Xconstraint, _Yconstraint;
...
};

ConstrainedRectComp::ConstrainedRectComp () {
    _Xconstraint = _left + _right == 2.0 * _centerx;
    _Yconstraint = _top + _bottom == 2.0 * _centery;
}
```

Figure 7: Excerpt from ConstrainedRectComp class declaration and definition

and from CVariable. CSVar inherits both the constraint semantics of CVariable and the persistence and other Unidraw-oriented aspects of StateVars without introducing dependencies between the base classes.

The mechanism for keeping CSVars consistent with the constraint system builds upon both the QOCA and Unidraw architectures. Ordinary CVariables receive requests for change and later have their values updated in one pass via `ConstraintSolver::Solve`. However, Unidraw programs do not call this operation directly. Unidraw already defines a global Update operation that synchronizes the application and the state of its constituent objects, which may involve solving connectivity constraints, repainting the screen, and so on. We simply extended this operation to invoke Solve on the constraint solver.

CSVars have the added need to notify their enclosing component (if any) whenever they change. Therefore the CSVar class adds protocol for associating one or more components with an instance. CSVar also extends CVariable's Update operation to notify its associated components of a change in its value.

3.2 Constraining Components

To place constraints on components, variables that represent attributes of components must be defined in terms of CSVars. This lets us establish constraints between an object's internal values (i.e., **internal constraints**) and across objects (**external constraints**).

Internal constraints simplify a component's definition. Code previously required to maintain relationships between member variables is now delegated to the solver through the constraints. Internal constraints also simplify alternate definitions of objects. For example a rectangle can be defined by a center point and one corner or by opposite corners. Consider the class **ConstrainedRectComp** shown in Figure 7, which defines six member CSVars representing its opposing corner points and its center. Note how internal constraints in the constructor define the center point in terms of its corners.

To present constraints graphically as components, we derive a new base class of graphical component called **ConstraintComp**, which defines an appearance and manipulation semantics for constraints. Derived classes add semantics for particular constraints. For example, the derived class **PointEqualityComp** takes two pairs of CSVars representing two points and establishes an equality constraint between them:

```
class PointEqualityComp : public ConstraintComp {
public:
    PointEqualityComp(CSVar&, CSVar&, CSVar&, CSVar&);
    ...
private:
    Constraint _XConstraint, _YConstraint;
};

PointEqualityComp::PointEqualityComp (
    CSVar& x1, CSVar& x2, CSVar& y1, CSVar& y2
) {
    _XConstraint = _x1 == _x2;
    _YConstraint = _y1 == _y2;
...
}
```

In general, graphical components in Unidraw use structured graphics objects [26] to depict themselves graphically. PointEqualityComp maintains a structured graphic object to present its constraint to the user in an intuitive manner.

ConstraintComp objects are often constructed by tools that query components for their CSVars using Unidraw's interpreted command mechanism. The tool provides the appropriate direct manipulation semantics, such as dragging or stretching a line between two points. Once a tool has obtained the required CSVars, it returns a command that pastes the component into the drawing and establishes the proper external constraints.

For example, the tool that creates an EqualityPointComp between two points asks the two components containing these points to return the appropriate CSVar objects. Then it instantiates an EqualityPointComp, passing the CSVars to the constructor. Finally, it returns a PasteCmd object containing the new instance. Later in the paper we discuss in more detail how we exploit Unidraw's direct manipulation model to involve constraint solving and how undoable commands containing constraints work.

3.3 Supporting Undo/Redo

In integrating QOCA and Unidraw, it is important to retain full undo and redo capabilities. Two semantics are essential:

1. Constraints and optimization functions can exist without affecting the constraint system.
2. The constraint system can be queried for its current state, and it can revert to exactly that state at an arbitrary point in the future.

3.3.1 Enabling and Disabling

The first semantics implies that an instantiated constraint or objective does not necessarily affect the behavior of the system: only an *enabled* constraint or objective may have an affect. This is relevant to the undo model in that structural changes to the system may have to be undone.

For example, suppose the user deletes the right-hand rectangle in Figure 10. In standard Unidraw this would be accomplished via a DeleteCmd, which removes the component being deleted from its enclosing structure *but does not destroy it.* Instead, the command stores both the component and its position in the structure. If the DeleteCmd is later undone, it reinserts the component in the structure at the proper place. It is far easier and cheaper to save the component that to reconstruct it, since a component can be arbitrarily complex.

Similarly, it is better to disable and enable constraints and objectives than it is to destroy and recreate them. When the constrained rectangle is removed, it disables all the constraints and objectives that affect its CSVars; if it is subsequently pasted or reinserted into the display, it simply enables them again.

The Constraint class in QOCA provides protocol for enabling and disabling its instances. The ability to switch constraints on and off at will is one of the novel features of QOCA and is intrinsic to supporting undo/redo semantics. It requires efficient incremental addition and deletion of constraints. No other constraint solving system that we know of provides this capability for the class of constraints that QOCA solves.

3.3.2 Saving and Restoring System State

The second semantics ensures that the editor does not suffer from hysteresis or round-off errors as operations are undone and redone. There is no guarantee, for example, that undoing a state-changing operation (such as a move) by performing the inverse operation will return the system to exactly the original state. Round-off errors can accumulate even in ostensibly well-behaved systems.

Hysteresis can occur in underconstrained systems as constraints are added and deleted. Consider the scenario in Figure 8. The endpoints of two lines are constrained to coincide via an equality constraint, which is subsequently removed. Because the lines are underconstrained, the top portion of stage 3 is a valid configuration. However, to support undo and redo, the display must be restored to the configuration at the bottom of stage 3; otherwise unpredictable results will occur as the user performs additional undo commands.

To ensure stability, state-changing commands query the constraint engine for **Solution** objects both before and after they carry out their operations. A Solution object captures the state of the constraint solver at a particular instant. On undo, these commands then direct the constraint engine to adopt the original (i.e., pre-execution) solution. On redo, they set the post-execution solution. The constraint system is thus guaranteed to compute the same values after arbitrarily many undo and redo operations.

3.4 Constraining Direct Manipulation

It is important to enforce constraints and to see their effects during direct manipulation. Otherwise, the result of the manipulation may not correspond to the feedback provided.

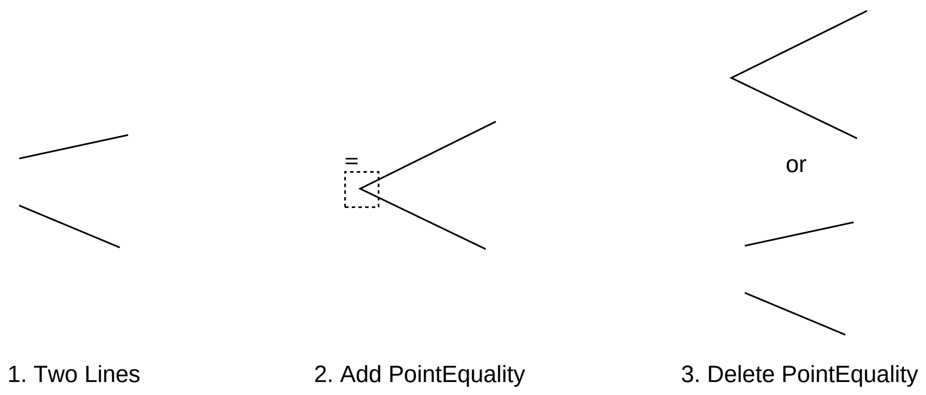


Figure 8: Hysteresis in underconstrained systems

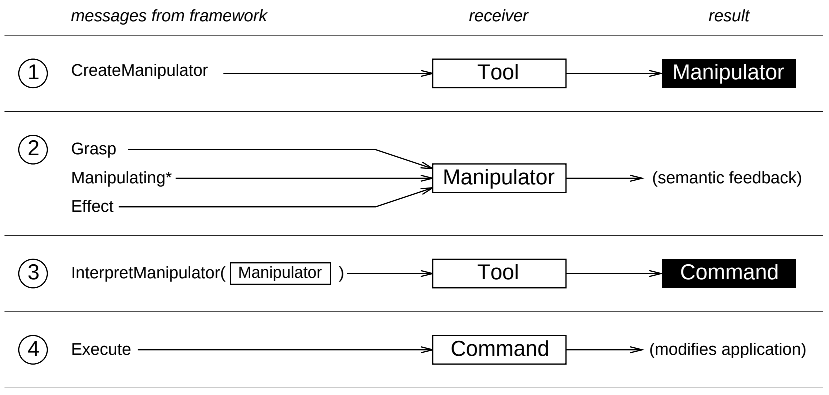


Figure 9: Basic direct manipulation sequence

For example, a drawing editor may let a user stretch an unconstrained rectangle arbitrarily. But a rectangle that is constrained to be square should stay that way as it is stretched, thereby reflecting the constraint in the manipulation. This section summarizes Unidraw's direct manipulation model and how it is integrated with QOCA to support constrained direct manipulation.

3.4.1 Unidraw Direct Manipulation Model

Tools are fundamental to Unidraw's direct manipulation model. The user *grasps* and *wields* a tool to achieve a desired *effect*. The effect may involve a change in component or other application object state, or it may change the way components are viewed, or there may be no effect at all (if, for example, the tool is used in an inappropriate context). Most tools generate animated effects as they are wielded to provide semantic feedback to the user.

Tools employ **Manipulator** objects and commands to handle the mechanics of the direct manipulation and enact its outcome. A manipulator abstracts and encapsulates the code that generates semantic feedback. Manipulator provides a standard interface to an abstract state machine that defines interaction semantics. Commands actually carry out the intent of the manipulation and permit its undoing and redoing.

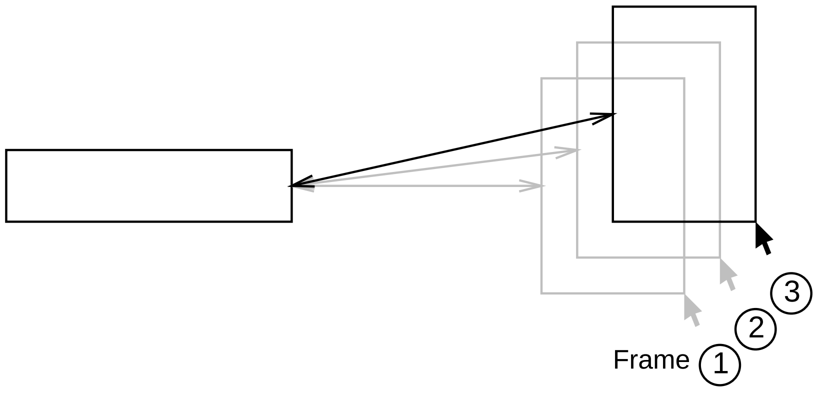


Figure 10: Frames of animation produced by `CDragManip::Manipulating`

Figure 9 depicts the four basic stages of a direct manipulation:[2]

1. The active tool receives a CreateManipulator message from the framework in response to user input. The tool creates an appropriate manipulator and returns it to the framework.

2. The framework exercises the manipulator in response to user input:

   (a) **Grasp** instructs the manipulator to prepare to generate semantic feedback. In response, the manipulator typically initializes internal state associated with the ensuing animation.

   (b) The framework issues **Manipulating** messages repeatedly in response to user input until the manipulator indicates that manipulation has ceased. Each call to Manipulating usually generates a new frame of animation.

   (c) **Effect** instructs the manipulator to finalize its internal state following the direct manipulation.

3. The framework asks the active tool to interpret the manipulator it had created via the InterpretManipulator message. The manipulator returns a command in response.

4. The framework executes the command to carry out the user's intent.

3.4.2 Integrating Constraints

To enforce constraints during direct manipulation, Unidraw may solve the constraint system on every input event. The state of the system thus changes *before* manipulator interpretation at stage 3 above. This contrasts with interactions that do not involve constraints, wherein the application is affected only after manipulation has ended.

[2]This discussion omits many details of Unidraw's direct manipulation model to focus on the parts that relate directly to its interplay with QOCA. A detailed description of the model appears elsewhere [27].

Consider the boxes-and-arrows connectivity example from Section 2.1. Figure 10 depicts three frames of animation produced when the user moves the right-hand rectangle with a **MoveTool**. In this case, each frame is generated by a call to Manipulating on an instance of **CDragManip** (short for "constrained drag manipulator"), the manipulator that the MoveTool created. MoveTool initializes the CDragManip with the CSVars that define the lower-left and upper-right corners of the rectangle being moved.

CDragManip's Grasp operation records the current values of the system's CSVars in a Solution object. Each subsequent call to Manipulating generates a frame of animation: CDragManip requests changes to the rectangle's CSVar values each time the cursor moves during manipulation. Then CDragManip calls `Unidraw::Update`, which solves the constraint system and updates the display. Unidraw thus maintains the connectivity constraints during direct manipulation simply by treating each frame of the animation as an incremental change to the constraint system.

QOCA's incremental parametric constraint solver performs each step of the manipulation efficiently. It treats the variables being manipulated (that is, those that receive requests to change value in the call to Manipulating) as parameters. The solver minimizes the manipulation of the constraints by solving parametric quadratic optimization problems incrementally. Most often it computes new values of variables that depend on the parameters directly—constraint manipulation occurs relatively infrequently.

A subtle point in this strategy concerns when to change the objective functions to reflect the rectangle's final position. Recall that the system includes objective functions (expressed via Objective objects) that minimize the distance between the rectangle's initial and final positions. After manipulation it is necessary to adjust the constants appearing in these objectives to make their values correspond to the new position.

We refer to this process as **leapfrogging** the objective functions at the end of each manipulation step to catch up to the current values of the CSVars they effect. This adjustment takes place in the command that records the overall effect of the direct manipulation. When the framework issues the InterpretManipulator message (passing the CDragManip as an argument) to the MoveTool, it responds by producing a **CMoveCmd**, or "constrained move command." This command's purpose is twofold: (1) to adjust the rectangle component's objectives, and (2) to provide a record of the manipulation should it be undone or redone later. If the command is undone (or redone), CMoveCmd moves the rectangle back to its original position (or to its new position) and adjusts the rectangle's objectives accordingly.

4 QOCA Internals

4.1 Architectural Overview

QOCA has the four main components illustrated in Figure 11:

1. A **solver**, which adds or deletes constraints while incrementally maintaining the solved form. The test for satisfiability is a byproduct of maintaining the solved form.

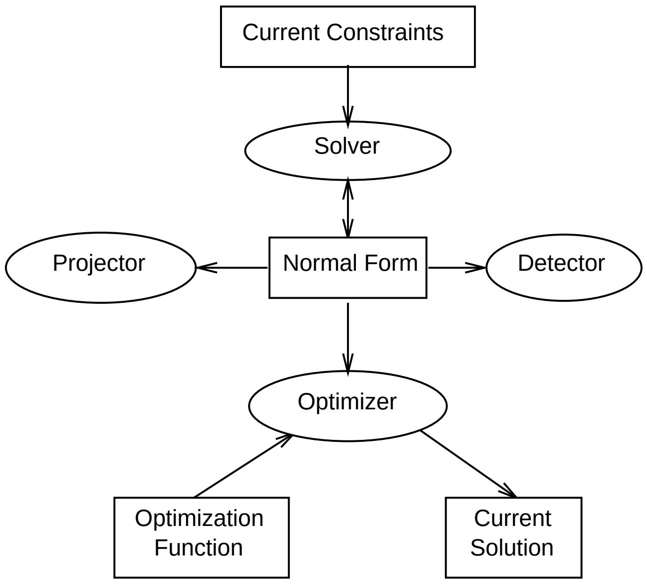


Figure 11: QOCA architecture

2. A **detector**, which takes a constraint that is inconsistent with the current constraints and identifies sources of the inconsistency.

3. A **projector**, which takes a set of variables and projects the current constraints onto those variables.

4. An **optimizer**, which recomputes the current solution given requested values for parameters. It finds values for non-parametric variables that both satisfy the constraints and minimize the optimization problems.

In addition, QOCA maintains a record of the **current constraints** and the **current optimization function**. It also maintains a **current solution**, the assignment of variables that satisfies the current constraints and minimizes the current optimization function.

QOCA's architecture is designed to be flexible. It permits experimentation with different classes of constraints and domains (e.g., reals, booleans, etc.), different constraint solving algorithms for these domains, and different representations for objects in these domains. QOCA's object-oriented design allows parts of the system to be varied independently of others. For example, real numbers, currently represented as doubles, can be changed to infinite precision or rational representations simply by changing the definition of a single class.

Moreover, as improved algorithms and solvers are developed, existing algorithms can be replaced with minimal disturbance. This modularity highlights an advantage of using global constraint solvers such as QOCA. Systems that employ local propagation [16, 19] often distribute constraint solving methods throughout the system, relegating to each object the responsibility to solve its own constraints. This makes it difficult to exploit efficient representations and constraint solving algorithms in these systems.

4.2 Implementation

Here we describe briefly the algorithms and techniques used in the constraint system. A complete description of QOCA is forthcoming [11], and preliminary performance measurements have already been reported [10].

QOCA leverages the well-developed theory and efficient algorithms that have been investigated extensively in operations research for handling linear constraints. The Simplex algorithm is the key technique used in the system. The Simplex is an efficient symbolic manipulation technique for testing satisfiability and for optimizing linear constraints. QOCA also takes advantage of new results from symbolic computation, both for efficient representation of constraints and in incremental algorithms for constraint manipulation. QOCA currently supports linear arithmetic constraints, that is, linear equalities and inequalities over the real numbers, and the optimization of convex quadratic functions.

4.2.1 Normal Form

Almost all constraint manipulation in QOCA is on the **normal form** of the current constraints. The normal form is essentially a compiled non-redundant representation of the constraints in which as many variables as possible are eliminated. Elsewhere [10] we discuss in detail some of the ramifications of normal forms for constraint solving. Briefly, the normal form is constructed as follows. Assume that we have a set of linear equalities and inequalities over the variables $x_1, ..., x_n$. We can rewrite them into a set of equalities by replacing each inequality

$$a_1x_1 + a_2x_2 + \cdots + a_nx_n \leq b$$

by

$$a_1x_1 + a_2x_2 + \cdots + a_nx_n + s = b$$

where s is a distinct new **slack variable** and $s \geq 0$. The normal form of this rewritten set is obtained by eliminating as many of the original variables $x_1, ..., x_n$ as possible using Gauss-Jordan elimination. The remaining equations will contain only slack variables. These equations are collected, and the Simplex algorithm is used to find their **feasible basic form**. Thus the normal form consists of two sets of equalities. The first set, called the **defining equations**, contains the equations used to eliminate the original variables $x_1, ..., x_n$. The second set, called the **slack equations**, is a basic form of the equations in slack variables.

In practice we do not explicitly compute the normal form of a constraint set C. Rather, we represent the normal form implicitly as the product MC, where M is an invertible matrix called the **quasi-inverse**. M is essentially the product of the elementary row operations used to compute a normal form from C. One advantage of this implicit representation is that M is smaller than C, which means performing a pivot on M is cheaper than performing one on C.

4.2.2 Adding and Deleting Constraints

The main advantage of the quasi-inverse representation, however, is that M captures how the original constraints were used to obtain the solved form. This lets the solver (re)compute a normal form efficiently when a constraint is deleted. We handle the addition of constraints and incremental computation of a new normal form with standard techniques in sensitivity analysis [18]. The expected cost[3] is proportional to the cost of one pivot in M. The actual cost of this pivot depends on the representation of M. With a non-sparse representation, the actual cost is $O(n^2)$, where n is the number of original constraints. The cost should be significantly less with a representation that preserves the sparseness in the original system. At present, however, the system uses a non-sparse representation for simplicity.

4.2.3 Causes of Unsatisfiability

Each time a constraint is added to the solver, it is first simplified using the defining equations. If the new constraint becomes a contradiction after the simplification, the causes of the unsatisfiability can be traced back immediately using the quasi-inverse—indices of the non-zero elements in the row of the quasi-inverse corresponding to the new constraint indicate the constraints that contradict it. When the simplified constraint contains only slack variables, it is added into the slack equations; then the Simplex is activated to solve the system. If this system is infeasible, one can apply the technique proposed by Gleeson and Ryan [8] to identify the minimally infeasible subsystems and hence decide which constraints should be removed to obtain feasibility.

It follows from the construction of the normal form that the number of slack equations is less than or equal to the number of inequalities in the original system. This is critical because (except for constraint deletion and addition) all operations in the constraint system have cost proportional to the number of slack equations rather than the size of the original system. The defining equations are only used to transform solutions in terms of the slack variables to solutions in terms of the original variables.

4.2.4 Projection

Given a set of variables to project on, the projector first combines the defining equations for these variables with all the slack equations. Then a projection algorithm computes the actual projection. Since the projection space is assumed to be small, we use a projection algorithm called the Convex Hull Method [14], which is based on a geometric approach. For small projection spaces, it is much faster than other projection algorithms based on algebraic manipulation. It uses the Simplex algorithm repeatedly to compute the convex hull of the projected constraints.

4.2.5 Quadratic Optimization

The algorithm used for optimizing convex quadratic functions is a variant of the Simplex algorithm; see Murty [18] for details. When a new constraint is added, the optimization problem is (re)solved to find the new solution. During direct manipulation, however, a sequence of very similar optimization problems are solved in which the values of parameters change only slightly. In this case we solve the optimization problem incrementally, making

[3]In fact, we use the Dual Simplex, and so adding a constraint has in the worst-case exponential complexity. However in practice, the Simplex algorithm has incremental cost proportional to the number of constraints added. In fact, the Simplex is routinely used in problems with many millions of constraints, and it is often preferred to the more complex interior point methods that have polynomial worst case complexity.

use of the basis of the last solution as the starting basis for the new optimization. If the parametric values are sufficiently close, the cost of each optimization is expected to be one pivot on the slack equations. In fact, during direct manipulation we often know that the optimal solution for the initial parameter values is just the current solution. This means that the initial basis can be constructed efficiently, since we know which variables are basic.

To our knowledge, optimization functions are a new technique for handling underconstrained systems in user interface applications. This approach is related to Witkin's system for graphical animation [28], which uses functions to define the total energy of a system. In this system a global solver tries to minimize the total energy during manipulation to control the movement of graphical objects. Our constraint toolkit can be viewed as combining an energy model approach and pure constraints.

5 Conclusion

Basic constraint technology has matured to the point that highly interactive applications can incorporate constraints in both their interface and their implementation. Concurrently, advances in reusable user interface frameworks have made graphical editing systems easier to implement. Our work has focused on combining these developments to create a powerful, object-oriented architecture for constraint-based graphical editing.

We have integrated QOCA, an extensible constraint system, with Unidraw, a framework for building direct-manipulation graphical editors. Critical to QOCA's effectiveness in supporting constraints in Unidraw-based applications are its ability to solve simultaneous equations and inequations, optimize convex quadratic objective functions, incrementally add and delete constraints, incrementally re-solve parametric quadratic optimization problems, and detect causes of unsatisfiability in inconsistent constraints. An important goal of the integration was to avoid compromising existing Unidraw capabilities such as its direct-manipulation model and unlimited undo/redo. QOCA's powerful linear arithmetic constraints, constraint manipulation techniques, and sound theoretical foundation make QOCA an advanced platform for interactive constraint-based editors.

We plan to use QOCA extensively in the future. One project will extend key glyphs in InterViews, such as trays and glue, to be implemented in terms of QOCA constraints. QOCA will also serve as a basic element in our pen-based visual language parsing system [9]. We will continue research into new algorithms for manipulating constraints, QOCA being a good vehicle for exploring new algorithms. We also hope to make QOCA freely available, thereby promoting more widespread applications for constraints.

References

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

[2] A. Borning. The programming language aspects of ThingLab – a constraint-oriented simulation laboratory. *ACM Transactions on Programming Languages and Systems*, 3(4):343–387, October 1981.

[3] A. Borning and R.A. Duisberg. Constraint based tools for building user interfaces. *ACM Transactions on Graphics*, 4(4), 1986.

[4] A. Borning, M. Maher, A. Martindale, and M. Wilson. Constraint hierarchies and logic programming. In *International Conference on Logic Programming.* MIT Press, 1989.

[5] D. Epstein and W.R. Lalonde. A smalltalk window system based on constraints,. In *Object-Oriented Programming Systems, Languages and Applications Conference*, pages 83–94. ACM Press, 1988.

[6] B. N. Freeman-Benson. Kaleidoscope: Mixing objects, constraints, and imperative programming. In *Object-Oriented Programming Systems, Languages and Applications Conference*, pages 77–88, 1990.

[7] B. N. Freeman-Benson and Alan Borning. Integrating constraints with an object-oriented language. In *European Conference on Object-Oriented Programming*, pages 268–286, 1992.

[8] J. Gleeson and J. Ryan. Identifying minimally infeasible subsystems of inequalities. *ORSA Journal on Computing*, 2(1):61–63, Winter 1990.

[9] R. Helm, K. Marriott, and M. Odersky. Building visual language parsers. In *Computer Human Interaction (CHI)*, pages 105–112. ACM Press, 1991.

[10] Richard Helm, Tien Huynh, Catherine Lassez, and Kim Marriott. A linear constraint technology for user interfaces. In *Graphics Interface*, pages 301–309, Vancouver, Canada, 1992.

[11] Richard Helm, Tien Huynh, Kim Marriott, and John Vlissides. QOCA: An extensible object-oriented constraint solving toolkit. Technical Report In Preparation, IBM T.J. Watson Research Center, 1992.

[12] Scott E. Hudson. Adaptive semantic snapping—a technique for semantic feedback at the lexical level. In *ACM CHI '90 Conference Proceedings*, pages 65–70, April 1990.

[13] D.R. Olson Jr. and K. Allan. Creating interactive techniques by symbolically solving geometric constraints. In *ACM/SIGGRAPH/SIGCHI User Interface Software Technologies Conference*, pages 102–107, Snowbird, Utah, October 1990.

[14] C. Lassez and J.-L. Lassez. Quantifier elimination for conjunctions of linear constraints via a convex hull algorithm. Research Report RC 16779, IBM T.J. Watson Research Center, 1991. To appear, *Symbolic and Numerical Computation—Towards Integration*, Kapur and Mundy editors, Academic Press.

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

[16] John H. Maloney, Alan H. Borning, and Bjorn N. Freeman-Benson. Constraint technology for user interface construction in ThingLab II. In *ACM OOPSLA '89 Conference Proceedings*, pages 381–388, New Orleans, LA, October 1989.

[17] John H. Maloney, Alan H. Borning, and Bjorn N. Freeman-Benson. An incremental constraint solver. *Communications of the ACM*, 33(1):55–63, January 1990.

[18] K. G. Murty. *Linear Complementarity, Linear and Nonlinear Programming.* Heldermann Verlag, Berlin, 1988.

[19] Brad A. Myers, Dario A. Guise, Roger B. Dannenberg, Brad Vander Zanden, David S. Kosbie, Ed Pervin, Andrew Mickish, and Philippe Marchel. Comprehensive support for graphical, highly interactive user interfaces: The Garnet system. *Computer*, 23(11):71–85, November 1990.

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

[21] I.E. Sutherland. Sketchpad: A man-machine graphical communication system. In *Spring Joint Computer Conference*, pages 329–345, 1963.

[22] P.A. Szekely and B.A. Myers. A user interface toolkit based on graphical objects and constraints. In *Object-Oriented Programming Systems, Languages and Applications Conference*, pages 36–45, 1988.

[23] Hiroyuki Tarumi, Jun Rekimoto, Masaru Sugai, Go Yamazake, Takahiro Sugiyama, and Chuzo Akiguchi. Canae—a user interface construction environment with editors as software parts. *NEC Research and Development*, (98):89–98, July 1990.

[24] V.I. Corporation. *GECK User's Guide*, 1990.

[25] John M. Vlissides. *Generalized Graphical Object Editing.* PhD thesis, Stanford University, 1990.

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

[27] John M. Vlissides and Mark A. Linton. Unidraw: A framework for building domain-specific graphical editors. *ACM Transactions on Information Systems*, 8(3):237–268, July 1990.

[28] A. Witkin, M. Gleicher, and W. Welch. Interactive dynamics. In *ACM SIGGRAPH '90 Conference Proceedings*, 1990.