

The Programming Language Aspects of ThingLab, a Constraint-Oriented Simulation Laboratory

ALAN BORNING

Xerox Palo Alto Research Center

The programming language aspects of a graphic simulation laboratory named ThingLab are presented. The design and implementation of ThingLab are extensions to Smalltalk. In ThingLab, *constraints* are used to specify the relations that must hold among the parts of the simulation. The system is object-oriented and employs *inheritance* and *part-whole* hierarchies to describe the structure of a simulation. An interactive, graphic user interface is provided that allows the user to view and edit a simulation.

Key Words and Phrases: constraints, constraint satisfaction, object-oriented languages, inheritance, part-whole hierarchies, Smalltalk, ThingLab

CR Categories: 3.69, 4.22, 8.1, 8.2

1. INTRODUCTION

This paper describes the programming language aspects of a simulation laboratory named ThingLab. The principal research issue addressed is the representation and satisfaction of *constraints*. A constraint specifies a relation that must be maintained. For example, suppose that a user desires that the value of some integer always be displayed as a piece of text at a certain location on the screen. In a conventional language, one must remember to update the text whenever the value of the integer is changed, and to update the integer if the text is edited. In a constraint-oriented system such as ThingLab, the user can specify the relation between the text and the integer and leave it to the system to maintain that relation. If additional constraints are placed on the integer or the text, the system takes care of keeping these satisfied as well.

The notion of an *object* provides a basic organizational tool; in particular, the modularity gained by the use of object-oriented programming techniques is important for constraint satisfaction, where it is essential to know what is affected by a given change. Nonprimitive objects are constructed hierarchically from *parts*, which are themselves other objects. As is shown below, constraints provide a natural way to express the relations among parts and subparts. Methods are

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

This work was supported in part by the Xerox Corporation.

Author's present address: Department of Computer Science, FR-35, University of Washington, Seattle, WA 98195.

© 1981 ACM 0164-0925/81/1000-0353 \$00.75

also described for integrating the use of constraints with *inheritance hierarchies*, allowing new kinds of objects to be described in terms of existing ones. Finally, an interactive, graphic user interface is described that is integrated with the constraint, part-whole, and inheritance mechanisms, allowing a user to view and edit objects conveniently.

The concept of constraints, combined with inheritance and part-whole hierarchies, is one that could add significant power to programming languages. While ThingLab is not a general-purpose language, many of the concepts and techniques described here would be useful in such a context. A promising direction for future research is to explore the design of a full constraint-oriented programming language.

ThingLab is an extension to the Smalltalk-76 programming language [6, 7] and runs on a personal computer. This paper is based on the author's Stanford Ph.D. dissertation [2].

1.1 The ThingLab System

The original question addressed by the research described in this paper is as follows: "How can we design a computer-based environment for constructing interactive, graphic simulations of experiments in physics and geometry?" Examples of the sorts of things that a user should be able to simulate are simple electrical circuits and mechanical linkages. However, the underlying system should be general. Rather than a program with knowledge built into it about electrical circuit components and linkages, we envisioned a sort of kit-building kit, in which environments tailored for domains such as electrical circuit simulations or geometric figures could be constructed. There would thus be two kinds of users of the system. The first kind would employ ThingLab to construct a set of building blocks for a given domain; for example, for use in simulating electrical circuits, such a user would construct definitions of basic parts such as resistors, batteries, wires, and meters. The second kind of user could then employ these building blocks to construct and explore particular simulations.

Another requirement on the system was that it have an appropriate user interface, particularly for the second kind of user working in a particular domain. For example, to create a geometric object such as a triangle, the user should be able simply to draw it on the screen, rather than having to type in its coordinates or (worse) write some code. Similarly, making changes to an object should also be natural. To move a vertex of the triangle, the user should be able to point to it on the screen and drag it along with a pointing device, seeing it in continuous motion, rather than pointing to the destination and having the triangle jump suddenly, or (again, worse) typing in the coordinates of the destination.

As specified in the above problem description, ThingLab provides an environment for constructing dynamic models of experiments in geometry and physics, such as simulations of constrained geometric objects, simple electrical circuits, mechanical linkages, and bridges under load. However, the techniques developed in ThingLab have wider application and have also been used to model other sorts of objects, such as a graphic calculator, and documents with constraints on their layout and contents. Examples of the system in operation are presented in Section 2.

1.2 Constraints

The range of relations that can be specified in ThingLab using constraints is broad. Some examples of constraints that have been defined by various users are

- (1) that a line be horizontal;
- (2) that the height of a bar in a bar chart correspond to an entry in a table;
- (3) that one triangle be twice as big as another;
- (4) that a resistor obey Ohm's law;
- (5) that a beam in a bridge obey Hooke's law;
- (6) that the gray-scale level of an area on the computer's display correspond to a number between zero and one;
- (7) that a rectangle on the display be precisely big enough to hold a given paragraph.

The representation of constraints reflects their dual nature as both descriptions and commands. Constraints in ThingLab are represented as a *rule* and a set of *methods* that can be invoked to satisfy the constraint. The rule is used by the system to construct a procedural test for whether or not the constraint is satisfied and to construct an error expression that indicates how well the constraint is satisfied. The methods describe alternate ways of satisfying the constraint; if any one of the methods is invoked, the constraint will be satisfied.

It is up to the user to specify the constraints on an object, but it is up to the system to satisfy them. Satisfying constraints is not always trivial. A basic problem is that constraints are typically multidirectional. For example, the text-integer constraint mentioned above is allowed to change either the text or the integer. Thus, one of the tasks of the system is to choose among several possible ways of locally satisfying each constraint. One constraint may interfere with another; in general, the collection of all the constraints on an object may be incomplete, circular, or contradictory. Again, it is up to the system to sort this out.

Further, the user interface as specified in the problem description demands that constraint satisfaction be rapid. Consider the case of the user continuously moving some part of a complex geometric figure. Every time the part moves, the object's constraints may need to be satisfied again. To meet this speed requirement, constraint satisfaction techniques have been implemented that incrementally analyze constraint interactions and compile the results of this analysis into executable code. When possible, the system compiles code that satisfies the constraints in one pass. Constraint satisfaction thus takes place in two stages: there is an initial planning stage, in which a constraint satisfaction plan is formulated and compiled; then at run time this compiled code is invoked to update the object being altered.

Constraint representation is described in Section 4; constraint satisfaction is discussed in Section 5.

1.3 Object-Oriented Language Techniques

Smalltalk, in which ThingLab is written, is a language based on the idea of objects that communicate by sending and receiving messages. This object-centered factorization of knowledge provides one of the basic organizational tools.

For example, in representing a geometric construction, the objects used in the representation are things such as points, lines, and triangles. This provides a natural way of bundling together the information and procedures relevant to each object. Each object holds its own state and is also able to send and receive messages to obtain results.

Object descriptions and computational methods are organized into *classes*. Every object is an *instance* of some class. In broad terms, a class represents a generic concept, while an instance represents an individual. A class holds the similarities among a group of objects; instances hold the differences. More specifically, a class has a description of the internal storage required for each of its instances and a dictionary of messages that its instances understand, along with *methods* (i.e., procedures) for computing the appropriate responses. An instance holds the particular values that distinguish it from other instances of its class.

A new class is normally defined as a subclass of an existing class. The subclass inherits the instance storage requirements and message protocol of its superclass. It may add new information of its own and may override inherited responses to messages.

One of the important features of Smalltalk is the sharp distinction it makes between the inside and the outside of an object. The internal aspects of an object are (1) its class and (2) its instance fields and their contents; the external aspects are the messages that it understands and its responses. Since other parts of the system and the user interact with the object by sending and receiving messages, they need not know about its internal representation. This makes it easier to construct modular systems. For example, the class *Rectangle* defines the message *center*. It makes no difference to the user of this message whether a rectangle actually has a center stored as one of its instance fields or whether the center is computed on demand (in fact, it is computed on demand).

ThingLab extends Smalltalk in a number of respects. The principal extension is the inclusion of constraints and constraint satisfaction mechanisms. The other significant extensions are provision for multiple superclasses rather than just a single superclass; a part-whole hierarchy with an explicit, symbolic representation of shared substructure; the use of *paths* for symbolic references to subparts and *prototypes* for the representation of default instances; and a facility for class definition by example. The latter extensions are discussed in Section 3.

Object-oriented languages generally emphasize a very localized approach to interaction within a program: an object interacts with other parts of the system only by sending and receiving messages to other objects that it knows about. On the other hand, it is very difficult to do constraint satisfaction in a purely local way: there are problems of circularity and the like that are better spotted by a more global analysis. There is consequently a tension between the object and constraint metaphors; the integration of these approaches in ThingLab is one of its points of interest.

1.4 The User Interface

Considerable effort has been spent on designing a good user interface to the system. Some quite general graphic editing tools are provided, and purely graphic

objects, such as a triangle, can be constructed using graphic techniques only. The user interface allows objects to be viewed in other ways as well, for example, as a structural description or as a table of values.

The user interface allows smooth access to the constraint mechanism and to the inheritance and part-whole hierarchies. Thus, when the user edits an object, say by selecting a point and moving it with the cursor, the constraint satisfaction mechanism is invoked automatically to keep all the constraints satisfied. New classes may be defined by example, that is, by constructing a typical instance. The structural descriptions provided by the interface present the part-whole hierarchy, the constraints, and so forth.

1.5 Relation to Other Work

One of the principal influences on the design of ThingLab has been Sketchpad [16], a general-purpose system for drawing and editing pictures on a computer. In Sketchpad the user interacts directly with the display, using a light pen for adding, moving, and deleting parts of the drawing. ThingLab has adopted much of Sketchpad's flavor of user interaction, and the Sketchpad notions of constraints and of recursive merging have been central to its design. ThingLab has extended Sketchpad's constraint mechanism in a number of respects, most notably by integrating it with an inheritance hierarchy, by allowing local procedures for satisfying a constraint to be included as part of its definition, and by incrementally compiling the results of constraint satisfaction planning into Smalltalk code.

The other principal ancestor of ThingLab is Smalltalk. Not only is ThingLab written in Smalltalk, but the important ideas in Smalltalk—objects, classes and instances, and messages—are all used directly in ThingLab. As previously described, ThingLab adds a number of new features to the language. Smalltalk has proved to be an excellent language to support research of this sort, in terms of both linguistic constructs and programming environment.

ThingLab is also related to some very interesting work on constraint languages done at M.I.T. by Guy Steele and Gerald Sussman [13]. The ThingLab representation of an object in terms of parts and subparts, with explicit representation of shared parts, is nearly isomorphic to the representation independently developed by Steele and Sussman. Their system has a built-in set of primitive constraints, such as adders and multipliers, from which compound constraints can be constructed. This is similar to the method used in the ThingLab calculator example described in Section 2.2. To handle constraints that cannot be satisfied using a one-pass ordering, they employ multiple redundant views that can cooperate in solving the problem; in their previous work, symbolic algebraic manipulation techniques were employed. Their use of multiple views has been adopted in ThingLab. Among the differences between the two systems is that Steele and Sussman's language retains dependency information, that is, a record of the justifications for each conclusion, for producing explanations and for implementing efficient backtracing when search is needed (dependency-directed backtracking). On the other hand, their system has no graphics capabilities. Also, ThingLab has two significant advantages in regard to efficiency. First, it compiles plans into the base language, whereas in Steele and Sussman's system constraint satisfaction is done interpretively. Compilation is essential if constraint languages are to

become practical tools. Second, ThingLab has a class-instance mechanism, including multiple inheritance, that allows information common to several objects to be factored out, while their system uses a macro facility for abstraction, which has the disadvantage that a complete copy of the constraint network is required for each instance.

Steele's recent Ph.D. dissertation [12], completed after the work described above and the author's own dissertation, gives a clear statement of design goals for a complete, general-purpose language organized around constraints and describes further progress toward implementing such a language. The system deals explicitly with the problem of behaving properly in the presence of contradictions, which is important for the interactive construction of large systems, and further develops the notions of assumptions and defaults. While its usual mode of operation is interpretive, it also includes a constraint compiler like that used in ThingLab.

Other related work on languages includes SIMULA [3], which is one of the principal ancestors of Smalltalk. The distinction that Smalltalk makes between the inside and the outside of an object is also closely related to the data-abstraction mechanisms in languages such as MESA [10], CLU [9], and ALPHARD [17]. These languages separate the interface specification of a type from its internal implementation, just as Smalltalk distinguishes the external message protocol of an object from its internal aspects. Thus, in programs in these data-abstraction languages, changes to the implementation of a type (but not its interface) do not affect the users of that type; so more modular systems result.

ABSET [4] is a set-oriented language developed at the University of Aberdeen with a number of constraint-like features; for example, given the statement $A + B = 3$ AND $A = 1$, it can deduce B 's value. Also, it emphasizes the avoidance of unnecessary ordering restrictions in the statement of a program. The ACTOR languages [5] use and extend the notion of objects that communicate by passing messages. Representation languages for artificial intelligence work, such as KRL [1], develop the notion of multiple inheritance. ThingLab's facility for class definition by example is related to work on programming by example [8, 11].

There is a large body of work in artificial intelligence on reasoning and problem-solving systems of various kinds. Most of these systems are concerned with more complex problem-solving tasks than those tackled in ThingLab. By contrast, in ThingLab much of the emphasis has been on finding ways of generalizing plans and compiling them as procedures so that they may be used efficiently in a graphic environment. However, the problem-solving techniques developed in these other systems may well prove useful if ThingLab's constraint satisfaction abilities are to be strengthened.

This artificial intelligence work includes a number of systems that use constraints and constraint satisfaction as such. Steels [14] has constructed a reasoning system, modeled on a society of communicating experts, that uses propagation of constraints in its reasoning process. Unlike either ThingLab or Steele and Sussman's system, Steels' system is description-oriented and does not require that constraint satisfaction yield a unique value. Stefik [15] uses the technique of constraint posting in MOLGEN, a system for planning experiments in molecular genetics. His system uses hierarchical planning and dynamically formulates and propagates constraints during its planning process.

2. SOME EXAMPLES

Before plunging into a technical discussion of the system, it is useful to present some examples of its operation. A brief description of the operation of the ThingLab user interface is needed first. The user interacts with ThingLab via a *window*, a rectangular area on the computer's display. The window notion is central to Smalltalk's user interface philosophy. The ThingLab window described here is typically one of several windows on the screen, with other windows being available for debugging, editing system code, freehand sketching, and so on.

The ThingLab window is divided into five panes: the *class pane*, the *format pane*, the *messages pane*, the *arguments pane*, and the *picture pane*. The class pane is a *menu* of names of classes that may be viewed and edited. Once a class has been selected, a menu of formats in which it can display itself appears in the *format pane* immediately to the right. The class shows itself in the chosen format in the large *picture pane* at the bottom of the window.

The two remaining panes, messages and arguments, contain menus used for graphic editing of the class' prototype. All editing operations are performed by sending a message to the object being edited; the ThingLab window allows us to compose and send certain kinds of editing messages graphically. The messages pane contains a list of message names, such as *insert* and *delete*, while the arguments pane contains a list of possible classes for the message argument. The argument itself will be an instance of that class, either newly created or selected from among the parts in the picture.

The user communicates with the system primarily by means of a *mouse* and secondarily by use of a keyboard. The mouse is a small box-shaped object that can be moved about on the user's desk top; as it moves, its relative position is tracked by a cursor on the screen (the arrow in the illustrations). If some graphic object on the screen is "attached" to the cursor, that object moves as well. The mouse also has three buttons on it, which serve as control keys.

In the menu panes, a black stripe indicates a selected item. Thus, in Figure 1, *Triangle* and *prototype's picture* have been selected. Since a menu may be too long to fit in its pane, all the menus can be *scrolled* up or down so that the user can view and select any of the items. To make a selection, the user positions the cursor over the item to be selected and pushes a button on the mouse.

2.1 A Geometric Example

As an introductory example, we use ThingLab to construct a quadrilateral and to view it in several ways. We then use the system to demonstrate a theorem about quadrilaterals.

2.1.1 Defining the Class of Quadrilaterals. First, we define the class of quadrilaterals. New classes are always defined as a subclass of some more general class; if nothing better is available, they can be made subclasses of class *Object*, the most general class in the system. In this case, we create the new class *Quadrilateral* as a subclass of *GeometricObject*.

One of the important features of the ThingLab environment is that the user can define classes by example. To be more precise, the structural aspects of a class (its part descriptions and constraints) may be specified incrementally by editing its prototypical instance. We define the class *Quadrilateral* in this way.

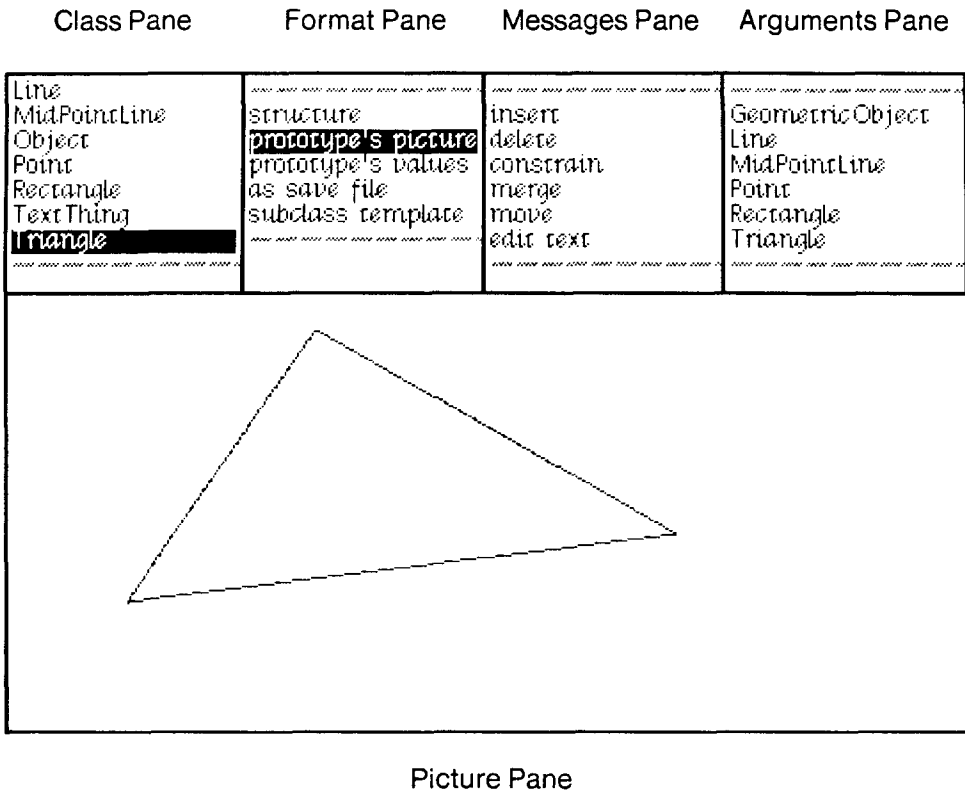


Fig. 1. Panes of the ThingLab window.

First, we ask to view the picture of the prototype Quadrilateral. So far, the prototype has no parts, and so its picture is blank. We now edit the prototype by adding and connecting four sides. Using the mouse, we select the word *insert* in the messages pane and the word *Line* in the arguments pane. When we move the cursor into the bottom pane, a blinking picture of a line appears, attached to the cursor by one of its endpoints. As the cursor is moved, the entire line follows. When the endpoint attached to the cursor is in the desired location, we press a button. This first endpoint stops moving, and the cursor jumps to the second endpoint. The second endpoint follows the cursor, but this time the first endpoint remains stationary. We press the button again to position the second endpoint (Figure 2).

We insert another line in the same way. To connect the new line to the first, we position the endpoint attached to the cursor near one of the endpoints of the first line. When the two points are close together, the moving point locks onto the stationary point, and the line stops blinking. This indicates that the two points will merge if the button is pressed. We press the button and the points merge. The two lines now share a common endpoint. Also, a record of the merge is kept by the class Quadrilateral. Similarly, we position the other endpoint and insert the remaining two lines (Figure 3).

During this editing session, the system has been updating the structure common to all quadrilaterals that is stored in the class Quadrilateral, as well as saving the

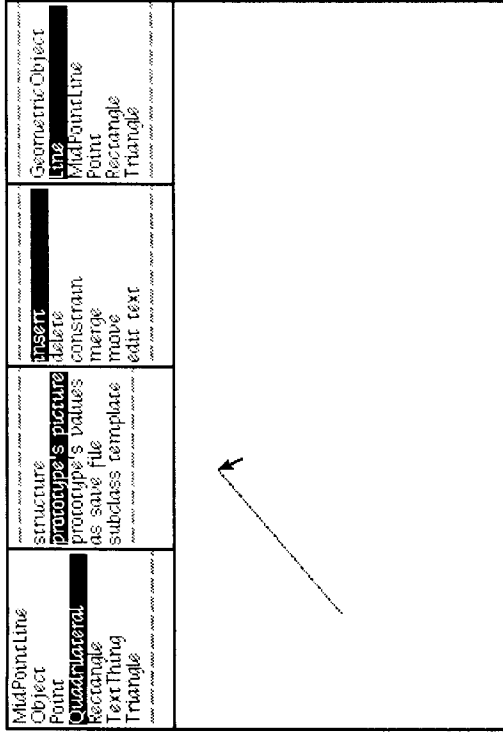


Fig. 2. Positioning the second endpoint of a line.

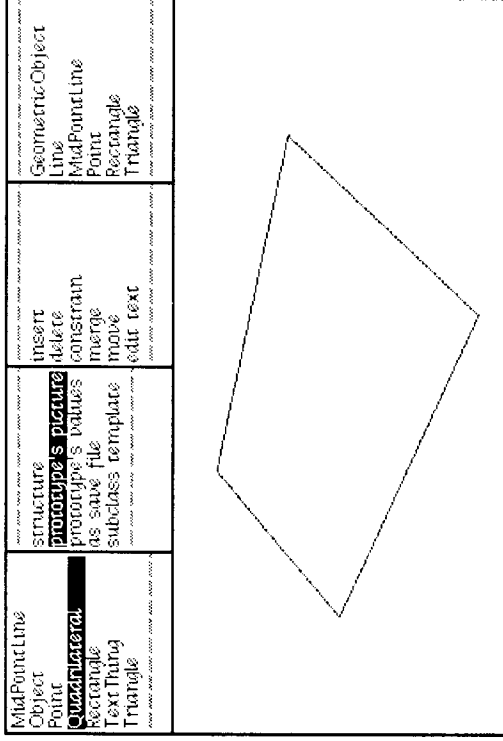


Fig. 3. The completed quadrilateral.

particular locations of the prototype's sides. To see the structure of the class *Quadrilateral*, we select *structure* in the menu of formats. The class responds by listing its name, superclasses, part descriptions, and constraints (Figure 4). We may also view the values stored in the prototype by selecting *prototype's values* (Figure 5).

2.1.2 Demonstrating a Geometry Theorem. We may now use the new class in demonstrating a geometry theorem. The theorem states that, given an arbitrary quadrilateral, if one bisects each of the sides and draws lines between the adjacent midpoints, the new lines form a parallelogram.

To perform the construction, we make a new class named *QTheorem*. As before, we create it as a subclass of *GeometricObject* and define it by example. We first add an instance of class *Quadrilateral* as a part. We select *insert* and *Quadrilateral*. As we move the cursor into the bottom pane, a blinking picture of a quadrilateral, whose shape has been copied from the prototype, appears. We position the quadrilateral and press a button.

The next step is to add midpoints to the sides of the quadrilateral. To do this, we use four instances of the class *MidPointLine*. This class specifies that each of its instances has two parts: a line and a point. In addition, it has a constraint that, for each instance, the point be halfway between the endpoints of the line. As we insert each instance of *MidPointLine*, we move it near the center of one of the sides of the quadrilateral and merge the line part of the *MidPointLine* with the side of the quadrilateral (Figure 6). The last step is to add four lines connecting the midpoints to form the parallelogram.

Once the construction is complete, we may move any of the parts of the prototype *QTheorem* and observe the results. In general, it is not enough for the system simply to move the selected part; because of the constraints we have placed on the object, other parts, such as the midpoints, may need to be moved as well to keep all the constraints satisfied. Suppose we want to move a vertex. We select the message *move* and the argument *Point*. A blinking point appears in the picture that is attached to the cursor. We position it over the vertex to be moved and hold down a button. The vertex follows the cursor until the button is released (Figure 7). (The first time we try to move the vertex, there will be a long pause as the system plans how to satisfy the constraints.) We notice that indeed the lines connecting the midpoints form a parallelogram no matter how the quadrilateral is deformed. The theorem remains true even when the quadrilateral is turned inside out!

2.1.3 Constraint Satisfaction. The user described how *QTheorem* should behave in terms of the midpoint constraint and the various merges, but not by writing separate methods for moving each part of *QTheorem*. The midpoint constraint (as defined by an experienced user) describes methods that can be invoked to satisfy itself. Three such methods were specified: the first asks the midpoint to move to halfway between the line's endpoints; the second asks one of the line's endpoints to move; and the third asks the other endpoint to move. It was up to *QTheorem* to decide which of these methods to invoke, and when and in what order to use them.

In general, the constraints on an object might specify its behavior incompletely or redundantly, or they might be unsatisfiable. *QTheorem*, for example, is

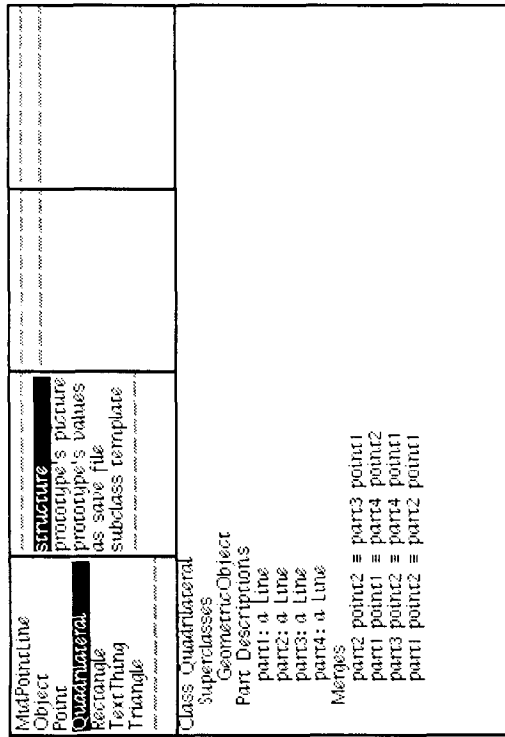


Fig. 4. Structure described by the class Quadrilateral.

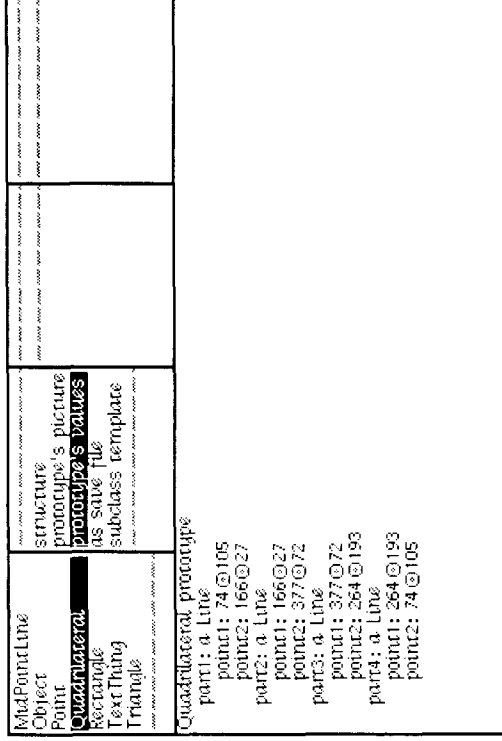


Fig. 5. Values of the prototype Quadrilateral.

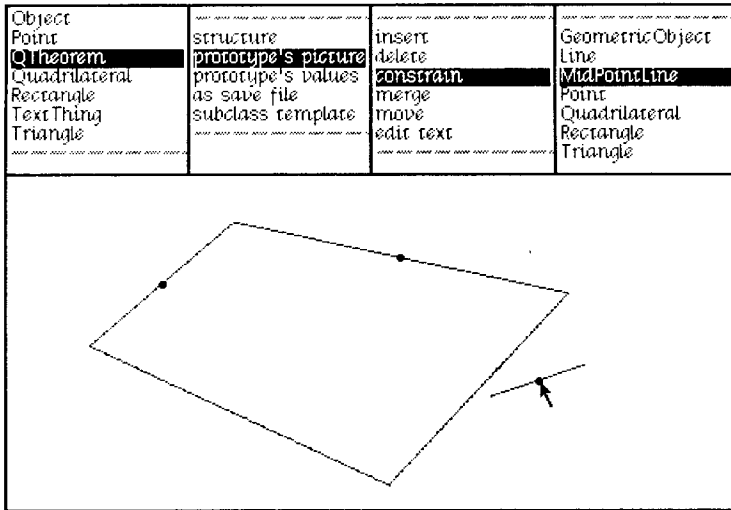


Fig. 6. Adding a midpoint.

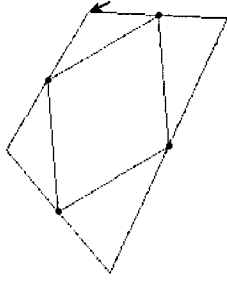
underconstrained. The behavior we observed was only one way of moving the vertex while satisfying the constraints. Two other possibilities would have been for the entire object to move, or for the midpoints to remain fixed while the other vertices moved. Neither of these responses would have been as pleasing to us as human observers. (If we had wanted the entire object to move, we would have specified *move QTheorem* instead.) Therefore, besides the more mathematical techniques for finding *some* way of satisfying its constraints, or for deciding that they are unsatisfiable, an object can also take the user's preferences into account in deciding its behavior. In this case, the midpoint constraint specified that the midpoint was to be moved in preference to one of the endpoints of the line.

We might override the preference specified in the midpoint constraint by anchoring the midpoints, as in Figure 8. (Anchor is a subclass of Point, with an added constraint that its instances may not be moved during constraint satisfaction.)

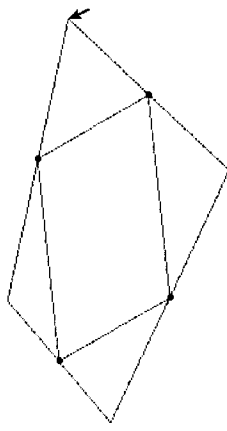
2.2 Constructing a Program for a Graphic Calculator

In this second example, we construct some graphic programs for a simulated calculator. In the process, we use a number of classes from a "calculator kit." One simple but important class is *NumberNode*. An instance of *NumberNode* has two parts: a real number and a point. Its purpose is to provide a graphic representation of a register in the calculator. Another class is *NumberLead*, consisting of a number node and an attached line. As with leads on electrical components, it is used to connect parts of the calculator. Also, classes that represent the various arithmetic operations have been defined. There is a general class *NumberOperator*, whose parts are a frame containing the operator's symbol and three number leads that terminate on the edges of the frame. Four subclasses of *NumberOperator* are defined, namely, *Plus*, *Minus*, *Times*, and *Divide*. *Plus*, for example, has three number leads with number nodes at the ends, which are inherited from *NumberOperator* (Figure 9). It has an added constraint that the

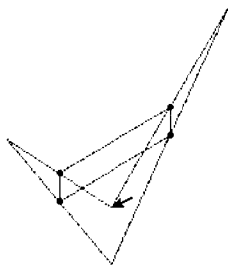
Object Point Quadrilateral Rectangle TextThing Triangle	structure prototype's picture prototype's values as same file subclass template	insert delete constraint merge undo edit text	GeometricObject Line MidPointLine Point Quadrilateral Rectangle Triangle
------------------------------------------------------------------------	---------------------------------------------------------------------------------------------	--------------------------------------------------------------	--------------------------------------------------------------------------------------------



Object Point Quadrilateral Rectangle TextThing Triangle	structure prototype's picture prototype's values as same file subclass template	insert delete constraint merge undo edit text	GeometricObject Line MidPointLine Point Quadrilateral Rectangle Triangle
------------------------------------------------------------------------	---------------------------------------------------------------------------------------------	--------------------------------------------------------------	--------------------------------------------------------------------------------------------



Object Point Quadrilateral Rectangle TextThing Triangle	structure prototype's picture prototype's values as same file subclass template	insert delete constraint merge undo edit text	GeometricObject Line MidPointLine Point Quadrilateral Rectangle Triangle
------------------------------------------------------------------------	---------------------------------------------------------------------------------------------	--------------------------------------------------------------	--------------------------------------------------------------------------------------------



Object Point Quadrilateral Rectangle TextThing Triangle	structure prototype's picture prototype's values as same file subclass template	insert delete constraint merge undo edit text	GeometricObject Line MidPointLine Point Quadrilateral Rectangle Triangle
------------------------------------------------------------------------	---------------------------------------------------------------------------------------------	--------------------------------------------------------------	--------------------------------------------------------------------------------------------

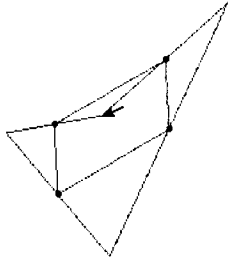


Fig. 7. Moving a vertex of the quadrilateral.

Anchor GeometricObject Line MidPointLine Object Point	structure prototype's picture prototype's values as save file subclass template	insert delete constrain merge move edit text	Anchor GeometricObject Line MidPointLine Point Quadrilateral Rectangle
----------------------------------------------------------------------	---------------------------------------------------------------------------------------------	-------------------------------------------------------------	------------------------------------------------------------------------------------------

Anchor GeometricObject Line MidPointLine Point Quadrilateral Rectangle	insert delete constrain merge move edit text	structure prototype's picture prototype's values as save file subclass template	Anchor GeometricObject Line MidPointLine Object Point
------------------------------------------------------------------------------------------	-------------------------------------------------------------	---------------------------------------------------------------------------------------------	----------------------------------------------------------------------

Fig. 8. A quadrilateral with anchored midpoints.

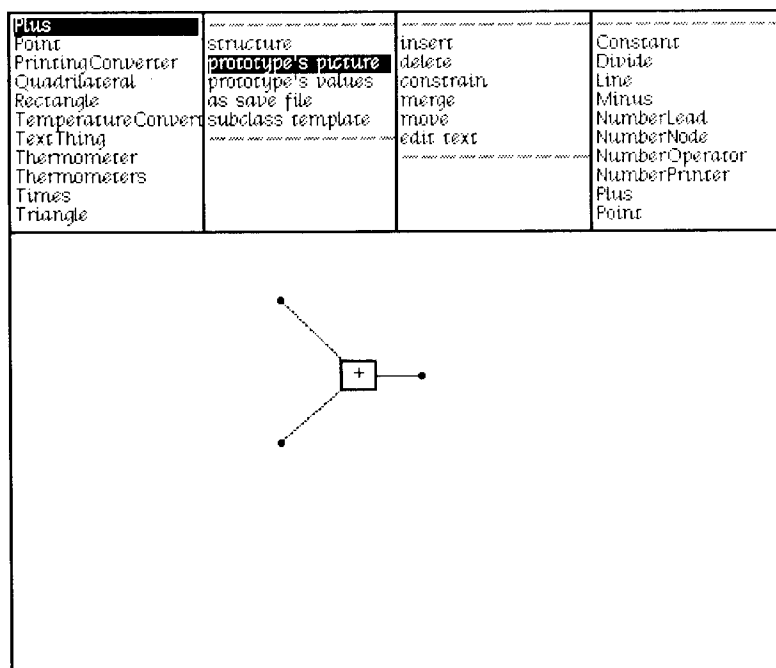


Fig. 9. Picture of the prototype for Plus.

number at the node on the right always be the sum of the numbers at the leads on the left. The classes for Minus, Times, and Divide prototypes have been defined analogously.

To view and edit a number at a node, the class `NumberPrinter` has been constructed. Its parts are a number lead and an editable piece of text. Also, it has a constraint that the number at its node correspond to that displayed in the text. If the node's number changes, the text is updated; if the text is edited, the node's number is changed correspondingly. A special kind of `NumberPrinter` is a `Constant`. For constants, the constraint is unidirectional. The text may be edited, thus changing the number; but the number may not be changed to alter the text.

2.2.1 Constructing a Celsius-to-Fahrenheit Converter. Using these parts, let us construct a Celsius-to-Fahrenheit converter. After creating a new class, `TemperatureConverter`, we select *insert* and *Times*. As we move the cursor into the picture pane, a blinking picture of an instance of the class `Times` appears. We position the frame that holds the multiplication symbol, and then the three nodes. Next, we insert a Plus operator in the same manner, connecting its addend node to the product node of the times operator. (The connection is made by merging the nodes, in the same way that the endpoints of the sides of the quadrilateral were connected.) Finally, we insert two instances of `Constant`, connecting them to the appropriate nodes of the operators. We then invoke the *edit text* message and change the constants to 1.8 and 32.0. The result is shown in Figure 10.

Once the converter has been defined, we may use it as a part of other objects (i.e., as a subroutine). As an example, we define a new class `PrintingConverter`. We add an instance of `TemperatureConverter` as a part, and also two instances

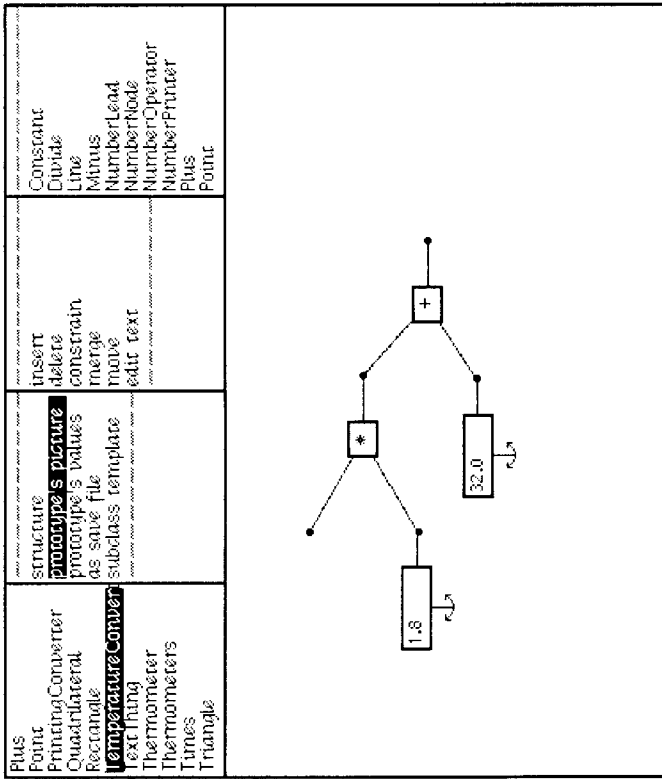


Fig. 10. Picture of the completed Celsius-to-Fahrenheit Converter.

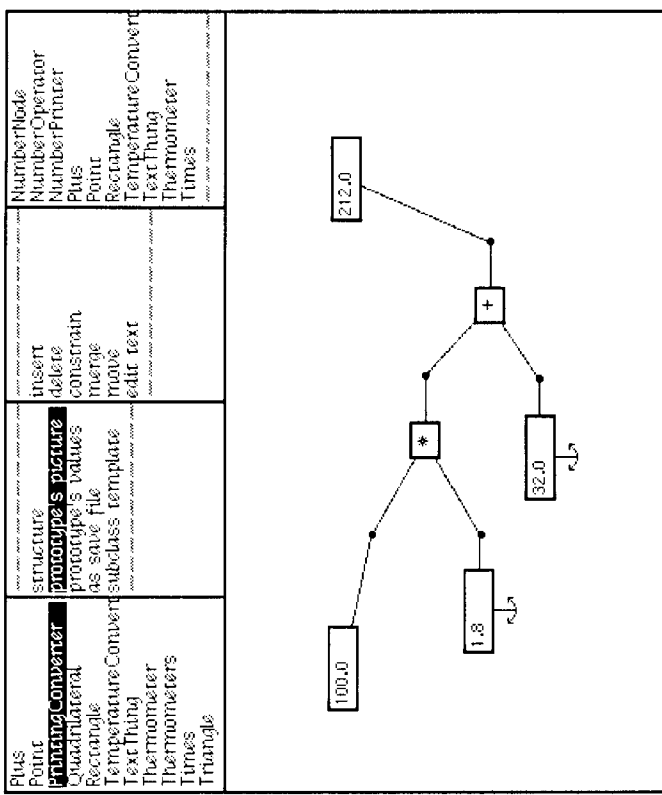


Fig. 11. A PrintingConverter.

of NumberPrinter to display the Celsius and Fahrenheit temperatures (Figure 11). If we edit the Celsius temperature, the PrintingConverter satisfies its constraints by updating the numbers at its nodes and the Fahrenheit temperature displayed in the frame on the right.

However, because of the multiway nature of the constraints, the device works backward as well as forward! Thus, we can edit the Fahrenheit temperature, and the Celsius temperature is updated correspondingly (Figure 12). This demonstrates the need for the special class Constant: without it, the system could equally well have satisfied the constraints by changing one of these coefficients rather than the temperatures.

We may also connect the converter to other types of input/output devices, for example, a simulated thermometer. We can select *move* and *Point* and grab either of the columns of mercury with the cursor. When we move one of the columns up or down, the other column moves correspondingly (Figure 13).

2.2.2 Solving a Quadratic Equation. After experimenting with the converter, we might try building a more complex device, such as the network for solving quadratic equations shown in Figure 14.

When we edit any of the constants, the value in the frame on the left changes to satisfy the equation. In the picture, the coefficients of the equation $x^2 - 6x + 9 = 0$ have been entered, and a solution, $x = 3$, has been found. This case is unlike the temperature converter examples: the system was unable to find a one-pass ordering for solving the constraints and has resorted to the relaxation method. Relaxation will converge to one of the two roots of the equation, depending on the initial value of x .

Now let us try changing the constant term c from 9 to 10. This time, the system puts up an error message, protesting that the constraints cannot be satisfied. Some simple algebra reveals that the roots of this new equation are complex; but the number nodes hold real numbers, and so the system was unable to satisfy the constraints.

A better way of finding the roots of a quadratic equation is to use the standard solution to the quadratic equation $ax^2 + bx + c = 0$, namely, $x = (-b \pm (b^2 - 4ac)^{1/2}) / 2a$. The system can be told about this canned formula by defining a class QuadraticSolver whose parts include four NumberNodes a , b , c , and x and a constraint that $x = (-b + (b^2 - 4ac)^{1/2}) / 2a$. (Since the class NumberNode does not allow multiple values, in the QuadraticSolver's constraint one of the roots has been chosen arbitrarily as the value for x . A more general solution would be to define a class MultipleRoots and set up the constraint so that it determined both the number of roots and their values.)

We can insert an instance of QuadraticSolver into the network, merging its number nodes with the appropriate existing nodes in the network (Figure 15). Now, the system can find a simple one-pass ordering for satisfying the constraints and does not need to use relaxation.

In inserting an instance of QuadraticSolver into the network, we have added another view of the constraints on x . In the sense that the permissible values of x are the same with or without it (ignoring the multiple-root problem), the new constraint adds no new information. However, QuadraticSolver's constraint is computationally better suited to finding the value of x . This technique of

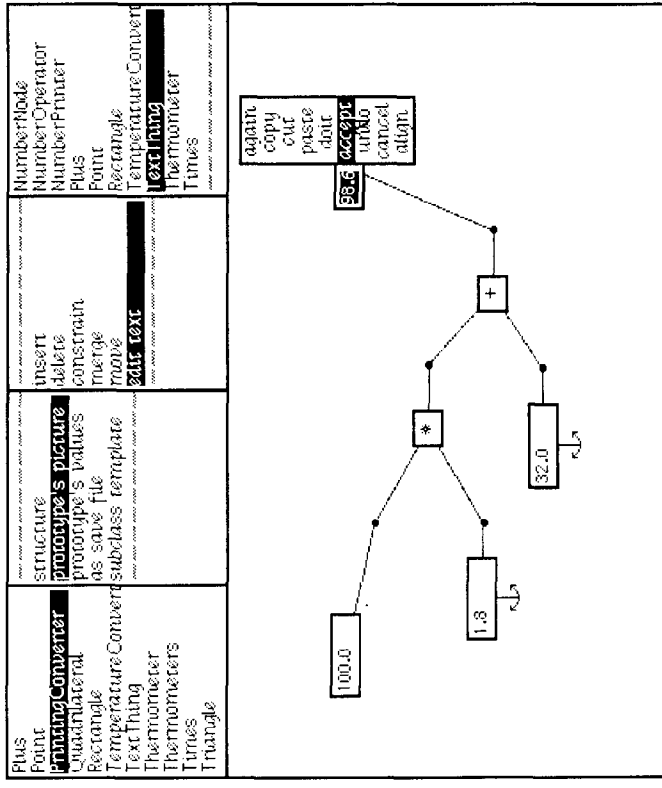
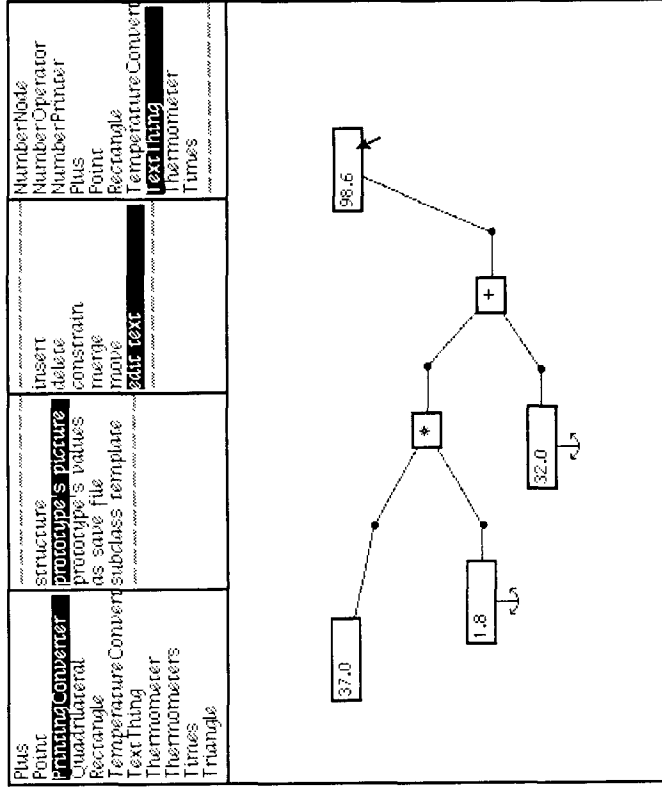


Fig. 12. Editing the Fahrenheit temperature.

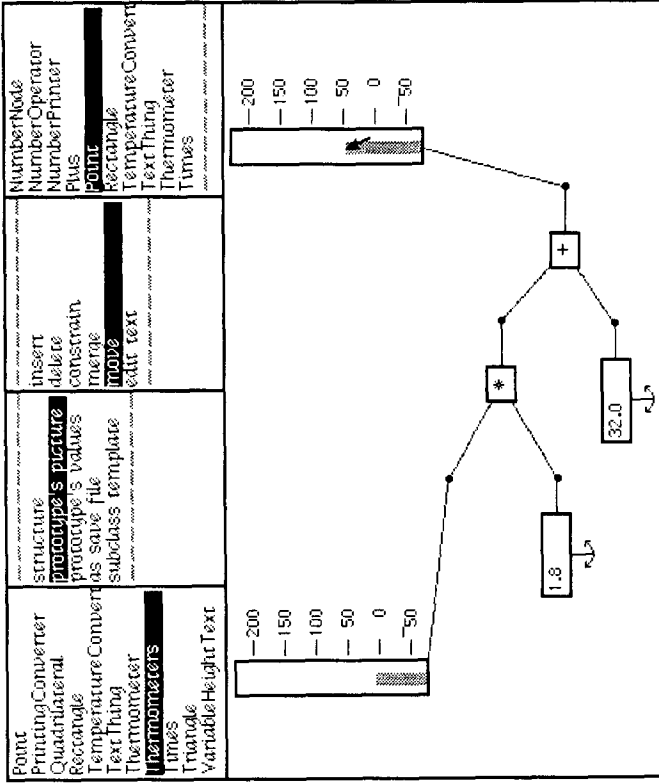


Fig. 13. The temperature converter with thermometers for input and output.

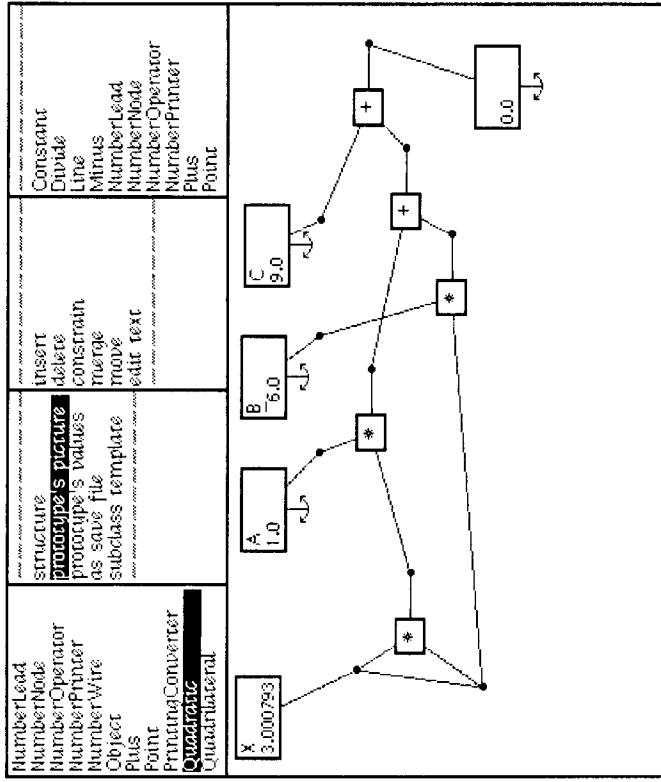


Fig. 14. A quadratic equation network.

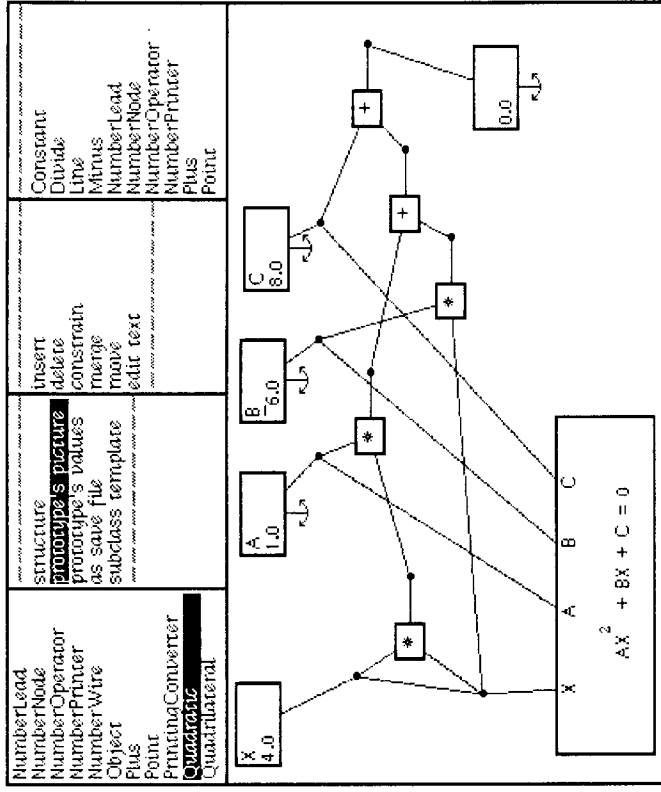


Fig. 15. The network after adding of QuadraticSolver.

introducing multiple redundant constraints on an object is an important way of dealing with circularity.

3. OBJECTS

3.1 The Part–Whole Relationship

In ThingLab, an object is composed of named parts, each of which is in turn another object. The parts are thus composed of subparts, and so on. The recursion stops with primitive objects such as integers and strings. Consider a line:

Line

point1: a Point

x: 50

y: 100

point2: a Point

x: 200

y: 200.

The line is composed of two parts that are its endpoints. Each endpoint is in turn composed of an *x* and a *y* value; these are primitive objects (integers). An object is sometimes referred to as the *owner* of its parts. For example, the above line owns its endpoints.

3.1.1 Part Descriptions. A *PartDescription* is an object that describes the common properties of the corresponding parts of all instances of a class. Every class has a list of part descriptions, one for each part owned by its instances. The following things are associated with each part description:

<i>name</i>	an identifier;
<i>constraints</i>	the set of constraints that apply to the corresponding part of each instance;
<i>merges</i>	the set of merges that apply to the corresponding part of each instance;
<i>class</i>	the class of the corresponding part of each instance. This is more restrictive than in Smalltalk, where the class of the contents of an instance field is not declared. Imposing this restriction makes the job of constraint satisfaction easier.

When a part description is added to a class, messages are compiled automatically in the class' message dictionary to read and write the part.

For example, the class *Line* has two part descriptions that describe the parts of each instance of *Line*. The first part description has the name *point1*. It has no constraints or merges, and it specifies that the *point1* part of each line be an instance of class *Point*. The other part description is defined analogously. For a class that specifies some constraints, for example, the class *HorizontalLine*, the *point1* part description would also indicate that there was a constraint that applied to the *point1* part of each of its instances.

3.1.2 Insides and Outsides. As described in Section 1.3, one of the important features of Smalltalk is the sharp distinction it makes between the inside and the outside of an object. In ThingLab, the notion of having a part has implications for both the internal and external aspects of the object that owns the part.

Internally, the object must have an instance field in which the part is stored, as well as a corresponding part description in its class; externally, the object should understand messages to read and write the part. However, these internal and external aspects are separate. A *virtual part*, as proposed in [2], is an example of the use of this separation. Such a part would have all the external manifestations of a part, that is, messages to read and write it. Internally, however, there would be no corresponding field; rather, the part would be computed as needed. (Smalltalk already has virtual parts; the proposed mechanism would add the necessary declarative superstructure so that the constraint satisfaction mechanism could know about them.)

3.1.3 Paths. A *path* is a ThingLab object that represents a symbolic reference to a subpart. Each path is a hierarchical name, consisting of a list of part names that indicates a way to get from some object to one of its subparts. The path itself does not own a pointer to the object to which it is applied; this must be supplied by the user of the path. Thus the same path can be used to refer to the corresponding subpart of many different objects. For example, *point1 x* is a path to get to the *x* value of the first endpoint of any line. Typically, the path as such is used only during compilation; this path would compile code that sent the message *point1* to a line and then sent the message *x* to the result.

While the definition of a path is simple, the idea behind it has proved quite powerful and has been essential in allowing the constraint- and object-oriented metaphors to be integrated. As mentioned above, Smalltalk draws a distinction between the inside and the outside of an object. The notion of a path helps strengthen this distinction by providing a protected way for an object to provide external references to its parts and subparts. For example, if a triangle wishes to allow another object to refer to one of its vertices, it does so by handing back a path such as *side2 point1*, rather than by providing a direct pointer to the vertex. If this other object wants to change the location of the vertex, it must do so by routing the request through the triangle, rather than by simply making the change itself. This allows the triangle to decide whether or not to accept the change; if it does accept it, it knows what has been altered, so that it can update its other parts as necessary to satisfy all its constraints.

In addition to these semantic considerations, a major pragmatic benefit of this discipline is that no backpointers are needed. (If the triangle did hand out a direct pointer to its vertex, the vertex would need a pointer back to the triangle so that it could inform the triangle when it changed.) Access to parts is somewhat slower using this technique, since each access involves following a path. However, an access via a path can often be moved out of the inner loops by the constraint compiler. Another pragmatic consideration is that constraints and merges can be represented symbolically using paths, so that they apply to all instances of a class, rather than to a particular instance. This allows the system to compile constraint satisfaction plans in the form of standard Smalltalk methods.

ThingLab's constraint satisfaction techniques all depend on noticing when one constraint applies to the same subpart as another. Paths are used to specify which parts or subparts of an object are affected by the constraint. Two paths *overlap* if one can be produced from the other by adding zero or more names to

the end of the other's list. The following paths overlap the path *side1 point1*:

side1 point1 x
side1 point1
side1
 (*the empty path*)

The following paths do not overlap *side1 point1*:

side1 point2
side2

To test if two constraints apply to the same subpart, the system checks to see if any of their paths overlap.

3.2 Inheritance

A new class may be defined as a subclass of one or more existing classes. The subclass inherits the part descriptions, constraints, merges, and message protocol of its superclasses. It may add new information of its own, and it may override inherited responses to messages. Every class (except class Object) must be a subclass of at least one other class.

The superclasses of an object are represented by including an instance of each superclass as a part of the object. The field descriptions for such parts are instances of SuperclassDescription, a subclass of PartDescription. These parts may have constraints and merges applied to them in the usual way; among other things, this allows the user to indicate that parts inherited from several superclasses are in fact to be represented by only a single part in the subclass. The only difference between these instances of superclasses and ordinary parts is that messages are forwarded to them automatically. (The actual implementation is somewhat more arcane, to take advantage of the efficient single-superclass mechanism built into Smalltalk. However, the effect is as described, and the reader should think of it in this way.)

3.2.1 Class Object. The most general class in both Smalltalk and ThingLab is class Object. As part of the ThingLab kernel, a large number of methods have been added to this class. These methods provide defaults for adding or deleting parts, merging parts, satisfying constraints, showing in a ThingLab window, and so on. In general, these methods treat an object as the sum of its parts. For example, to show itself, an object asks each of its parts to show; to move itself by some increment, the object asks each of its parts to move by that increment. This strict hierarchy is, however, modified by the object's constraints and merges. Thus, when an object decides exactly how to move, it must watch for overlap between its parts due to merges, and it must also keep all its constraints satisfied.

3.2.2 Message Behavior. When an object receives a message, the object's class first checks its own message dictionary. If a corresponding method is found, that method is used. If not, the class asks each of its superclasses if any of them has an appropriate method. In turn, each superclass, if it does not itself define the method, will ask *its* superclasses, and so forth, thus implementing inheritance through multiple levels of the hierarchy. If there is a single inherited method for that message, then that method is used. If there is no method, or if there are

several conflicting inherited methods, an error occurs. Note that the overriding of inherited methods is still allowed; it is an error only if a class with no method of its own inherits different methods via two or more of its immediate superclasses. If the user wants to choose among conflicting messages, or to combine them somehow, an appropriate method for doing this should be defined in the subclass. To avoid this search the next time the message is received, the class automatically compiles a *message forwarder* that will intercept that message in the future and relay it directly to the appropriate superclass part.

As an example of the use of multiple superclasses, suppose that a user has available a class of horizontal lines and another class of lines of constant length. The class of horizontal lines of constant length may then be defined as a subclass of both of these.

Multiple superclasses also provide a way of implementing multiple representations of objects. For example, suppose the user desires to represent a point in both Cartesian and polar forms. This may be done as follows:

Class CartesianPoint

Superclasses

GeometricObject

Part Descriptions

x: a Real

y: a Real

Class PolarPoint

Superclasses

GeometricObject

Part Descriptions

r: a Real

theta: a Real

Class MultiplyRepresentedPoint

Superclasses

C: CartesianPoint

P: PolarPoint

Constraints

C = P asCartesian

C ← P asCartesian

P ← C asPolar

The constraint on *MultiplyRepresentedPoint* keeps the parts representing the two superclasses in coordination. It makes use of an auxiliary message to *PolarPoint* that returns its Cartesian equivalent, and of an analogous message to *CartesianPoint*.

3.2.3 Prototypes. For a given class, a prototype is a distinguished instance that owns default or typical parts. All classes understand the message *prototype* and respond by returning their prototypical instance. If the user does not specify otherwise, the prototype has nil in each of its instance fields. However, if the user has defined the class by example, the prototype holds the particular values from the example. These values may also be set by writing an initialization message.

Prototypes provide a convenient mechanism for specifying default instance values. Thus, in the introductory example, when a new line was being inserted into the quadrilateral, its initial length and orientation were copied from the prototype *Line*. Such defaults are essential in graphic editing, since every object needs *some* appearance.

More important, a prototype serves as a representative of its class. ThingLab distinguishes between messages that have no side effects for the receiver (read-only messages), messages that alter the values stored in the receiver, and messages that alter the receiver's structure. Any instance accepts read-only or value-altering messages, but only prototypes accept structure-altering messages. The reason is that this latter type of message affects the class. The prototype is in charge of its class and is willing to alter it, but, for instances other than the prototypical one, the class is read-only. Requests to move a side of a polygon, or even turn it inside out, are examples of value-altering messages. On the other hand, requests to add or delete a side, edit a constraint, or merge two points are structure-altering messages.

3.2.4 Defining Classes by Example. When the user defines a class by example, the editing messages are always sent to the prototype, rather than sometimes to the class and sometimes to one of its instances. The prototype takes care of separating the generic information that applies to all instances of its class from the specific information that applies only to the default values that it holds in its fields. With its class it associates the number and class of the parts, the constraints, and the merges. With its own instance fields it associates the default values for its parts.

It is not possible to define all classes by example; some, such as classes for new constraint types and abstract classes like *GeometricObject*, must be entered by writing an appropriate Smalltalk class definition. In general, there are many possible classes that could be abstracted from a given example; which one *should* be abstracted depends on the user's purposes. The ThingLab facility for definition by example provides a reasonable default, but it is not a general solution to this problem. If the user wants some other sort of class, he or she should write an appropriate definition.

4. CONSTRAINT REPRESENTATION

This section describes the representation of ThingLab constraints. To support constraints, some new kinds of objects were implemented. In Smalltalk, objects communicate by sending and receiving messages; an object's response to a message is implemented by a method (i.e., a procedure). ThingLab objects are described that stand for Smalltalk messages and methods. The purpose of this additional mechanism is to provide tools for reasoning about messages and methods, and in particular about the interactions among messages and constraints.

4.1 Message Plans

A message plan is an abstraction of the Smalltalk notion of sending a message. A message plan does not stand for a particular act of sending a message; rather, it is a template for any number of messages that might be sent. A message plan is itself an object: an instance of class *MessagePlan*. The parts of a message plan include a *receiver*, a *path*, an *action*, and zero or more *arguments*. The receiver is normally a particular object, although for some uses it may be nil or may be a prototype representing any instance of a class of objects that might receive the message. The path tells how to get to one of the receiver's subparts, which will be

called the *target* of the message plan. The action is a *selector* for a Smalltalk method understood by the target. The arguments may be either actual or symbolic. Actual arguments are pointers to other objects; symbolic arguments are simply names (strings). The arguments correspond to the arguments passed at run time to the Smalltalk method invoked by the action. For example, here is a message plan asking a triangle to move one of its vertices right by ten screen dots:

```
triangle side1 point2 moveby: 10@0.
```

The receiver is *triangle*, the path is *side1 point2*, the action is *moveby:*, and the argument is the point *10@0*.

An important use of message plans is to describe the methods for satisfying a constraint. If a message plan is used in this way, the plan will have several Boolean flags and a pointer to the constraint that generated it, in addition to the parts listed above. The flags are the following:

<i>uniqueState</i>	true if there is only one state of the target that will satisfy the constraint (given that all other parts of the receiver are fixed). See Section 4.3.2 below;
<i>referenceOnly</i>	true if the action described by the message plan only references its target, rather than altering it;
<i>compileTimeOnly</i>	true if the message plan is used only during constraint satisfaction planning and not in producing executable code.

4.2 Methods

In ThingLab, an explicit class *Method* has been defined. The parts of a method are a list of *keywords*, a matching list of symbolic *arguments*, a list of *temporaries*, and a procedural *body*. The selector for the method is constructed by concatenating the keywords. These parts are the same as those of a Smalltalk method, the only difference being that in Smalltalk the method is stored as text, and the parts must be found by parsing the text. One reason for defining an explicit class in ThingLab is to simplify access to the parts of a method. This is useful because methods are often generated by the system rather than being entered by the user, with different parts of the method coming from different parts of the system. Also, some methods have their own special properties. For example, all the methods that an object has for showing itself are indexed in a table used by the ThingLab user interface.

After a ThingLab method has been constructed, it is usually asked to add itself to some class' method dictionary. In the implementation, the method does this by constructing a piece of text and handing it to the regular Smalltalk compiler. The Smalltalk compiler in turn produces a byte-coded string for use at run time and indexes it in the class' method dictionary.

4.3 The Structure of a Constraint

As described in Section 1, a constraint represents a relation among the parts of an object that must always hold. Constraints are themselves objects. New kinds of constraints are defined by specifying both a *rule* and a set of *methods* for satisfying the constraint. Adding or modifying a constraint is a structural change; so only prototypes accept new constraints or allow existing ones to be edited.

Constraints are indexed in several tables in the prototype's class for easy retrieval during constraint satisfaction.

The constraint's methods describe alternate ways of satisfying the constraint; if any one of the methods is invoked, the constraint will be satisfied. These methods are represented as a list of instances of class `Method`. The constraint also has a matching list of instances of `MessagePlan`. Each message plan specifies how to invoke the corresponding method and describes its effects. When the constraint satisfier decides that one of the methods will need to be invoked at run time, the message plan that represents that method is asked to generate code that will send the appropriate Smalltalk message to activate the method. Exactly which methods are used depends on the other constraints and on the user's preferences as to what should be done if the object is underconstrained.

The rule is used to construct a procedural test for checking whether or not the constraint is satisfied and to construct an error expression that indicates how well the constraint is satisfied. Both the test and the error expression are instances of class `Method`. These methods are constructed in a fairly simple-minded way. If the constraint's rule equates numbers or points, the test checks that the two sides of the equation are equal to within some tolerance; the error will be the difference of the two sides of the equation. If the constraint is nonnumerical, the rule is used directly to generate the test; the error will be zero if the constraint is satisfied and one if it is not. If the user wants to override these default methods, he or she can replace them with hand-coded Smalltalk methods.

4.3.1 Example of a Constraint. Consider the structure described by the class `MidPointLine` used in the quadrilateral example.

Class `MidPointLine`

Superclasses

Geometric Object

Part Descriptions

line: a Line

midpoint: a Point

Constraints

midpoint = (line point1 + line point2)/2

midpoint ← (line point1 + line point2)/2

*line point1 ← midpoint * 2 - line point2*

*line point2 ← midpoint * 2 - line point1*

The class `MidPointLine` has a constraint that the midpoint lie halfway between the endpoints of the line. The constraint has three alternate ways of satisfying itself, as described by the methods listed under the rule. The first method alters the midpoint, the second one alters one endpoint of the line, and the third alters the other endpoint.

The user may want one method to be used in preference to another if there is a choice. This is indicated by the order of the methods: if the system has a choice about which method to use to satisfy the constraint, the first one on the list is used. In the case of the midpoint, the user preferred that the constraint be satisfied by moving the midpoint rather than by moving an end of the line.

4.3.2 Relations Among the Parts of a Constraint. The relations among the parts of a constraint are fairly rigidly defined. Each of the methods, if invoked,

must cause the constraint to be satisfied. For every part that is referenced by the rule, there must be either a method that alters that part or a dummy method referencing it. Currently, it is up to the user to see that these requirements are met; none of this is checked by the system.

As has been previously discussed, Smalltalk makes a strong distinction between the inside and the outside of an object. A method for satisfying a constraint is internal to the constraint and its owner, while the message plan that describes the method is the external handle of that method. It is the message plan that is used by the constraint satisfier in planning how to satisfy an object's constraints.

In particular, the path of a message plan describes the side effects of its method. The constraint satisfier uses this information to detect overlap in the parts affected by the various methods. Therefore, the more precisely one can specify which subparts are affected by the method, the more information the constraint satisfier has to work with. Also, the constraint satisfier can do more with a method if it is known that there is only one state of the subpart affected by the method that satisfies the constraint, given the states of all other parts. This is described by the Boolean variable *uniqueState* listed previously; in the example above, *uniqueState* is true.

This way of describing constraints allows the representation of relations that are not very tractable analytically. Any sort of relation can be expressed as a constraint, if a procedural test exists and some algorithm can be specified for satisfying the relation. In the most extreme case of analytical intractability, the constraint has a single method that affects the entire object that owns the constraint, and this message is not *uniqueState*. However, in such a case, the constraint satisfier has little to work with, and only one such constraint can be handled.

4.4 Merges

An important special case of a constraint is a *merge*. When several parts are merged, they are constrained to be all equal. For efficiency, they are usually replaced by a single object, rather than being kept as several separate objects. The owner of the parts maintains a symbolic representation of the merge for use by constraint satisfiers, as well as for reconstruction of the original parts if the merge is deleted. There are two principal uses of merging, both of which were illustrated by the introductory example in Section 2.1. The first use is to represent connectivity, for example, to connect the sides of the quadrilateral. The other is for applying predefined constraints, as was done with the midpoint constraint. As with constraints, adding or modifying a merge is a structural change; so only prototypes allow their merges to be edited. The process of merging is the same for both these uses. The object that owns the parts to be merged (e.g., *QTheorem*) is sent the message *merge: paths*, where *paths* is a list of paths to the parts to be merged.

When it can be done, the replacement of several merged objects by a single object yields a more compact storage format and speeds up constraint satisfaction considerably, since information need not be copied back and forth between the parts that have been declared equal. It does not result in any loss of information,

since the owner of the parts keeps a symbolic representation of the merge that contains enough information to reconstruct the original parts. On the other hand, it is slower to merge or unmerge parts, since more computation is required; so, for applications in which the structure of the object changes frequently, equality constraints would be more efficient. Another efficiency consideration is that a single merge can apply to an indefinite number of objects, while constraints have built into them the number of objects to which they apply. Thus, it is simple to make five separate points be equal using merges. To do this with equality constraints would require either that four separate constraints be used or that a special equality constraint be defined for use with five objects.

The most difficult parts of the ThingLab system to program and debug were those that deal with adding and deleting merges, due especially to interactions among merges at different levels of the part-whole hierarchy. For example, in the quadrilateral construction presented in Section 2.1, when merging the line part of the `MidPointLine` with the side of the quadrilateral, the system not only had to substitute a new line for the two line parts, but because of the merges connecting the sides of the quadrilateral it also had to substitute a new endpoint for the two connecting sides. In fact, at one point the author gave up in disgust and always represented merges by using equality constraints; but he eventually backtracked on this choice because it made things too slow for typical uses of ThingLab. Future implementers of systems using merges are hereby warned!

5. CONSTRAINT SATISFACTION

5.1 Overview

Constraint satisfaction is divided into two stages: planning and run time. Planning commences when an object is presented with a message plan. This message plan is not an actual request to do something; rather, it is a declaration of intent: a description of a message that might be sent to the object. Given this description, the object generates a plan to be used at run time for receiving such messages, while satisfying any constraints that might be affected. The results of this planning are compiled as a Smalltalk method. Directions for calling the compiled method are returned as a new message plan.

Consider the quadrilateral example described in Section 2.1. When the user selects *move Point* and first positions the cursor over a vertex of the quadrilateral, the ThingLab window composes a message plan and presents it to the quadrilateral. The quadrilateral decides how to move its vertex while still keeping all the midpoint constraints satisfied and embeds this plan in a compiled Smalltalk method. It then returns another message plan that gives directions for invoking that method. As the user pulls on the vertex with the cursor, the window repeatedly sends the quadrilateral a message asking it to update its position. This message invokes the Smalltalk method that was just compiled.

During planning, the object that is presented with the message plan creates an instance of `ConstraintSatisfier` to handle all the work. The constraint satisfier gathers up all the constraints that might be affected by the change and plans a method for satisfying them. The constraint satisfier first attempts to find a one-pass ordering for satisfying the constraints. There are two techniques available

for doing this: propagation of degrees of freedom and propagation of known states. If there are constraints that cannot be handled by either of these techniques, the constraint satisfier asks the object for a method for dealing with circularity. Currently, relaxation is the only such method available. If relaxation is used, the user is warned, so that perhaps some other redundant constraints can be supplied that eliminate the need for relaxation. Relaxation is described in Section 5.2.3.

5.2 Constraint Satisfaction Methods

The constraint satisfaction methods used in ThingLab are now described in more detail. To illustrate the operation of the methods, an electrical circuit example is used (Figure 16). Briefly, the classes involved are as follows. Instances of class Node are connection points. The parts of a node are a voltage and a set of currents flowing into that node; there is also a constraint that the sum of the currents be zero. (This is Kirchhoff's current law.) A subclass of Node is Ground, which has an additional constraint that its voltage be zero. Instances of Lead, like their physical counterparts, are used to connect devices. The parts of a lead are a node and a current; there is a constraint that the current belong to the node's set of currents flowing into it. Leads are connected by merging their nodes. There is a general class TwoLeadedObject, whose parts are two instances of Lead, and which has a constraint that the currents in the lead be equal and opposite. A number of subclasses of TwoLeadedObject are defined, including Resistor, Battery, Wire, and Meter; Meter in turn has subclasses Ammeter and Voltmeter. All these objects have appropriate constraints on their behavior: a resistor must obey the Ohm's law constraint relating its resistance, the current flowing through it, and the voltage across it; an ammeter must display the current flowing through it; and so forth. A complete listing of the ThingLab classes for building electrical circuit simulations is given in [2].

5.2.1 Propagation of Degrees of Freedom. In propagating degrees of freedom, the constraint satisfier looks for a part with enough degrees of freedom so that it can be altered to satisfy all its constraints. If such a part is found, that part and all the constraints that apply to it can be removed from further consideration. Once this is done, another part may acquire enough degrees of freedom to satisfy all its constraints. The process continues in this manner until either all constraints have been taken care of or no more degrees of freedom can be propagated.

Because of the difficulty of giving a precise definition of degrees of freedom for nonnumeric objects, the constraint satisfier uses a simpleminded criterion for deciding if a part has enough degrees of freedom to satisfy its constraints: it has enough degrees of freedom if there is only one constraint that affects it. It does not matter whether or not the constraint determines the part's state uniquely (removes all its degrees of freedom).

In deciding when a constraint affects a part, the part-whole hierarchy must be taken into account. The set of constraints that affect a given part is found by checking whether the path to the part overlaps the paths of any of the message plans generated by the constraints. Thus, a constraint on the first endpoint of a line affects the line as a whole, the first endpoint, and the x coordinate of the first endpoint; but it does not affect the line's second endpoint.

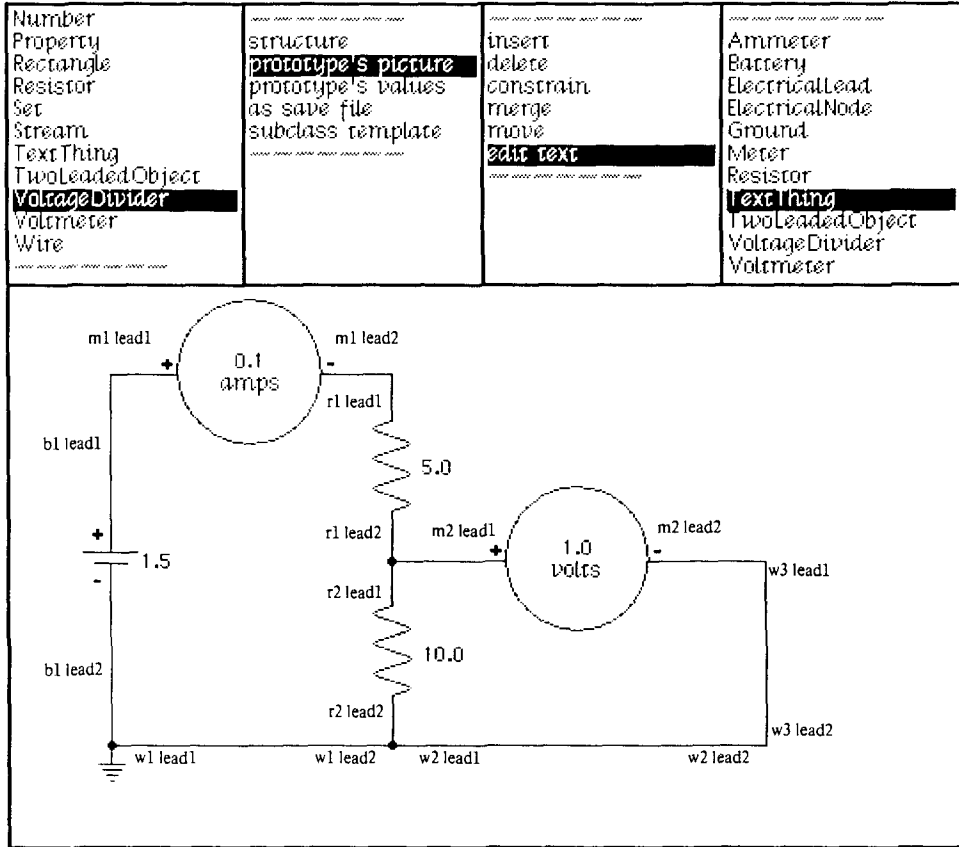


Fig. 16. A voltage divider.

In the voltage divider example, the text that displays the voltmeter's reading has only a single constraint on it: that it correspond to the voltage drop between *m2 lead1 node* and *m2 lead2 node*. Similarly, the text in the ammeter is constrained only by its relation to *m1 lead1 current*. Therefore, these pieces of text can be updated after the voltage drop and current are determined, and their constraints can be removed from further consideration. In this case, there are no propagations that follow.

5.2.2 Propagation of Known States. This method is very similar to the previous one. In propagating known states, the constraint satisfier looks for parts whose state will be completely known at run time, that is, parts that have no degrees of freedom. If such a part is found, the constraint satisfier looks for one-step deductions that will allow the states of other parts to be known at run time, and so on recursively. For the state of part *A* to be known (in one step) from the state of part *B*, there must be a constraint that connects *A* and *B* and that determines *A*'s state uniquely. This is indicated by the *uniqueState* flag on the message plan whose target is *A*. When propagating known states, the constraint satisfier can use information from different levels in the part-whole hierarchy: if

the state of an object is known, the states of all its parts are known; if the states of all the parts of an object are known, the state of the object is known.

If the state of a part is uniquely determined by several different constraints, one of the constraints is used to find its state, and run-time checks are compiled to see if the other constraints are satisfied.

In the example, this method would be used as follows. By the constraint on the ground, at run time *b1 lead2 node voltage* is known. (Actually, it was already known during planning, but the constraint satisfier does not use this information.) Also, by the battery's constraint, *b1 lead1 node voltage* is known, and it is the same as *m1 lead1 node voltage*. The ammeter has a constraint that there be no voltage drop across it, and so *m1 lead2 node voltage* is known. Similarly, the voltmeter has a constraint that it draw no current, and so the current in its leads and connecting wires is known. Finally, by the constraint on the wires, *w1 lead2 node voltage*, *w2 lead2 node voltage*, and *w3 lead1 node voltage* are all known.

The voltage at the node between the resistors, and all the other currents, are still unknown.

5.2.3 Relaxation. If there are constraints that cannot be handled by either of these techniques, the constraint satisfier asks the object for a method for dealing with circularity. Currently, relaxation is the only such method available (unless the user supplies more information; see below). Relaxation can be used only with objects that have all numeric values; also, the constraints must be such that they can be adequately approximated by a linear equation.

When relaxation is to be used, a call on an instance of Relaxer is compiled. At run time, the relaxer changes each of the object's numerical values in turn so as to minimize the error expressions of its constraints. These changes are determined by approximating the constraints on a given value as a set of linear equations and finding a least-mean-squares fit to this set of equations. The coefficients of each linear equation are calculated by noting the initial error and by numerically finding the derivative of the error expressions with respect to the value. Relaxation continues until all the constraints are satisfied (all the errors are less than some cutoff), or until the system decides that it cannot satisfy the constraints (the errors fail to decrease after an iteration).

Often, many more parts would be relaxed than need to be. To help ease this situation, a trick is used during planning. The trick is to try assuming that the state of one of the parts to be relaxed, say *P*, is known. This part *P* is chosen by looking for the part with the largest number of constraints connecting it to other still unknown parts. *P* is placed in a set *S*. Then the method of propagation of known states is invoked to see if the states of any other parts would become known as a result. All the parts which would become known, along with *P* itself, are eliminated from the set of parts to be relaxed. The process is repeated until the set of parts to be relaxed is empty. At run time, only the parts in *S* are relaxed. As each part *P* in *S* is relaxed, the system also computes the new states of the parts which had become known as a result of assuming that *P* was known. In computing the error in satisfying the constraints on *P*, the system considers the errors in satisfying the constraints on both *P* itself and also these other parts.

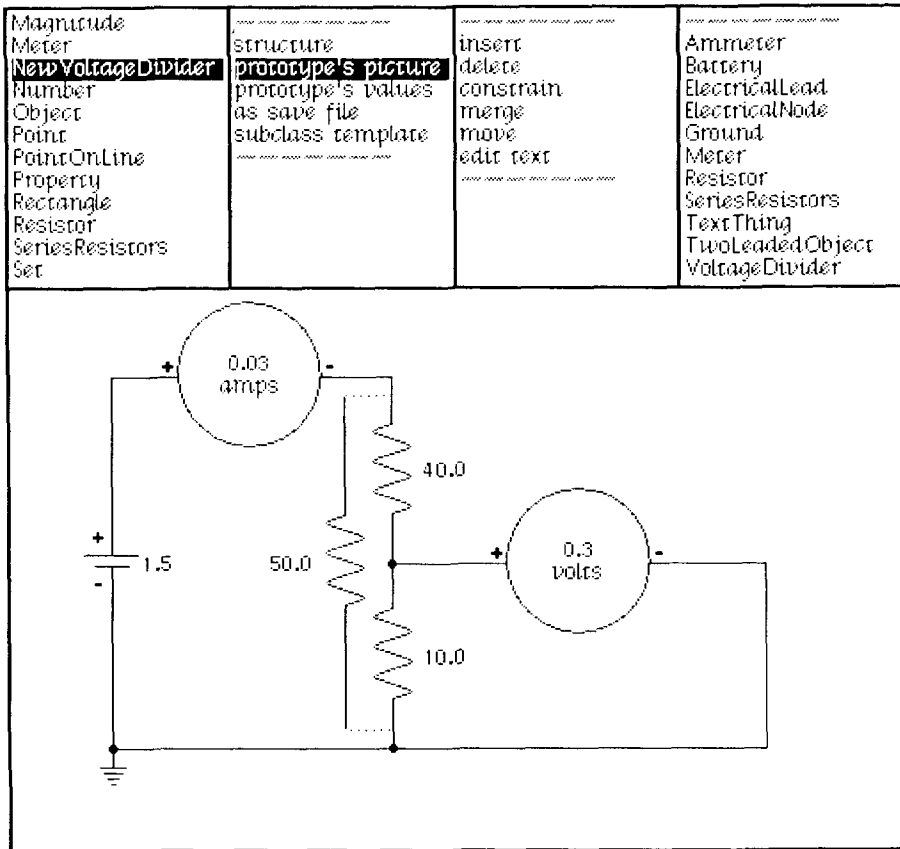


Fig. 17. The voltage divider with an added instance of SeriesResistors.

In the voltage divider, *r2 lead1 current* has three constraints connecting it to other unknowns: the Ohm's law constraint on *r2*, *r2*'s constraint inherited from *TwoLeadedObject*, and the Kirchhoff's law constraint on *r2 lead1 node*. No other unknown has more constraints, and so the system tries assuming that it is known. Given its value, *r2 lead1 node voltage* and all the other currents would be known. Therefore, at run time, only *r2 lead1 current* is relaxed.

5.2.4 *Using Multiple Views to Avoid Relaxation.* Using the method employed by Steele and Sussman [13], another view of the voltage divider may be added that obviates the need for relaxation. First, a new class *SeriesResistors* is defined that embodies the fact that two resistors in series are equivalent to a single resistor. An instance of *SeriesResistors* has three parts: resistors *rA* and *rB*, which are connected in series, and an equivalent single resistor *rSeries*. There is a constraint that the resistance of *rSeries* be equal to the sum of *rA*'s resistance and *rB*'s resistance.

To add this new description to the voltage divider, an instance of *SeriesResistors* is inserted in the circuit (call it *series*), and the resistors *rA* and *rB* of *series* are merged with the existing resistors *r1* and *r2* in the circuit (Figure 17).

Using this additional description, all the constraints can be satisfied in one pass. As previously described, *m1 lead2 node voltage* and *w1 lead2 node voltage* are both known. These are the same as *series rSeries lead1 node voltage* and *series rSeries lead2 node voltage*, respectively. Thus, by the Ohm's law constraint on *series rSeries*, *series rSeries lead1 current* is known. But this is the same current as *series rA lead1 current* and also the same as *r1 lead1 current*. Again by Ohm's law, the voltage at the midpoint, *r1 lead2 node voltage*, is known. All the other currents are also known.

It is appropriate to apply this redundant view to a pair of resistors in series only if there is no significant current flowing from the center node of the resistors. If this is not the case, then some of the constraints are not satisfiable, and the user is notified. However, in the present implementation there is no explicit representation of the fact that a redundant description has been provided; the system could do a better job of describing the reason that the constraints could not be satisfied if it knew about the use of such descriptions.

6. CONCLUSION

This paper has described ThingLab, a simulation laboratory. The system uses a number of concepts and techniques (in particular, constraints) that could add significant power to programming languages. A promising direction for future research is to explore the design of a full constraint-oriented programming language; work on this topic is underway, both by the author and by other researchers. Constraints will be taking an increasingly prominent position in our paradigms for programming in the years to come.

ACKNOWLEDGMENTS

Among the many people who have helped with this research, I would particularly like to thank all the members of the Learning Research Group at Xerox Palo Alto Research Center and my dissertation advisor, Terry Winograd. Thanks also to the referees for their useful comments.

REFERENCES

1. BOBROW, D., AND WINOGRAD, T. An overview of KRL, a Knowledge Representation Language. *Cognitive Sci.* 1, 1 (Jan. 1977), 3-46.
2. BORNING, A. ThingLab—A Constraint-Oriented Simulation Laboratory. Ph.D. dissertation, Dep. Computer Science, Stanford Univ., Stanford, Calif., March 1979 (revised version available as Rep. SSL-79-3, Xerox PARC, Palo Alto, Calif., July 1979).
3. DAHL, O.-J., AND NYGAARD, K. SIMULA—An ALGOL-based simulation language. *Commun. ACM* 9, 9 (Sept. 1966), 671-678.
4. ELCOCK, E.W., FOSTER, J.M., GRAY, P.M.D., MCGREGOR, J.J., AND MURRAY, A.M. ABSET, a programming language based on sets: Motivation and examples. In *Machine Intelligence*, vol. 6, B. Meltzer and D. Michie (Eds.). Edinburgh University Press, Edinburgh, Scotland, 1971, pp. 467-492.
5. HEWITT, C. Viewing control structures as patterns of passing messages. *Artif. Intell.* 8, 3 (June 1977), 323-364.
6. INGALLS, D.H.H. The Smalltalk-76 programming system: Design and implementation. In Conf. Rec., 5th Ann. ACM Symp. Principles of Programming Languages, Tucson, Ariz., Jan. 23-25, 1978, pp. 9-16.
7. KAY, A., AND GOLDBERG, A. Personal dynamic media. *Computer* 10, 3 (March 1977), 31-42.

8. LIEBERMAN, H., AND HEWITT, C. A session with TINKER: Interleaving program testing with program design. In Proc. 1980 LISP Conf., Stanford Univ., Stanford, Calif., Aug. 1980, pp. 90-99.
9. LISKOV, B., SNYDER, A., ATKINSON, R., AND SHAFFERT, C. Abstraction mechanisms in CLU. *Commun. ACM* 20, 8 (Aug. 1977), 564-576.
10. MITCHELL, J., MAYBURY, W., AND SWEET, R. Mesa language manual. Rep. CSL-79-3, Xerox PARC, Palo Alto, Calif., April 1979.
11. SMITH, D. PYGMALION: A creative programming environment. Rep. AIM-260, Dep. Computer Science, Stanford Univ., June 1975.
12. STEELE, G. The Definition and Implementation of a Computer Programming Language Based on Constraints. Ph.D. dissertation, Dep. Electrical Engineering and Computer Science, M.I.T., Cambridge, Mass., Aug. 1980 (available as MIT-AI TR 595, Aug. 1980).
13. STEELE, G.L., JR., AND SUSSMAN, G.J. Constraints. MIT AI Lab. Memo 502, M.I.T., Cambridge, Mass., Nov. 1978. Also in APL '79: Conf. Proc., *APL Quote Quad* (ACM SIGPLAN/STAPL) 9, 4 (June 1979), part 1, pp. 208-225.
14. STEELS, L. Reasoning modelled as a society of communicating experts. MIT-AI TR 542, M.I.T., Cambridge, Mass., 1979.
15. STEFIK, M. Planning with constraints (MOLGEN: part 1). *Artif. Intell.* 16, 2 (May 1981), 111-139.
16. SUTHERLAND, I. Sketchpad: A Man-Machine Graphical Communication System. Ph.D. dissertation, Dep. Electrical Engineering, M.I.T., Cambridge, Mass., 1963.
17. WULF, W.A., LONDON, R., AND SHAW, M. An introduction to the construction and verification of Alphard programs. *IEEE Trans. Softw. Eng. SE-2*, 4 (Dec. 1976), 253-264.

Received August 1980; revised April 1981; accepted May 1981