

THINGLAB -- A CONSTRAINT-ORIENTED SIMULATION LABORATORY

by Alan Borning

SSL-79-3 July 1979

Abstract: See next page

CR Categories: 4.22, 3.69, 8.1, 8.2

Key Words and Phrases: constraints, constraint satisfaction, object-oriented languages, part-whole hierarchies, interactive computer graphics, Sketchpad, Smalltalk

This report is a revised version of a Ph. D. thesis submitted to the Department of Computer Science at Stanford University. It is also available as Stanford Computer Science Department Report STAN-CS-79-746.

XEROX

PALO ALTO RESEARCH CENTER

3333 Coyote Hill Road / Palo Alto / California 94304

© 1979

by

Alan Hamilton Borning

Abstract

ThingLab is a system that provides an object-oriented environment for building simulations. Within this environment, part-whole and inheritance hierarchies are used to describe the structure of a simulation, while constraints are employed as a way of describing the relations among its parts.

One of the principal goals of this research has been to design and implement a language that helps the user describe complex simulations easily. Constraints are a particularly important tool for dealing with complexity, because they allow the user to specify independently all the relations to be maintained, leaving it up to the system to plan exactly how the constraints are to be satisfied.

ThingLab is implemented in the Smalltalk-76 programming language, and runs on a personal computer. Among the things that have been simulated using the system are constrained geometric objects, a bridge under load, electrical circuits, documents with constraints on their layout and content, and a graphical calculator.

Acknowledgements

It is a pleasure to thank Alan Kay and all the other members of the Learning Research Group at Xerox Palo Alto Research Center, both for help with this research and for providing a congenial and stimulating place in which to work. I would also especially like to thank my adviser, Terry Winograd, for his many helpful insights and patient advice. Among the other people who have aided in this effort, I would like to express my gratitude to Giuseppe Attardi, Danny Bobrow, John Seely Brown, Peter Deutsch, Carl Hewitt, Ken Kahn, Henry Lieberman, David Shaw, Gerry Sussman, Bert Sutherland, and Gio Wiederhold. The Xerox Corporation generously provided support for doing this research.

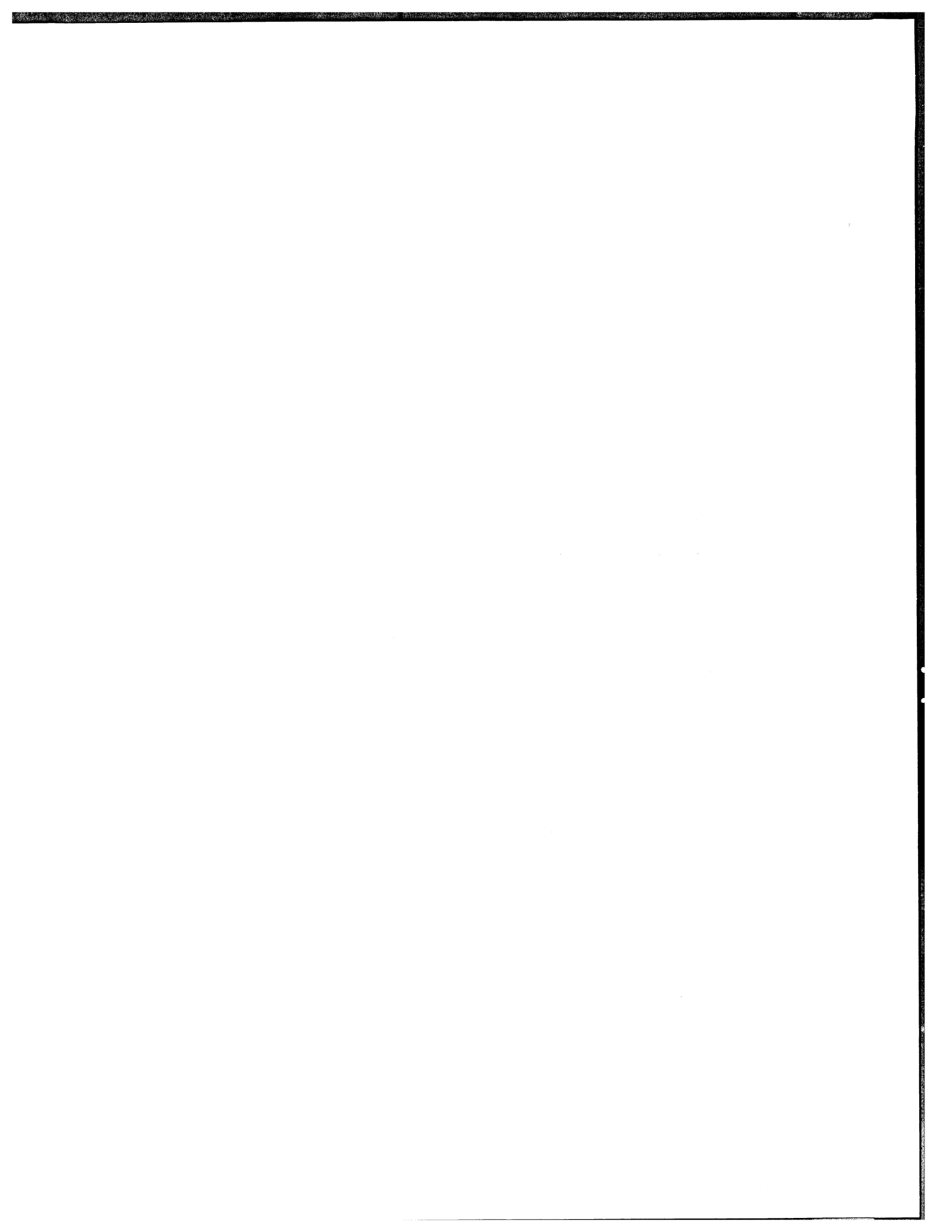
Table of Contents

Chapter 1	Introduction	1
Chapter 2	Some Scenarios	14
Chapter 3	Objects	38
Chapter 4	Constraints	52
Chapter 5	Constraint Satisfaction	65
Chapter 6	Directions for Future Research	86
	Bibliography	98

List of Illustrations

Figure 2.1 - A typical Smalltalk display	14
Figure 2.2 - Panes of the ThingLab window	15
Figure 2.3 - Filling in a subclass template	16
Figure 2.4 - Positioning the second endpoint of a line	16
Figure 2.5 - The completed quadrilateral	17
Figure 2.6 - Structure described by the class Quadrilateral	17
Figure 2.7 - Values of the prototype Quadrilateral	17
Figure 2.8 - Picture of the prototype MidPointLine	18
Figure 2.9 - Structure described by the class MidPointLine	18
Figure 2.10 - Adding a midpoint	19
Figure 2.11 - Deleting a line	19
Figure 2.12 - Moving a vertex of the quadrilateral	20
Figure 2.13 - A quadrilateral with anchored midpoints	21
Figure 2.14 - Picture of the prototype for Plus	22
Figure 2.15 - Picture of the completed Centigrade to Fahrenheit Converter	23
Figure 2.16 - A PrintingConverter	23
Figure 2.17 - Editing the Fahrenheit temperature	24
Figure 2.18 - The temperature converter with thermometers for input and output	24
Figure 2.19 - A quadratic equation network	25

Figure 2.20 - Constructing the class QuadraticSolver	26
Figure 2.21 - The network after adding an instance of QuadraticSolver	26
Figure 2.22 - A document with constraints	27
Figure 2.23 - The document after editing the number of employees in United PickPocket	28
Figure 2.24 - Moving the top of a bar	28
Figure 2.25 - A document with layout constraints	29
Figure 2.26 - Building a paned window	30
Figure 2.27 - Moving the corner of a pane in a paned window	30
Figure 2.28 - Two views of a triangle	31
Figure 2.29 - Two triangles connected by a scaling constraint	31
Figure 2.30 - Editing the scaling constraint	31
Figure 2.31 - A bridge under load	32
Figure 2.32 - Building a voltage divider	36
Figure 2.33 - The completed voltage divider	37
Figure 2.34 - Changing a resistance in the voltage divider	37
Figure 5.1 - Moving the vertex of a quadrilateral	65
Figure 5.2 - A voltage divider	66
Figure 5.3 - Picture of the prototype SeriesResistors	70
Figure 5.4 - The voltage divider with an added instance of SeriesResistors	70
Figure 5.5 - An instance of DemoRectangle	73
Figure 5.6 - An anchored horizontal line	83
Figure 6.1 - Successive states of a logic clock	90



Chapter 1 - Introduction

Overview of ThingLab

ThingLab is a simulation laboratory. It 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. Using the techniques developed for these domains, the system has also been used to model other sorts of objects, such as a graphical calculator, and documents with constraints on their layout and contents.

One of the principal goals of this research has been to design and implement a language that helps the user describe complex simulations easily. Starting with a combination of ideas from Sketchpad [Sutherland 1963] and Smalltalk [Ingalls 1978], a number of features have been built up. *Part-whole* and *inheritance* hierarchies are used to describe the structure of a simulation. *Constraints* are employed as a way of describing the relations among the parts of a simulation. Constraints are a particularly important tool for dealing with complexity, because they allow the user to specify all the relations independently of one another, leaving it up to the system to plan exactly how the constraints are to be satisfied.

ThingLab is implemented in the Smalltalk-76 programming language, and runs on the Alto, a personal computer developed at Xerox Palo Alto Research Center. All the simulations described herein are working examples.

The Kernel ThingLab System

The kernel ThingLab system consists of a Smalltalk extension, written by the present author, that is used in all ThingLab simulations. Embedded in this program is knowledge about such things as inheritance hierarchies, part-whole relations, and constraint satisfaction techniques. The kernel system doesn't have any knowledge about specific domains in which ThingLab can be used, such as geometry or electrical circuits. Rather, it provides tools that make it easy to construct objects that contain such knowledge.

Another goal in constructing the system, besides the exploration of language design as described above, was to investigate computer-based tools for use in education. For example, a ThingLab-style system might prove valuable as part of a geometry curriculum, or as an adjunct to a physics laboratory. With this in mind, it is anticipated that there would be two sorts of users of the system. The first sort of user 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 sort of user could then employ these building blocks to construct and explore particular simulations. The knowledge and skills required by these two kinds of users would be different. The first kind of user would need to know about message passing, the constraint specification language, and so on; and should also be familiar with the physical laws or rules for the given domain (e.g. Ohm's Law). The second kind of user, on the other hand, could deal with the system using only simple interactive graphics techniques, such as selecting items in a menu or moving pictures around on the screen. Thus, this sort of user wouldn't need to be familiar with either the details of ThingLab, or with the domain-specific theory behind the simulation (although one of the objectives of a curriculum might be for such a user to acquire this domain-specific knowledge).

Tools for Dealing with Complexity

Interesting systems are usually complex. A simulation that abstracts any significant portion of such a system will probably be complex as well, making it important for the simulation system to provide tools for dealing with this. This is particularly true if the system is intended for use by people who aren't computer sophisticates, as is ThingLab. This section is a discussion of the tools available in ThingLab viewed in this light.

Objects

ThingLab, like Smalltalk, is organized around the idea of *objects* that communicate by sending and receiving *messages*. This object-oriented factorization of knowledge provides one of the basic organizational tools. For example, in representing a geometric construction, the objects used in the representation would be things such as points, lines, circles, 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.

Hierarchical Decomposition

A powerful method for dealing with complexity is to describe a system hierarchically. ThingLab provides two kinds of hierarchies: an inheritance hierarchy, and a part-whole hierarchy.

The inheritance hierarchy uses Smalltalk's class-instance structure. 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; the constraints on its instances; and a dictionary of messages that its instances understand, along with methods for computing the appropriate responses. An instance holds the particular values that distinguish it from other instances of its class.

For example, one of the classes defined in ThingLab is class Line. This class describes lines in general -- it is a description of all line instances, both existing and potential. It specifies that every line instance should be represented as a pair of endpoints. Lines themselves have no constraints. Finally, the class Line contains a dictionary of messages that its instances understand, along with methods for receiving those messages. Some examples of messages that lines understand are:

point1 - return your first endpoint

printon: stream - print a description of yourself on an output stream

showpicture: window - show your picture in a window.

A line instance has a pointer to its class, and two fields that hold pointers to its endpoints, which are instances of class Point.

A new class may be defined as a *subclass* of one or more existing classes. The subclass inherits the storage specifications, constraints, and message protocol of its superclasses. It may specify additional storage requirements, constraints, and message behavior of its own; it may delete or replace inherited constraints; and it may override inherited responses to messages. For example, one of the subclasses of Line is HorizontalLine. HorizontalLine inherits all the traits of Line, and adds a constraint that the *y* values of its endpoints be equal. The most abstract class in the system is class Object; all other classes in the system are descended from it.

The other kind of hierarchy in ThingLab is the part-whole hierarchy. In ThingLab, every object is composed of named *parts*, each of which is itself an object. The parts are thus composed of subparts, and so on. For example, a triangle is composed of three parts: its sides. Each side in turn has two endpoints; the endpoints have *x* and *y* coordinates. Note the relation between the inheritance and part-whole hierarchies: the triangle is an instance of class Triangle; its parts are instances of class Line; their endpoints are instances of class Point; and finally the coordinates are instances of class Integer. Each class contains a set of *part descriptions* that list the common properties of the corresponding parts of each of its instances. Thus, the class Triangle specifies that each of its instances should have three parts named *side1*, *side2*, and *side3*; and that each of these parts should be instances of class Line.

It is rare for the decomposition of an object into parts to be completely linear, i.e. such that there are no interactions among the parts. For example, each side of the triangle shares its endpoints with the other sides. If the triangle were constrained to have a horizontal base, there would be other interactions among the endpoints (the *y* values of the endpoints of the base would have to be equal). ThingLab provides ways of representing and reasoning about these

interactions, as will be described in the next section.

Constraints

The basic tool for representing relations among parts is the *constraint*. A constraint specifies a relation that must always be satisfied. Here are some examples of constraints that have been defined in ThingLab by its various users:

- a constraint that a line be horizontal;
- a constraint that one triangle be twice as big as another;
- a constraint that the digits displayed in an editable paragraph correspond to the height of a bar in a bar chart;
- a constraint that a resistor obey Ohm's law;
- a constraint determining the gray scale level of an area on the computer's display; and
- a constraint that a rectangle on the display be precisely big enough to hold a given paragraph.

A key fact about constraints is that the relation to be satisfied is separated from the process by which it is satisfied. All the constraints on a simulation can be specified independently of one another. It is up to the system to decide whether or not they can all be satisfied simultaneously, and if so, how to satisfy them. Constraints are thus a powerful tool for dealing with complexity. Although the interactions among the parts of the system may be numerous, the user can specify each relation without worrying about the others.

Constraints reflect a way that humans might think about a large class of situations. In mathematics, the hypotheses of a theorem, along with the fundamental axioms, may be viewed as constraints. The laws of physics may be thought of as constraints that physical objects must obey. Other branches of the physical and social sciences have built up some laws of this sort, e.g. valences in chemistry. In design, constraints represent the requirements that the thing being designed must satisfy.

In a simulation environment, constraints serve not only as descriptions of the simulated object, but also as commands to the system telling it that certain conditions must be satisfied. This makes the model of a system a dynamic, reactive one -- the user can prod it and observe its responses.

The User Interface

The ThingLab user interface incorporates a number of tools for dealing with complexity. Multiple viewpoints are supported: a typical object can be depicted in several ways, for example, as a picture, as a structural description, or as a table of values. The object itself defines the views that it can provide. Some quite general graphical editing tools are provided,

and purely graphical objects, such as a triangle, can be constructed using graphical techniques alone. When the user edits an object, say by selecting a point and moving it with the cursor, the user interface automatically asks the object for a plan for moving the point while keeping all its constraints satisfied.

Constraint Representation and Satisfaction

Representation of Constraints

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 checking 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.

Merges

An important special case of a constraint is a *merge*. When several parts are merged, they are constrained to be identical. For efficiency, they are usually replaced by a single part, rather than being kept as several separate objects. The owner of the parts keeps a symbolic representation of the merge for use in constraint satisfaction, as well as for reconstruction of the original parts if the merge is deleted. One use of merging is to represent connectivity. For example, to connect two sides of the triangle, an endpoint from one side is merged with an endpoint of the other. Another use of merging is to apply pre-defined constraints. Thus, to constrain the base of the triangle to be horizontal, one can simply merge an instance of `HorizontalLine` with the side to be constrained.

Constraint Satisfaction

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 multi-directional. For example, the horizontal line constraint is allowed to change either endpoint of the line. 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.

The approach taken in ThingLab is first to analyze the constraints on an object and plan how to satisfy them, and then to make the actual changes to satisfy the constraints. In ThingLab, the

particular values that an object holds usually change much more rapidly than its structure. For example, if on the display the user moves some part of a constrained geometric object with the cursor, the values held by this object will change every time its picture is refreshed. Each time some value is changed, other values may need to be changed as well to keep the constraints satisfied. However, the object's structure will change only when the user adds or deletes a part or constraint. The design of the ThingLab constraint satisfaction mechanism is optimized for this environment. A constraint satisfaction plan may depend on the particular structure of an object, but should work for any values that the object might hold. (If not, appropriate tests must be included as part of the plan.) Once a plan for satisfying some constraints has been constructed, Smalltalk code is compiled to carry out this plan. Thus, each time the part of the constrained geometric object is moved, it is this pre-compiled method that is invoked, rather than a general interpretive mechanism. Also, the plan is remembered in case it is needed again. Planning is done using symbolic references to the constrained parts, so that the same plan may be used by all instances of a class. If the class structure is changed so that the plan becomes obsolete, it will be automatically forgotten.

When an object is asked to make a change to one of its parts or subparts, it gathers up all the constraints that might be affected by the change, and plans a method for satisfying them. In planning a constraint satisfaction method, the object will first attempt to find a one-pass ordering for satisfying the constraints. There are two techniques available in ThingLab for doing this: propagation of degrees of freedom, and propagation of known states. In propagating degrees of freedom, the constraint satisfier looks for an object with enough degrees of freedom so that it can be altered to satisfy all its constraints. If such an object is found, that object and all the constraints that apply to it can be removed from further consideration. Once this is done, another object 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 until no more degrees of freedom can be propagated. In the second technique, propagating known states, the constraint satisfier looks for objects whose states are completely known. If such an object is found, the constraint satisfier will look for one-step deductions that allow the states of other objects to be known, and so on recursively.

If there are constraints that cannot be handled by either of these techniques, the object will invoke a method for dealing with circularity. Currently, the classical relaxation method is the only such method available. As will be described in Chapter 5, relaxation can be used only with certain numerical constraints, and is also slow. In this method, the object changes each of its 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 by finding the derivative of the error expressions with respect to the value, and solving this set of equations. 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).

If the relaxation method is used, the system issues a warning message to the user. The user can either let things stand, or else supply additional information in the form of redundant constraints that eliminate the need for relaxation.

Where are Constraints Useful?

Where are constraints useful? In discussing this question, it is important to differentiate what can be *expressed* using constraints from what sets of constraints can be *satisfied*. Many more things can be expressed than can be satisfied. For example, it is easy to state the following constraints:

$$x^n + y^n = z^n$$

x, y, z, n integers

$x, y, z > 0$

$n > 2$.

However, finding values that satisfy these constraints, or proving that no such values exist, requires either a counterexample or a proof of Fermat's Last Theorem.

What can be expressed using constraints? To express a relation as a constraint, the following information is needed: a rule (from which the system will derive a satisfaction test and an error expression); and one or more methods for satisfying the constraint. For numerical constraints, the methods may be omitted if the user is willing to live with the relaxation method. Any relation that must always hold, and for which this information can be supplied, may be expressed as a constraint. Some relations that cannot be expressed as constraints in a general way using current ThingLab techniques include: any relation involving ordering or time; relations that need hold only under certain conditions; and meta-constraints (constraints on other constraints or on constraint satisfaction strategies).

What sets of constraints can be satisfied? If the constraint dependency graph has no circularities, or if the circularities can all be broken using one-step deductions, then the one-pass constraint satisfaction techniques will always succeed, and will provide correct results. Further, the constraints can be satisfied, or determined to be unsatisfiable, in time proportional to that required to execute the local methods provided by the constraints. If the dependency graph does have circularities that cannot be broken by one-step deductions, the constraints can still be satisfied if the circular parts of the graph all involve numerical constraints for which relaxation can be used. These constraints must either be linear, or else constraints for which linearization is an adequate approximation. An example of a set of circular constraints for which the relaxation method does *not* work are those that describe a cryptarithmic problem, e.g. DONALD + GERALD = ROBERT with $D=5$. [See Newell & Simon 1972 for a description of this domain.] Relaxation is useless here, since the constraints cannot be approximated by linear equations. To solve such kinds of problems, other constraint satisfaction techniques would be needed, such as heuristic search.

Relation to Other Work

As mentioned previously, the two principal ancestors of ThingLab are Sketchpad and Smalltalk. It is also closely related to work on constraints by Gerald Sussman and his students; other related work includes Simula, the Actor languages, KRL, and a number of problem solving systems. Following a discussion of these and other systems, a summary of the novel features of ThingLab is presented.

Sketchpad

One of the principal influences on the design of ThingLab has been Sketchpad [Sutherland 1963]. Sketchpad was a general-purpose system for drawing and editing pictures on a computer. The user interacted directly with the display, using a light pen for adding, moving, and deleting parts of the drawing. Sketchpad's influence on the field of computer graphics has been tremendous. However, it contained many other important ideas besides that of interacting with a computer using pictures; and these ideas have been less widely followed up. In reading the following description, remember that this program was written in 1962!

Sketchpad allowed the user to define new kinds of pictures by composing primitive picture types (points, line segments, and circle arcs), and other user-defined pictures. These picture definitions could be used in two ways. First, the user could *copy* the picture definition, and use this copy in composing another picture. No record was maintained of the relation between the copy and the original, and the user was free to modify the copy in any way. Second, the picture definition could be used as a *master* for making arbitrarily many *instances*. Each instance had the same shape as the master, but could be translated, rotated, and scaled. In this case, if the master was edited, each instance would change correspondingly. In the master, certain points could be designated as *attachers*. The corresponding points in each instance could be used to connect it to points in the rest of the picture.

Constraints were used to specify conditions that the picture had to satisfy. For example, one could constrain a line to be horizontal, or a point to lie on a line. Constraints were uniformly described using error expressions, each of which returned a number indicating how well the constraint was satisfied. The system would adjust the constrained variables to minimize the values of these error expressions. Two methods for satisfying constraints were available: propagation of degrees of freedom ("the one-pass method") and relaxation.

The operation of recursive merging was used to connect parts of the drawing, and to apply pre-defined constraints. For example, to connect two sides of a polygon, an endpoint from one line was merged with an endpoint from the other. To constrain a line to be horizontal, first the constraint definition was copied, and then each endpoint of the copy was merged with the corresponding endpoint of the line in the picture. The resulting topology of the picture was

explicitly stored in ring structures. For example, every point had a ring of lines that terminated on it, while every line had pointers to its endpoints. These structures were automatically updated as the user edited the picture.

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 extends Sketchpad in a number of ways. The Sketchpad domain of constrained geometric objects has been expanded to include domains that are not purely graphical. For example, an object like a resistor has a picture, but also contains information such as its resistance and the voltage across it. ThingLab uses Smalltalk's class-instance structure. This mechanism is more general than the master-instance relation in Sketchpad, since Smalltalk instances have internal variables that can hold whatever instance state is desired.

Constraints in ThingLab can apply to non-numeric objects such as text, as well as to numeric values. While Sketchpad constraints were uniformly described using error expressions, in ThingLab local procedures for satisfying the constraint may be included as part of its definition. [ThingLab started out using only error expressions. Later, the use of local procedures was permitted to allow constraints to apply to non-numeric objects, as well as to speed up the program.] In addition to the Sketchpad constraint satisfaction methods, ThingLab provides a method for propagating known states. Constraint satisfaction in ThingLab has been divided into two stages: planning and run-time. During planning, a plan is generated for satisfying the constraints, and is then compiled as a Smalltalk method. At run-time, it is this compiled code that is invoked. Typically, the same plan will be used many times.

Internally, ThingLab objects are stored in a manner somewhat different from that used in Sketchpad. In contrast to Sketchpad's extensive ring structures, ThingLab uses no back pointers; rather, information about constraints and merges is represented using symbolic paths to the affected parts. These constraints and merges are associated with an entire class, and apply to each instance of that class. The ThingLab scheme has the advantage that objects are stored much more compactly, and less manipulation of pointers is required. On the other hand, it requires that a new class be defined whenever an object with a new kind of topology is to be constructed. It would be useful also to have a class of objects represented in such a way that constraints and merges could be associated with particular instances of that class, while preserving the efficiency of the current scheme for places where it is more appropriate.

Smalltalk

The other principal ancestor of ThingLab is Smalltalk [Kay 1972a, Kay 1972b, Kay & Goldberg 1977, Kay 1977, Ingalls 1978]. ThingLab started out as a system *simulated* in Smalltalk, but has evolved to become an *extension* of Smalltalk. The important ideas in Smalltalk -- objects, classes and instances, and messages -- are now all used directly in ThingLab. As these ideas were needed

in presenting the overview of ThingLab, they have been described previously. [A reader who is unfamiliar with Smalltalk should be able to understand Chapters 1 and 2; however, the remaining chapters assume familiarity with Ingalls 1978.]

ThingLab adds a number of new features to Smalltalk, including constraints, a facility for defining classes by example, and multiple superclasses. Also, a large number of declarative structures have been implemented that describe information previously embedded only in Smalltalk's computational methods and object references. Most of these declarative structures have been implemented in response to the demands of constraint satisfaction. In satisfying constraints, it is necessary to reason about the interactions among the parts of an object. To allow this, the static structure of an object is described using parts and part descriptions. Any sharing of parts must be explicitly represented using a merge. The dynamic relations among the parts are represented with constraints.

Smalltalk is an evolving system. Discussion regarding the next major version of Smalltalk is currently under way in the Learning Research Group, and it is likely that many of the ideas developed in ThingLab will find their way into the Smalltalk language itself. Some thoughts on this topic may be found in Chapter 6.

Work by Gerald Sussman and his Students

ThingLab is related to recent work on a constraint language at MIT by Guy Steele and Gerald Sussman [Steele & Sussman 1978], and also to other work by Sussman and his students on the problem of applying artificial intelligence techniques to computer-aided design [Sussman & Stallman 1975, Stallman & Sussman 1977, Doyle 1977, de Kleer & Sussman 1978].

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. Using these primitive constraints, compound constraints can be built up. This is much like the method used in the ThingLab calculator example described in Chapter 2. These similarities are interesting, given the rather different environments in which the systems were written (LISP and Smalltalk).

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. They note that powerful algebraic manipulation techniques alone are not enough to solve many interesting problems that can be solved by people; rather, ways are needed of organizing the solution so that the system can use canned theorems, coupled with simple algebra only. Multiple views are one way of encapsulating such theorems. Their use of multiple views has been adopted in ThingLab directly, and the voltage divider example in Chapter 5 is taken from their work. ThingLab does

not have any symbolic algebraic capabilities.

Their language retains dependency information -- a record of the justifications for each conclusion -- to identify which constraints are responsible in the event of an inconsistency, for use in propagating the effects of an edit, and to allow efficient backtracking when search is needed (dependency-directed backtracking). [Since ThingLab has no dependency information, when the structure of an object changes it checks more things, and throws away more constraint satisfaction plans, than it really needs to.]

On the other hand, ThingLab has a number of features that are not present in their language. Steele and Sussman have an abstraction mechanism like the one used in ThingLab for building a class given a prototypical example, but do not have a general inheritance hierarchy that allows subclassing. Their system does not have any graphics. In regard to constraints, ThingLab allows constraints on non-numerical objects such as text, as well as on numerical quantities, and can express preferences in addition to absolute requirements. Also, it incrementally compiles the results of constraint satisfaction planning, rather than using an interpreter.

Early Constraint Languages

Ideas about constraint languages have been around for some time. M. V. Wilkes [Wilkes 1964] proposed that constraint statements be allowed in an Algol-like programming language. The compiler would use symbolic differentiation to find a linearized form of the constraint statement; at run-time, relaxation techniques would be used to satisfy the constraints. Richard Fikes [Fikes 1970] constructed a system, REF-ARF, for solving problems stated as procedures. In the programming language REF, *select* functions were used to indicate the permissible values for a variable, while *condition statements* were used to build up sets of constraints that the variables had to satisfy. ARF, the problem solver, then attempted to find values for the variables that satisfied all the conditions by first using a number of rather clever constraint manipulation methods to limit the possible values of the variables, followed by a GPS-style search to find an answer. ABSET [Elcock *et al.* 1971] was a set-oriented language developed at the University of Aberdeen. It allowed statements of the form $A+B=3$ AND $A=1$; given this statement, it could deduce B 's value. ABSET had a number of interesting features: it emphasized the avoidance of unnecessary ordering restrictions in the statement of a program; and it allowed assertions (or constraints) to apply to non-numeric objects such as sets or text.

Other Languages

One of the principal ancestors of Smalltalk is the programming language Simula, which was one of the first systems to use the concepts of classes, subclasses, and instances [Dahl & Nygaard 1966, Dahl, Myhrhaug, & Nygaard 1970]. The Simula notion of an *event* plays an important part in the proposed way of dealing with constraints on time that is described in Chapter 6. The idea of

objects that communicate by passing messages is used in the Actor languages developed by Carl Hewitt and his students [Hewitt 1976, Yonezawa and Hewitt 1977]. Both the Actor languages, and the Director language developed by Kenneth Kahn [Kahn 1978], have also been very useful in thinking about constraints on time.

Other relevant work includes representation languages, in particular KRL [Bobrow & Winograd 1977a, Bobrow & Winograd 1977b]. Ideas in KRL regarding object-centered factorization of knowledge and the inheritance of properties have been very helpful. A comparison of the approaches taken in KRL and ThingLab to the questions of inheritance and the relation between classes and instances may be found in Chapter 3. Such questions also arise in the design of semantic nets [see e.g. Woods 1975, and Brachman 1976].

Problem Solving Systems

There is a large body of work in artificial intelligence on problem solving systems of various kinds. Most of these systems are concerned with more complex problem-solving tasks than those tackled in ThingLab. In 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 graphical 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.

In the domain of physics, Johan de Kleer describes a program, Newton, that understands and solves problems in the mini-world of objects moving on surfaces [de Kleer 1975]. Emphasis is placed on planning the solution to a problem using the technique of *envisionment*, or qualitative simulation of the event. Another mechanics problem solving system is the MECHO program [Bundy 1978], which has been used for a number of kinds of mechanics problems, including pulley problems, and also the roller-coaster problems investigated by de Kleer.

A rather different sort of physics problem solver is the Mechanisms Lab developed by Chuck Rieger and Milt Grinberg [Rieger and Grinberg 1977]. The Mechanisms Lab uses a cause-effect representation to describe both natural and artificial systems. Given such a declarative representation of a system, their program can then translate this representation into a population of associatively triggerable procedures, which can in turn be used to simulate the system under consideration.

There has also been considerable work in the more general domain of planning how to solve a problem, from the venerable General Problem Solver onward. In the work of Sacerdoti on planning nets [Sacerdoti 1975], and the related work of Tate [Tate 1977], plans are represented as a partial ordering of actions with respect to time, without premature commitments to a particular order for achieving subgoals. This methodology is compatible with the approach to constraint

specification taken in ThingLab, and may prove useful in expressing and satisfying constraints involving time (see Chapter 6).

ThingLab

Finally, a preliminary description of ThingLab itself was published in the IJCAI-77 proceedings [Borning 1977].

Novel Features of ThingLab

It is customary in documents such as this to give an explicit statement of what is new. Here is the author's list of novel features of ThingLab:

- ▶ An hierarchical part-whole representation of objects has been developed that includes an explicit, symbolic representation of shared substructure. Virtually the same representation has been independently invented by Steele and Sussman.
- ▶ The Smalltalk class-instance structure has been extended by the inclusion of multiple superclasses, prototypes, and a facility for class definition by example.
- ▶ A representation for constraints is used that bundles together a declarative description of the constraint with procedures for satisfying it.
- ▶ Constraint satisfaction techniques have been implemented that incrementally analyze constraint interactions and compile the results of this analysis into executable code.
- ▶ A user interface has been implemented that provides multiple views on each object, along with appropriate editing facilities for these views. The object itself defines the views that it can provide.

Chapter 2 - Some Scenarios

An Introductory Example

This chapter presents some examples of ThingLab in operation. As an introductory example, we will use ThingLab to construct a quadrilateral and view it in several ways. We will then use the system to demonstrate a theorem about quadrilaterals. Before presenting the example, a brief introduction to the ThingLab user interface is needed.

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 user interface is part of the kernel ThingLab system, and was adapted from the Smalltalk class editor designed by Larry Tesler. See Ingalls 1978, and Goldberg & Robson 1979, for more information about windows and the Smalltalk user interface.]

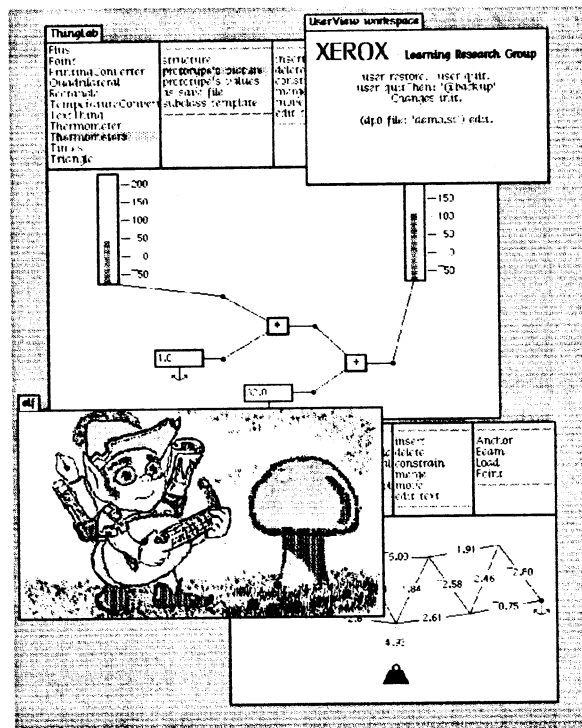


Figure 2.1 - A Smalltalk display

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 pane immediately to the right. The class shows itself in the chosen format in the large pane at the bottom of the window labelled *picture*.

The two remaining panes, messages and arguments, contain menus used for graphical editing of the class's 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. [A black stripe in a menu pane indicates a selected item. Thus in Figure 2.2, *Triangle* and *prototype's picture* have been selected.]

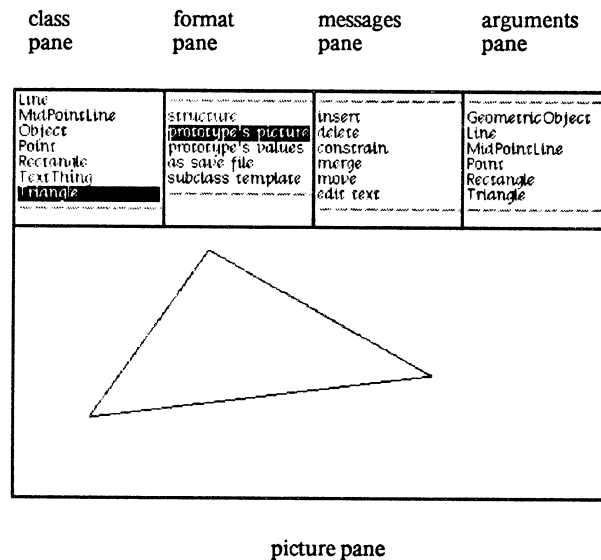


Figure 2.2 - Panes of the ThingLab window

Defining the Class of Quadrilaterals

The first thing we will do in this example is to 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.] We select *GeometricObject* in the class pane, and the phrase *subclass template* in the format pane (see Figure 2.3). *GeometricObject* responds by displaying in the picture pane a template for making a new subclass of itself, which we fill in by typing the name *Quadrilateral*. The system creates the new subclass, adds its name to the menu in the class pane, and selects the new menu item.

SOME SCENARIOS

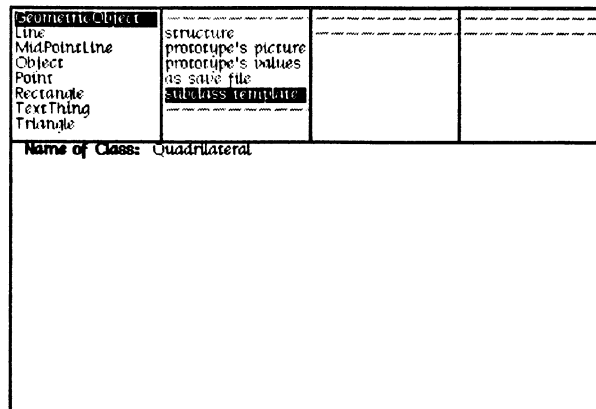


Figure 2.3 - Filling in a subclass template

One of the important features of the ThingLab environment is that the user can define classes by example. More precisely, the structural aspects of a class (its part descriptions and constraints), may be specified incrementally by editing its prototypical instance. We will define the class Quadrilateral in this way. We select the words *prototype's picture* in the format pane. The class Quadrilateral creates its prototypical instance, and asks this instance to show itself. So far, the prototype has no parts, and so its picture is blank. In addition to showing its picture, the prototype lists the editing messages that we may send to it in the message pane, and the possible classes of the arguments for these messages in the argument pane. [The lists of formats, messages, and arguments are each obtained by sending a message to the class being edited. In this case, the lists of formats and messages were both the defaults inherited from class Object, while the list of argument classes was inherited from GeometricObject.]

We will edit the prototype by adding and connecting four sides. We select the word *insert* in the message pane, and the word *Line* in the argument 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 2.4). [All this behavior arises because the class Line declares the two endpoints of each line instance to be *attachers*.]

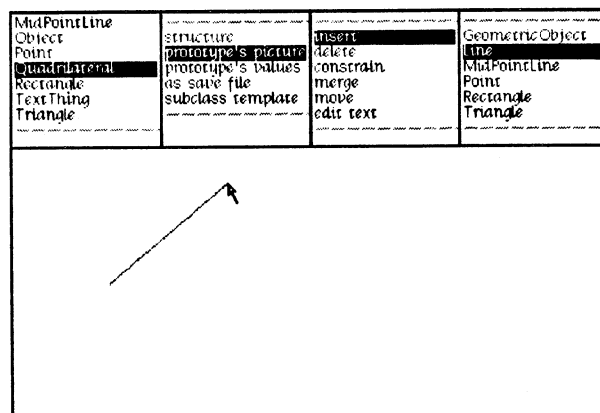


Figure 2.4 - Positioning the second endpoint of a line

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 will lock onto the stationary point, and the line will stop 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 2.5).

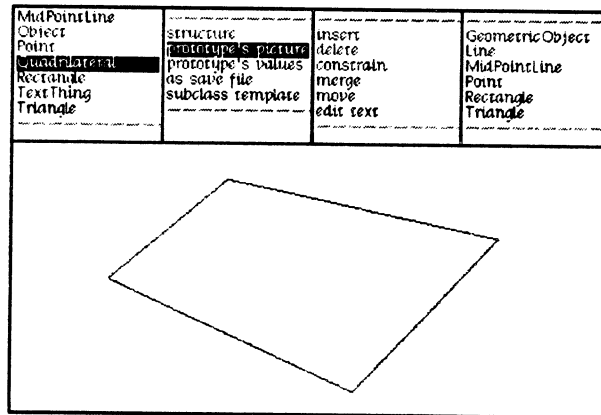


Figure 2.5 - The completed quadrilateral

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 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 2.6). We may also view the values stored in the prototype by selecting *prototype's values* (Figure 2.7). [In the table of values, a point with x=10 and y=20 is written as 10 @ 20.]

MidPointLine Object Point Quadrilateral Rectangle TextThing Triangle	structure prototype's picture prototype's values as save file subclass template		
Class Quadrilateral Superclasses GeometricObject Part Descriptions part1: a Line part2: a Line part3: a Line part4: a Line Merges part2 point2 = part3 point1 part1 point1 = part4 point2 part3 point2 = part4 point1 part1 point2 = part2 point1			

Figure 2.6

Structure described by the class Quadrilateral

MidPointLine Object Point Quadrilateral Rectangle TextThing Triangle	structure prototype's picture prototype's values as save file subclass template		
Quadrilateral prototype part1: a Line point1: 74 @ 105 point2: 166 @ 27 part2: a Line point1: 166 @ 27 point2: 377 @ 72 part3: a Line point1: 377 @ 72 point2: 264 @ 193 part4: a Line point1: 264 @ 193 point2: 74 @ 105			

Figure 2.7

Values of the prototype Quadrilateral

Having constructed the class Quadrilateral, we index it in the list of classes useful in geometric constructions. [This is currently done by typing and executing a Smalltalk statement; eventually, this should be handled graphically.]

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 will form a parallelogram. [The idea of demonstrating this theorem with ThingLab was suggested by John Seely Brown.] In the construction, instances of the class *MidPointLine* will be used to represent bisected lines. The class *MidPointLine* 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. This class has already been constructed by an experienced user as part of the package of geometric classes. Let's look at it in two different formats (Figures 2.8 and 2.9).


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

Figure 2.8

Picture of the prototype *MidPointLine*

MidPointLine	structure		
Object	prototype's picture		
Point	prototype's values		
Quadrilateral	as save file		
Rectangle	subclass template		
TextThing			
Triangle			
Class MidPointLine Superclasses GeometricObject Part Descriptions line: a Line midpoint: a Point Constraints midpoint = (line point1 + line point2) / 2 midpoint ← (line point1 + line point2) / 2 line point1 ← line point2 + ((midpoint-line point2)*2) line point2 ← line point1 + ((midpoint-line point1)*2)			

Figure 2.9

Structure described by the class *MidPointLine*

To perform the construction, we will make a new class named *QTheorem*. As before, we create it as a subclass of *GeometricObject*, and define it by example. We select *prototype's picture* in the format pane. We will 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 appears of a quadrilateral whose shape has been copied from the prototype. Since we didn't declare otherwise, the entire instance is the attacher. [Another reasonable choice would have been to designate the four corners as attachers. Designation of attachers is currently handled by typing and executing a Smalltalk statement.] We position the quadrilateral and press a button.

The next step is to add midpoints to the sides of the quadrilateral. We select the message *constrain* and the argument *MidPointLine*. A blinking picture of an instance of *MidPointLine* appears. As with the quadrilateral, the shape of the new *MidPointLine* instance has been copied from the prototype. When the action is *constrain*, its line part is the attacher. We move the *MidPointLine* near the center of one of the sides of the quadrilateral and press the button, thus merging the line part of the midpoint with the side of the quadrilateral. Similarly, we add midpoints to the other three sides (Figure 2.10).

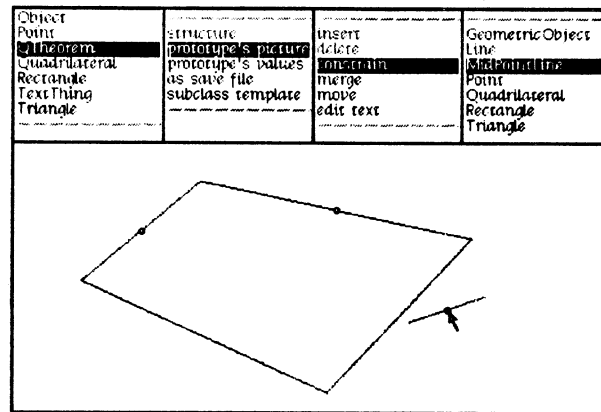


Figure 2.10 - Adding a midpoint

The last step is to add four lines connecting the midpoints to form the parallelogram. If we make a mistake along the way, the *delete* message may be used to remove the offending object. For example, to delete a line, we select *delete* and *Line*. A complemented image of a line appears that is attached to the cursor (Figure 2.11). This anti-line sticks to lines in the picture. When we position it near the unwanted line and push the button, the line and the anti-line annihilate each other.

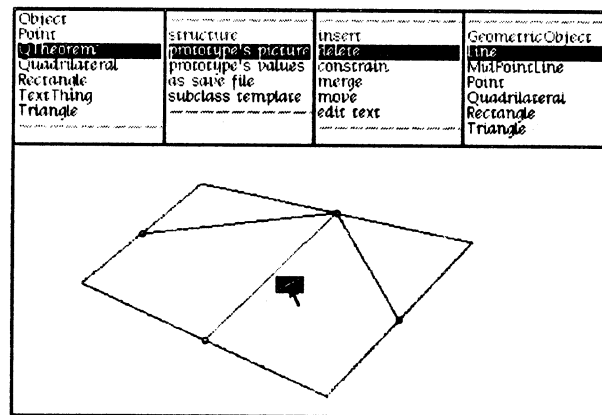


Figure 2.11 - Deleting a line

Once the construction is complete, we may move any of the parts of the prototype *QTheorem* and observe the results. In general, it will not be 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 2.12). [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!

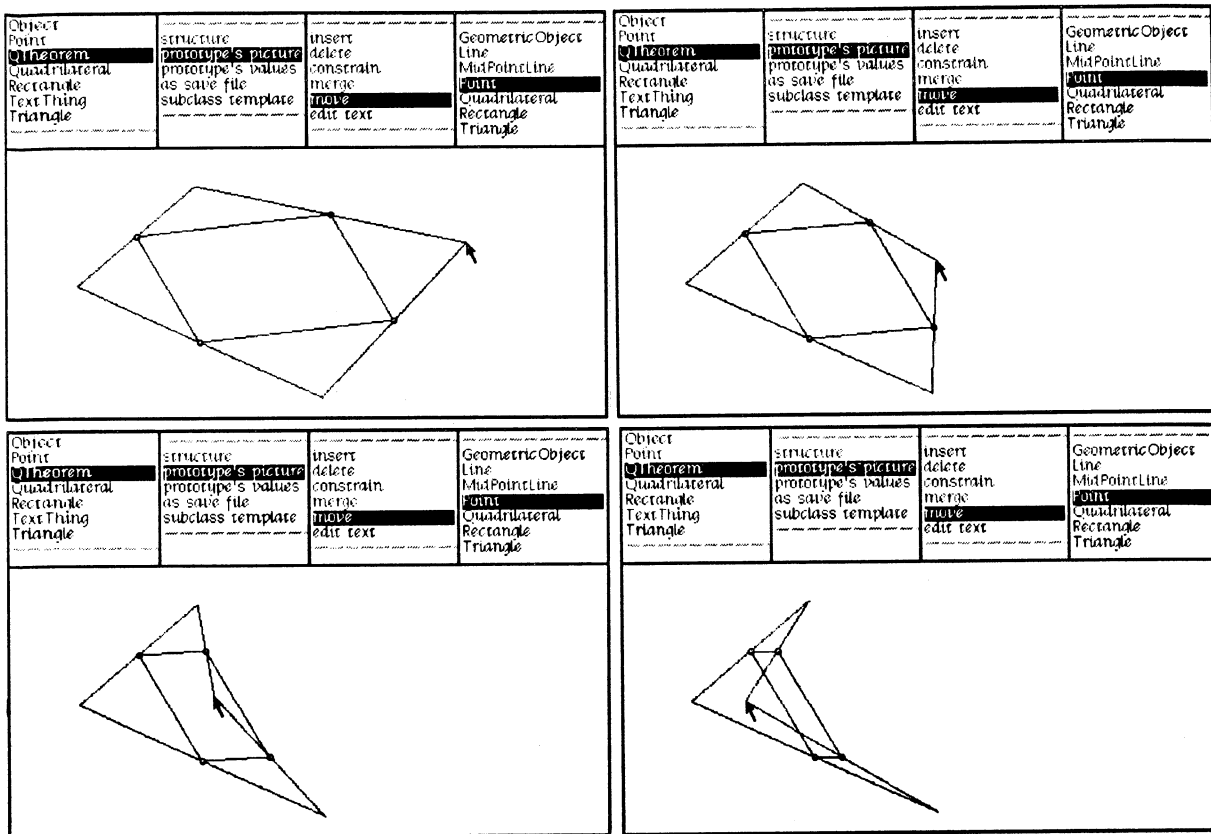


Figure 2.12 - Moving a vertex of the quadrilateral

Constraint Satisfaction

A few comments about the constraint satisfaction process are now in order. 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. A number of techniques for doing this have been built into the system, as will be described in Chapter 5.

In general, the constraints on an object might specify its behavior incompletely or redundantly, or they might be unsatisfiable. QTheorem, for example, is 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 this information by anchoring the midpoints, as in Figure 2.13. [The anchor symbol indicates a constraint that the anchored point may not be moved during constraint satisfaction.]

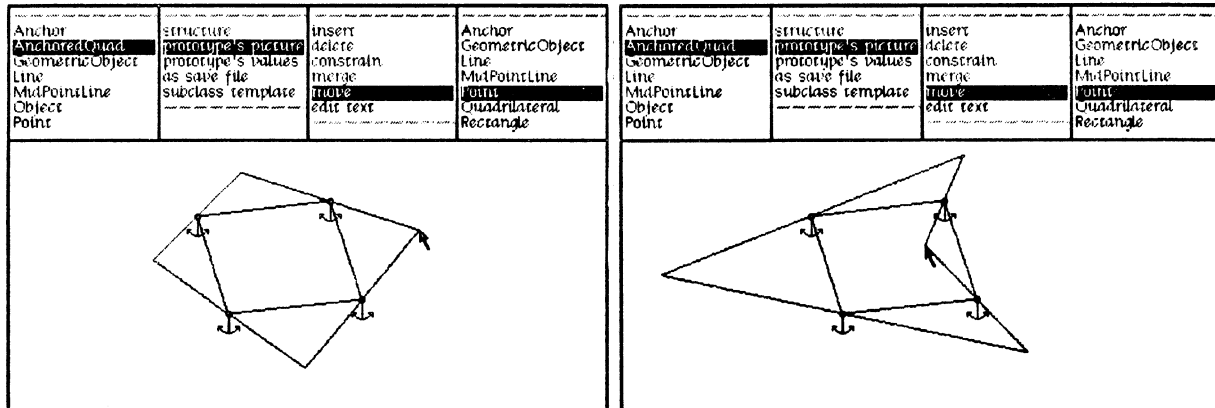


Figure 2.13 - A quadrilateral with anchored midpoints

Second Example - Constructing a Graphical Calculator

In this second example, we will construct some graphical programs for a simulated calculator. It is interesting to note that the use of ThingLab for this application was not anticipated -- all the classes used here were designed only after the system had been running for some time. However, there *is* a strong resemblance between the calculator parts and electrical circuit parts, and ThingLab was designed to be able to simulate simple electrical circuits.

Some useful classes for building calculator programs have already been designed by an experienced user. 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 graphical 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 together parts of the calculator.

Using these building blocks, classes that represent the various arithmetic operations have been defined. First, there is a general class `NumberOperator`. The parts of a `NumberOperator` are a frame for displaying the operator's symbol, and three number leads that terminate on the edges of the frame. The frame and the nodes at the ends of the three number leads are all designated as *attachers*. Four subclasses of `NumberOperator` are defined, namely `Plus`, `Minus`, `Times`, and `Divide`.

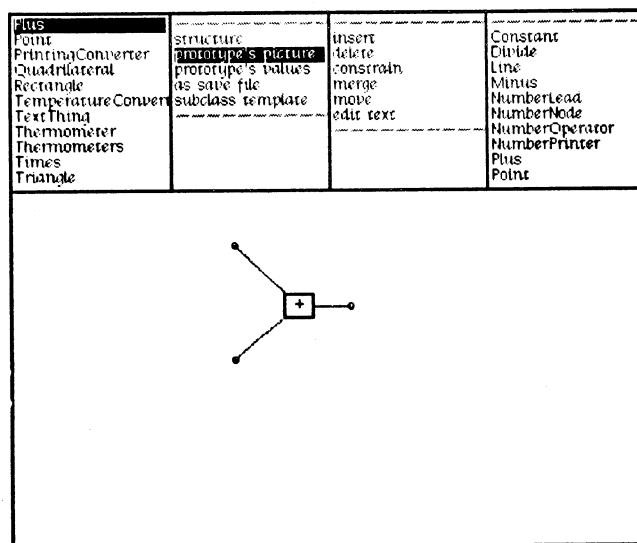


Figure 2.14 - Picture of the prototype for Plus

Plus, for example, has three number leads with number nodes at the ends, which are inherited from NumberOperator. It has an added constraint that the number at the node on the right always be the sum of the numbers of 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 will be updated; if the text is edited, the node's number will be changed correspondingly. A special kind of NumberPrinter is a Constant. For constants, the constraint is one way. The text may be edited, thus changing the number; but the number may not be changed to alter the text.

Constructing a Centigrade to Fahrenheit Converter

Using these parts, let's construct a Centigrade to Fahrenheit converter. After creating a new class TemperatureConverter, we choose the *prototype's picture* format for viewing and editing it. Next, we select the word *insert* in the message pane, and the word *Times* in the tool pane. 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. Following this, we insert a Plus operator in the same manner, connecting one of the nodes on its left to the node of the times operator on the right. [The author has avoided the terms input and output nodes in describing these operators, since, as shall be seen, information flow is not restricted in this way.] 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 2.15. [Incidentally, note the use of generic editing messages: the same *insert* message is used to insert lines, arithmetic operators, and resistors; the *edit text* message is used both to edit the value of a constant and to edit a paragraph in a document.]

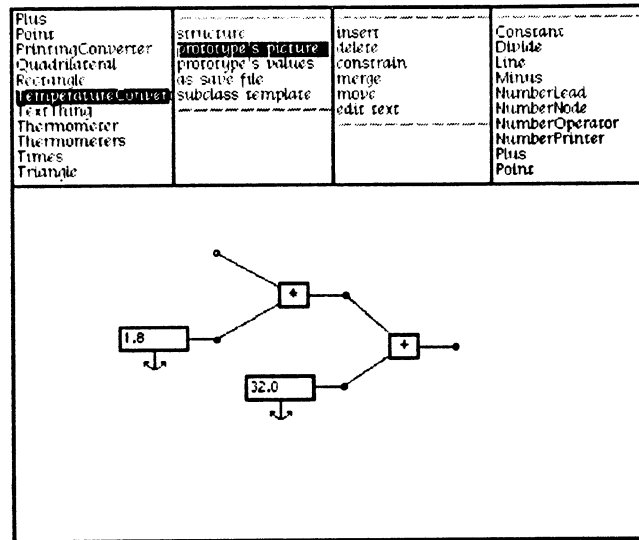


Figure 2.15 - Picture of the completed Centigrade to Fahrenheit Converter

Once the converter has been defined, we may use it as a part of other objects (i.e., as a subroutine). As an example of this, we define a new class PrintingConverter. We add an instance of TemperatureConverter as a part, and also two instances of NumberPrinter to display the Centigrade and Fahrenheit temperatures (Figure 2.16). If we edit the Centigrade temperature, the PrintingConverter will satisfy its constraints by updating the numbers at its nodes, and the Fahrenheit temperature displayed in the frame on the right.

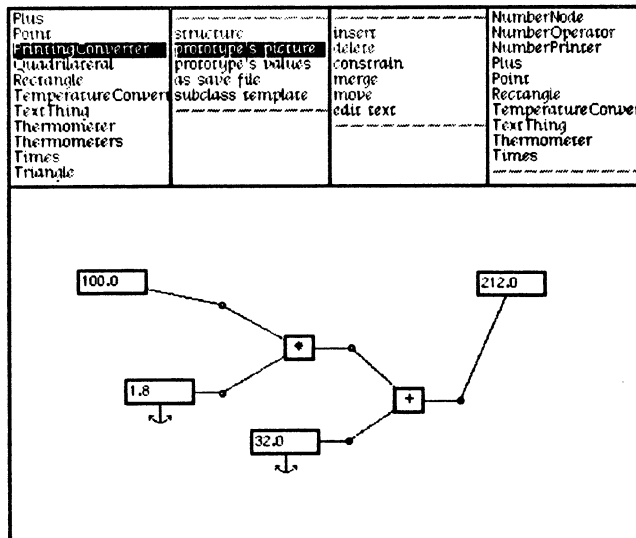


Figure 2.16 - A PrintingConverter

However, because of the multi-way nature of the constraints, the device works backwards as well as forwards! Thus, we can edit the Fahrenheit temperature, and the Centigrade temperature will be updated correspondingly (Figure 2.17). 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.

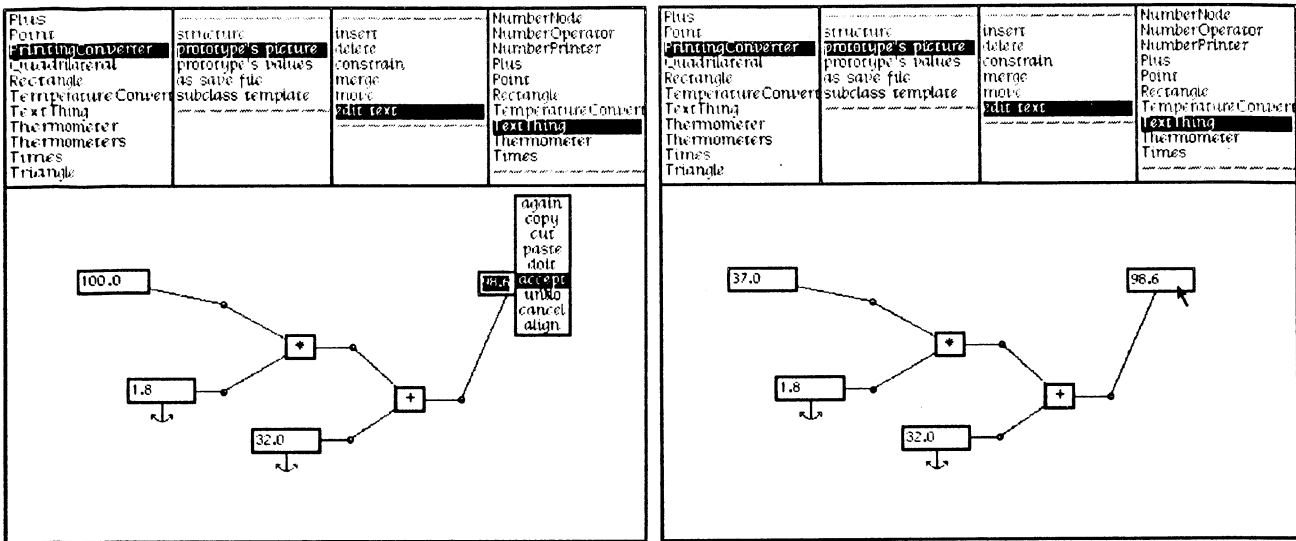


Figure 2.17 - Editing the Fahrenheit temperature

We may also connect the converter to other types of input-output devices, for example, a simulated thermometer. We find an experienced ThingLab user, and ask her to define a Thermometer class for us. A thermometer has a number lead for connecting it with other devices, and a constraint that the height of the mercury be proportional to the number at its node. We then build another class in analogy with PrintingConverter, and again use an instance of TemperatureConverter as one of its parts. This time, however, we hook up the converter to instances of class Thermometer. When the construction is complete, 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 2.18).

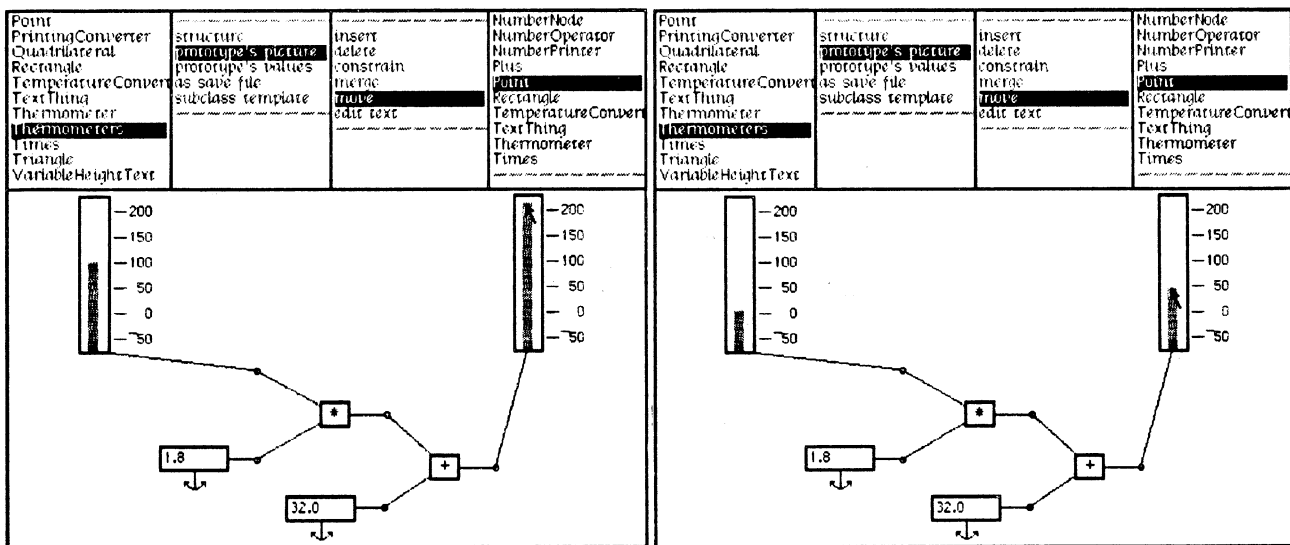


Figure 2.18 - The temperature converter with thermometers for input and output

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 2.19.

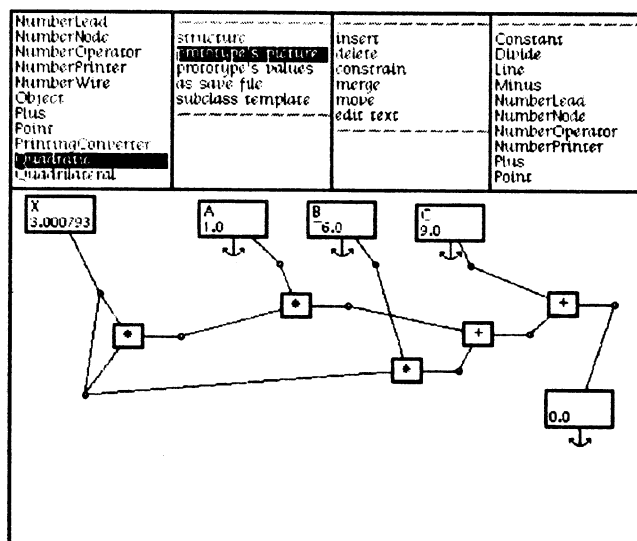


Figure 2.19 - A quadratic equation network

When we edit any of the constants, the value in the frame on the left will change 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. Unlike the temperature converter examples, in this case the system was unable to find a one-pass ordering for solving the constraints, and has resorted to the relaxation method. [Note that the graph structure has loops in it.] Relaxation will converge to one of the two roots of the equation, depending on the initial value of x .

Now let's 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 embodying it as a constraint. We find an expert user, who constructs a class QuadraticSolver for us (Figure 2.20). The parts of an instance of QuadraticSolver include four NumberNodes a , b , c , and x , and a constraint that $x = (-b + (b^2 - 4ac)^{1/2}) / 2a$. [Since the class NumberNode doesn't 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.]

NumberPrinter NumberWire Object Plus Point PrintingConverter Quadratic QuadraticSolver Quadrilateral Rectangle SolvedQuadratic	structure prototype's picture prototype's values as save file subclass template		
<p>Class new title: 'QuadraticSolver' subclassof: Object fields: 'X A B C text' declare: "]</p> <p>QuadraticSolver prototype parts: 'X A B C text'. QuadraticSolver prototype field: ↗ X replaceWith: NumberNode prototype recopy. QuadraticSolver prototype field: ↗ A replaceWith: NumberNode prototype recopy. QuadraticSolver prototype field: ↗ B replaceWith: NumberNode prototype recopy. QuadraticSolver prototype field: ↗ C replaceWith: NumberNode prototype recopy. QuadraticSolver prototype field: ↗ text replaceWith: TextThing prototype recopy.</p> <p>QuadraticSolver understands: 'show picture: window origin [super show picture: window. text frame show picture: window. origin ← text frame origin. window as Turtle place: X point; goto: origin+(40@0); place: A point; goto: origin+(90@0);</p>			

Figure 2.20 - Constructing the class QuadraticSolver

We then insert an instance of QuadraticSolver into the network, merging its number nodes with the appropriate existing nodes in the network (Figure 2.21). Now, the system can find a simple one-pass ordering for satisfying the constraints, and doesn't need to use relaxation.

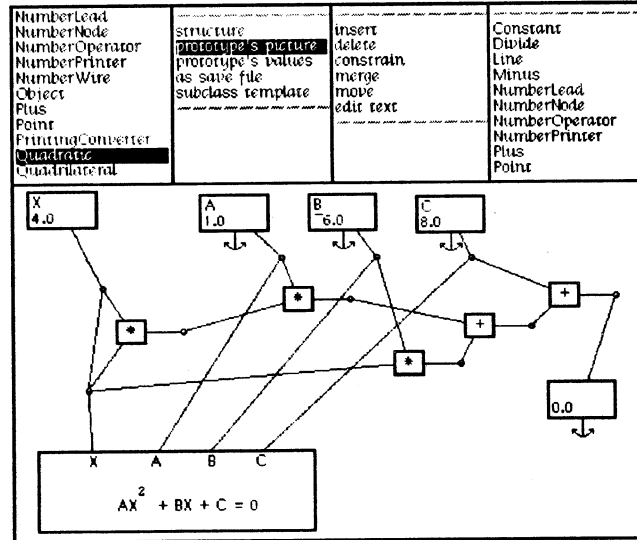


Figure 2.21 - The network after adding an instance of QuadraticSolver

In inserting an instance of QuadraticSolver to 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 doesn't add any new information. However, QuadraticSolver's constraint is computationally better suited to finding the value of x . This technique of introducing multiple redundant constraints on an object is an important way of dealing with circularity.

Beyond this thesis, one of directions of this research has been toward a complete programming language organized around constraints. Such a language is proposed in Chapter 6. This example gives some interesting glimpses of what such a language might be like. QuadraticSolver could have been described by constructing a network, as with the original class Quadratic, rather than having a specially-defined constraint. However, if one tries sketching the network for QuadraticSolver, one will find that it is not very illuminating, and also takes up much more space than the algebraic form. Even with the original network, the diagram is much harder to understand than the written equation for people who know algebra. It would be nice, therefore, to have a language in which, for example, the algebraic constraint $ax^2 + bx + c = 0$ had the same *semantics* as the diagram.

More Examples

A Document with Constraints

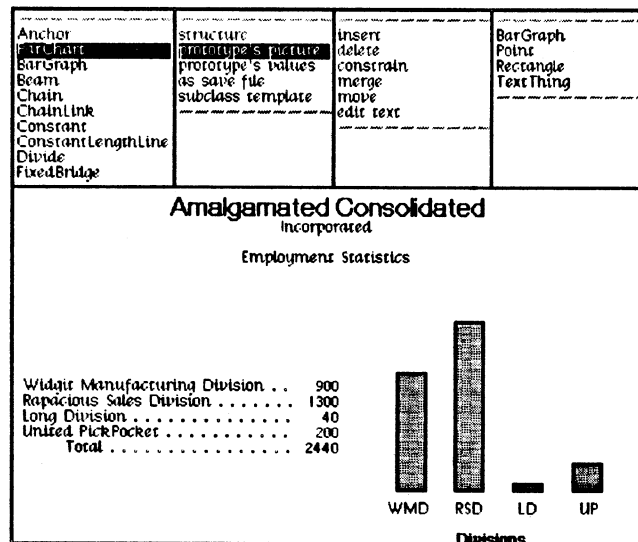


Figure 2.22 - A document with constraints

The example shown in Figure 2.22 demonstrates the use of constraints in describing a dynamic document. The document consists of a number of paragraphs of text, and four instances of BarGraph. Each instance of BarGraph has a constraint that relates the height of the bar to the number displayed in a text field. In addition, the document as a whole has a constraint that the sum of the numbers of employees in each division be equal to the total at the bottom.

We can edit one of the fields containing the number of employees in a division. When we change the number of employees in United PickPocket from 200 to 800 and issue the *accept* command, the sum will be updated, and the height of the corresponding bar will change appropriately to satisfy all the constraints (Figure 2.23).

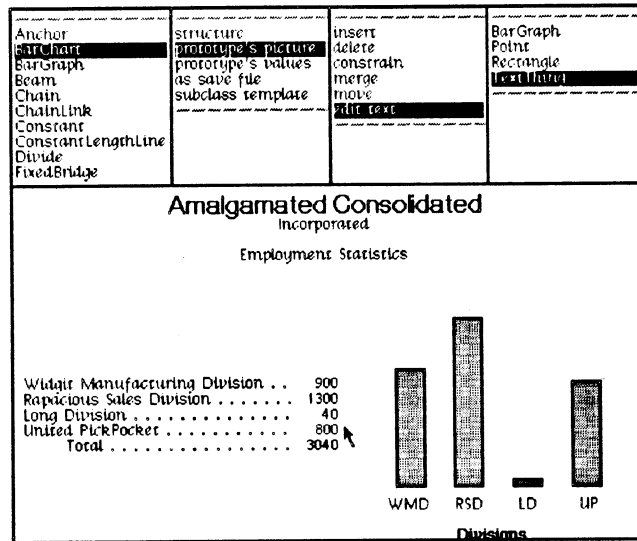


Figure 2.23 - The document after editing the number of employees in United PickPocket

Also, we can pick up the top of one of the bars with the cursor. As we move it up or down, the corresponding number and the total will change (Figure 2.24).

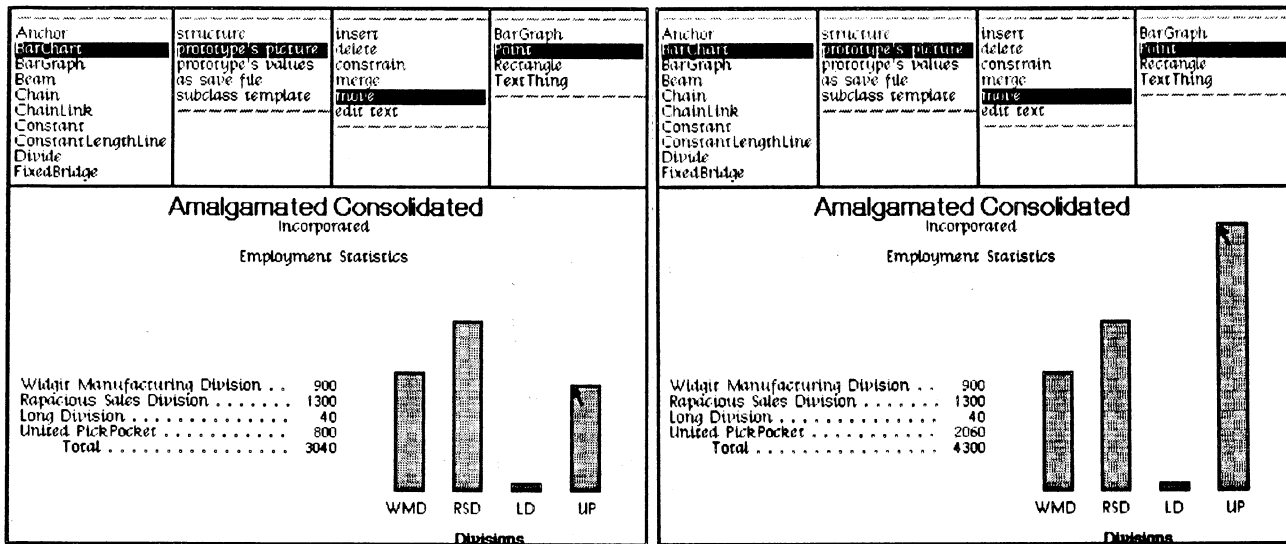


Figure 2.24 - Moving the top of a bar

Layout Constraints

The previous example demonstrated the use of constraints on the contents of a document; Figure 2.25 shows an example of a constraint on layout.

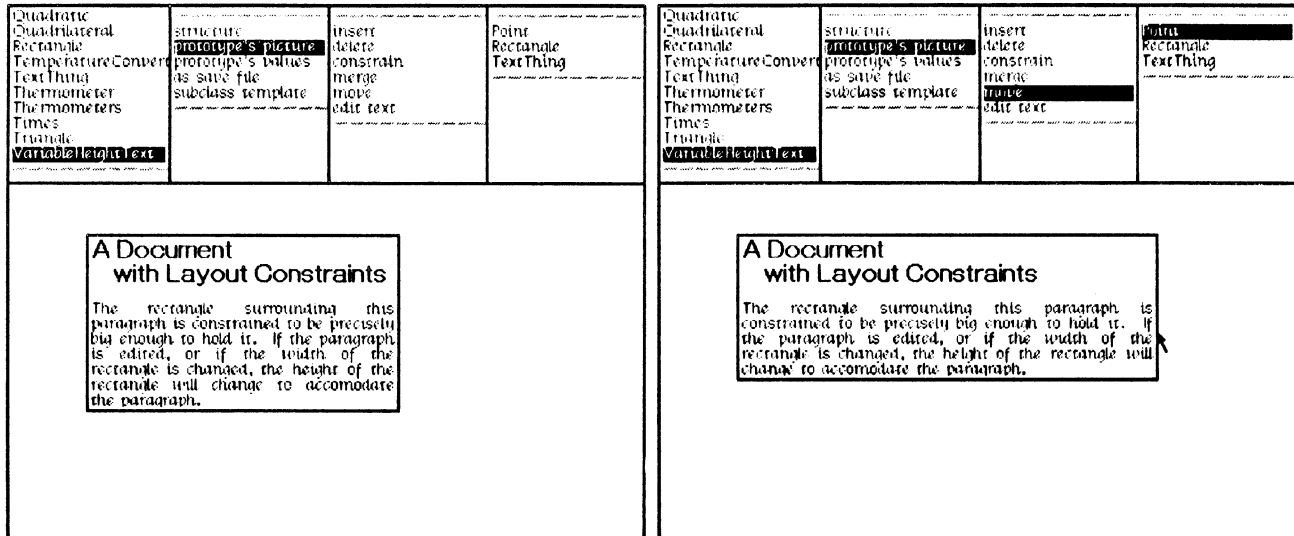


Figure 2.25 - A document with layout constraints

The text object has a constraint that the height of the rectangle be such that the rectangle is precisely big enough to hold the paragraph. If we change the width of the rectangle or edit the paragraph, the rectangle's height will adjust itself accordingly.

A Paned Window

The example in Figure 2.26 illustrates how the shape of a paned window, such as the one used in the ThingLab user interface, can be specified by constraints. The basic building block is the class Rectangle. Using Rectangle, an experienced user has defined two new building blocks, namely a class LeftRight and a class UpDown. The parts of an instance of LeftRight are a pair of rectangles constrained to be side-by-side and of the same height; UpDown is defined analogously. Using these classes in turn, instances of LeftRight and UpDown may be connected together by merging the appropriate corners to form a paned window.

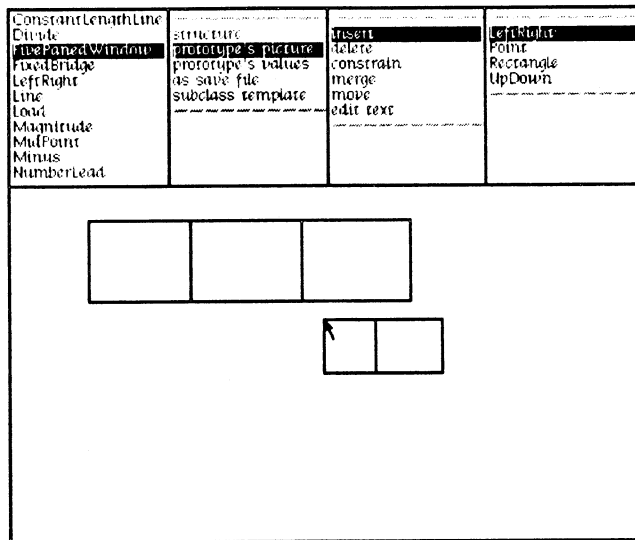


Figure 2.26 - Building a paned window

To change a corner of one of the panes, we may select *move Point*; to move a pane as a unit, we may select *move Rectangle*. As we move the part, the other parts of the paned window will adjust themselves to satisfy all their constraints (Figure 2.27).

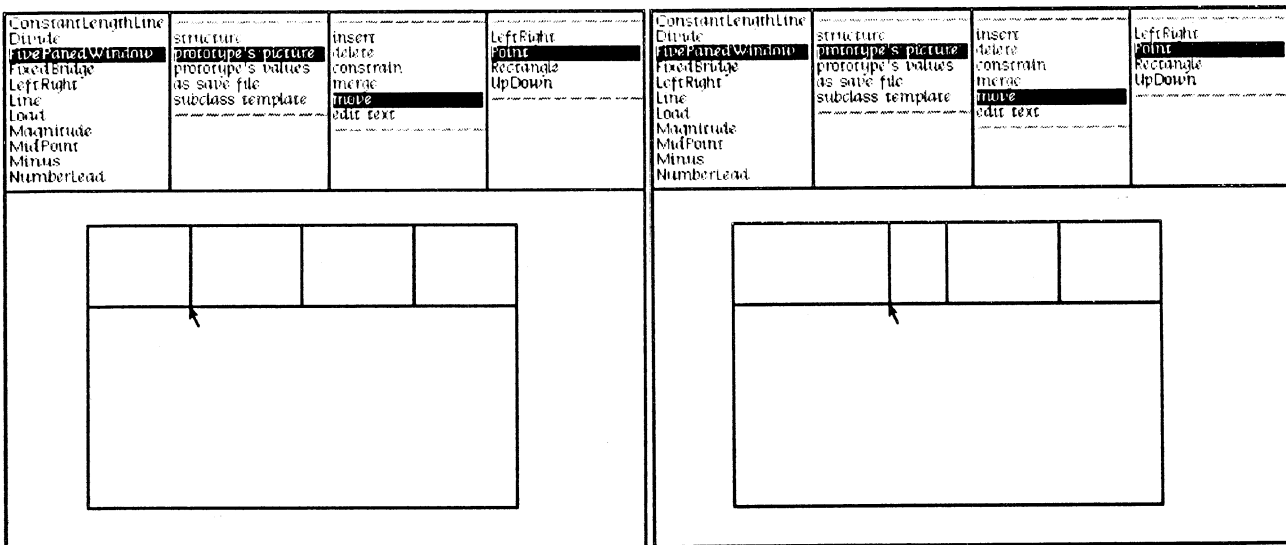


Figure 2.27 - Moving the corner of a pane in a paned window

Alternate Views of a Triangle

This example demonstrates some more things that can be done with multiple views in ThingLab. In Figures 2.28 and 2.29, a point is used to represent an reference to an object. Various views can be connected to the object; as the object changes, the views are updated correspondingly.

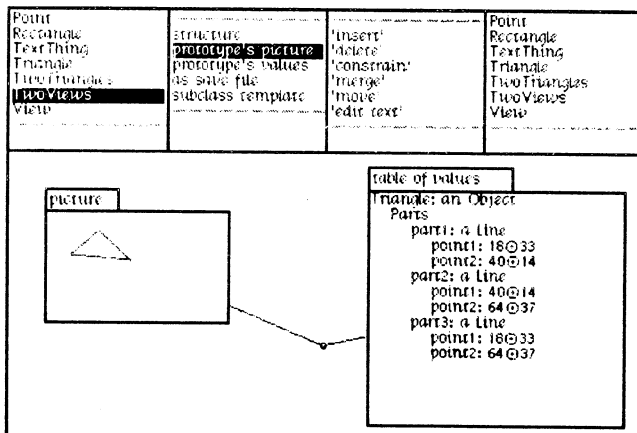


Figure 2.28
Two views of a triangle

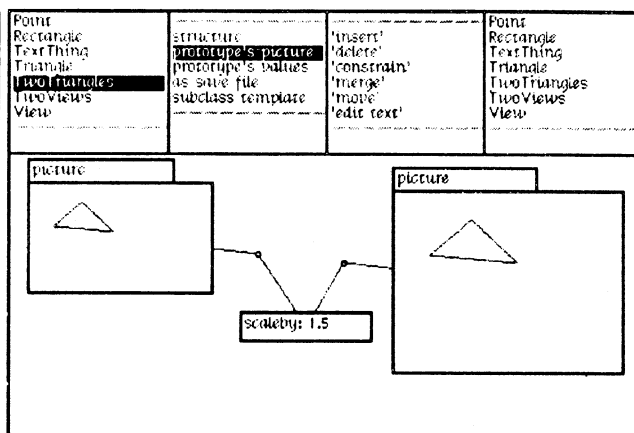


Figure 2.29
Two triangles connected by a scaling constraint

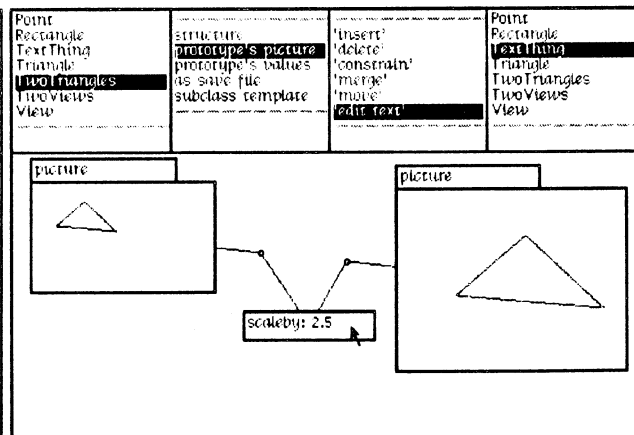
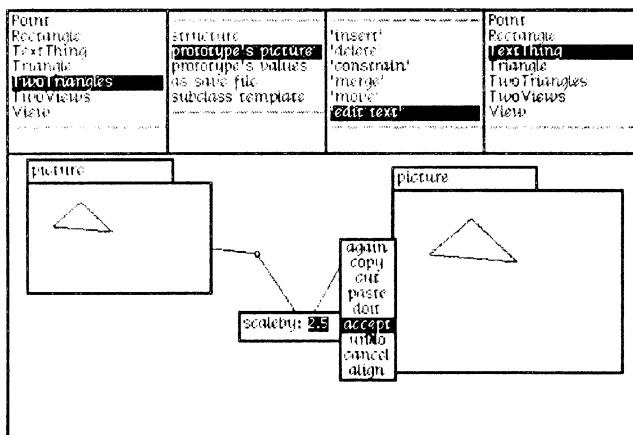


Figure 2.30 - Editing the scaling constraint

In Figure 2.29, two triangles are connected by a constraint that one triangle be 1.5 times as big as the other. If we edit the text in the constraint description, making the scale factor 2.5, the triangle on the right increases in size appropriately.

A Bridge

This example has been taken directly from Sketchpad [Sutherland 1963]. For use in constructing bridge simulations, a class Beam, a class Weight, and a class Anchor have been defined. The beams have a constraint that they obey Hooke's Law. Using these classes, we can construct a simulation of a bridge. When we apply the weight, the beams extend and compress accordingly (Figure 2.31). In this case the constraint satisfier couldn't find a one-pass ordering for satisfying these constraints, and so relaxation was used. [Each beam has displayed how much it has lengthened or shortened. A positive number indicates an extension; a negative number indicates a compression.]

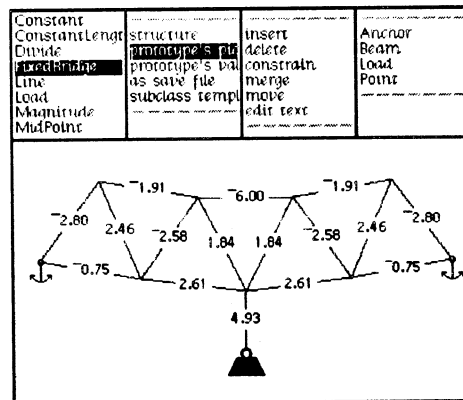


Figure 2.31 - A bridge under load

An Electrical Circuit

A simulation of a simple electrical circuit will now be presented. To illustrate a typical set of building blocks for a given domain, the basic classes used in constructing the circuit will be shown. These classes have been defined by an experienced user of the system. Using these classes, a less sophisticated user could then employ them in constructing a simulation such as that shown here.

The building blocks used in the circuit are resistors, batteries, meters, and wires. Also, there are two other kinds of objects used in connecting together the components of the circuit: nodes and leads. A node is a connection point in the circuit; its parts include a voltage and a graphical screen location. A lead is a terminal of one of the components such as a resistor or a meter, and has as its parts a node and a current. To connect, say, two resistors together, the node from the lead of the first resistor is merged with the node from the lead of the second.

Class *ElectricalNode*

Superclasses

ElectricalObject

Part Descriptions

voltage: a *Voltage*

currents: a *Set*

location: a *Point*

Constraints

currents sum = 0.0

forEach: *current in: currents methods:*

[*current* \leftarrow 0.0 - (*currents excluding: current*) *sum*]

The parts of a node are a voltage, a set of currents flowing into that node, and a screen location. The constraint is Kirchhoff's law: it specifies that the sum of the currents into the node be 0. The information under the constraint's rule is a method for computing any one of the currents, given the values of all the others.

Class *Ground*

Superclasses

ElectricalNode

Constraints

voltage = 0.0

voltage \leftarrow 0.0

A ground is a kind of node whose voltage must be 0.

Class ElectricalLead**Superclasses**

ElectricalObject

Part Descriptions

node: an ElectricalNode

current: a Current

Constraints

node currents has: current

node currents insert: current

A lead represents one of the connecting wires of a component.

Class TwoLeadedObject**Superclasses**

ElectricalObject

Part Descriptions

lead1: an ElectricalLead

lead2: an ElectricalLead

Constraints

$$\text{lead1 node current} + \text{lead2 node current} = 0.0$$

$$\text{lead1 node current} \leftarrow 0.0 - \text{lead2 node current}$$

$$\text{lead2 node current} \leftarrow 0.0 - \text{lead1 node current}$$

This is an abstract superclass used in defining components with two leads. It has a constraint that the current flowing out of one lead be equal and opposite to the current flowing out of the other.

Class Resistor**Superclasses**

TwoLeadedObject

Part Descriptions

resistance: a Resistance

label: a TextThing

Constraints

$$(\text{lead1 node voltage} - \text{lead2 node voltage}) = (\text{lead1 current} * \text{resistance})$$

$$\text{lead1 node voltage} \leftarrow$$

$$\text{lead2 node voltage} + (\text{lead1 current} * \text{resistance})$$

$$\text{lead2 node voltage} \leftarrow$$

$$\text{lead1 node voltage} - (\text{lead1 current} * \text{resistance})$$

$$\text{lead1 current} \leftarrow$$

$$(\text{lead1 node voltage} - \text{lead2 node voltage}) / \text{resistance}$$

resistance reference

resistance = label text asFloat

resistance \leftarrow label text asFloatlabel text \leftarrow resistance asString asParagraph

In addition to the constraint inherited from `TwoLeadedObject`, a resistor has an Ohm's law constraint, and a constraint that its resistance correspond to the text in its label. In the Ohm's law constraint, *resistance* has been designated as reference only, so that the system will have to satisfy the constraints by changing the voltages and currents in the circuit, rather than by changing the values of the components.

Class Battery

Superclasses

`TwoLeadedObject`

Part Descriptions

`internalVoltage`: a `Voltage`

`label`: a `TextThing`

Constraints

`lead1 node voltage = (lead2 node voltage + internalVoltage)`

`lead1 node voltage ← lead2 node voltage + internalVoltage`

`lead2 node voltage ← lead1 node voltage - internalVoltage`

`internalVoltage` reference

`internalVoltage = label text asFloat`

`internalVoltage ← label text asFloat`

`label text ← internalVoltage asString asParagraph`

Class Wire

Superclasses

`TwoLeadedObject`

Constraints

`lead1 node voltage = lead2 node voltage`

`lead1 node voltage ← lead2 node voltage`

`lead2 node voltage ← lead1 node voltage`

A wire is itself a kind of two leaded object. Its constraint specifies that the wire be a perfect conductor. [If it were important to represent the resistance of a real wire, this could be done using an instance of `Resistor` instead.]

Class Ammeter

Superclasses

`TwoLeadedObject`

Part Descriptions

`reading`: a `TextThing`

Constraints

`lead1 node voltage = lead2 node voltage`

`lead1 node voltage ← lead2 node voltage`

`lead2 node voltage ← lead1 node voltage`

`reading text = lead1 current asText`

`reading text ← lead1 current asText`

`lead1 current` reference

Class Voltmeter**Superclasses**

TwoLeadedObject

Part Descriptions

reading: a TextThing

Constraints

lead1 current = 0.0

lead1 current \leftarrow 0.0

reading text = (lead1 node voltage - lead2 node voltage) asText

reading text \leftarrow (lead1 node voltage - lead2 node voltage) asText

lead1 node voltage reference

lead2 node voltage reference

Both the class Ammeter and the class Voltmeter describe perfect meters. An instance of Ammeter has no voltage drop across it; an instance of Voltmeter draws no current.

The pictures for these building blocks were defined by writing an appropriate Smalltalk method. For example, the picture of a node is a dot if the number of currents in its set is three or more, and otherwise nothing; the picture of a resistor is the usual symbol, along with an editable label indicating its resistance.

Using the components, we may construct a voltage divider (Figure 2.32).

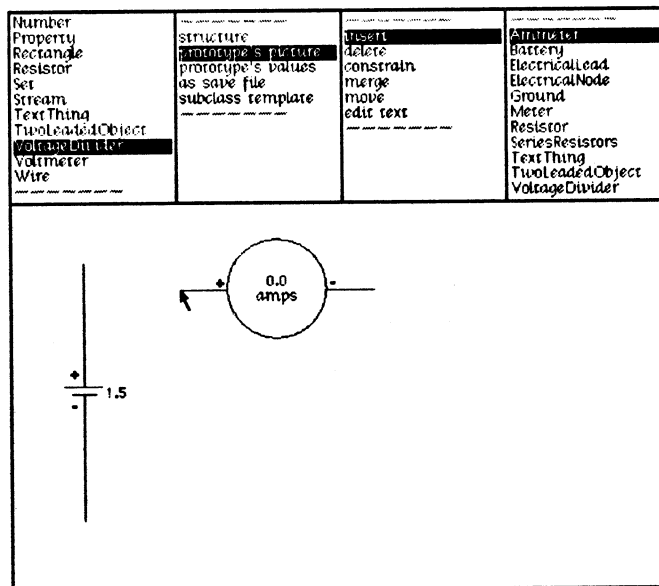


Figure 2.32 - Building a voltage divider

The nodes at the ends of the leads of the components are designated as attachers by the class TwoLeadedObject. Thus, when inserting the ammeter, the node at the end of each lead will try to merge with an existing node in the circuit.

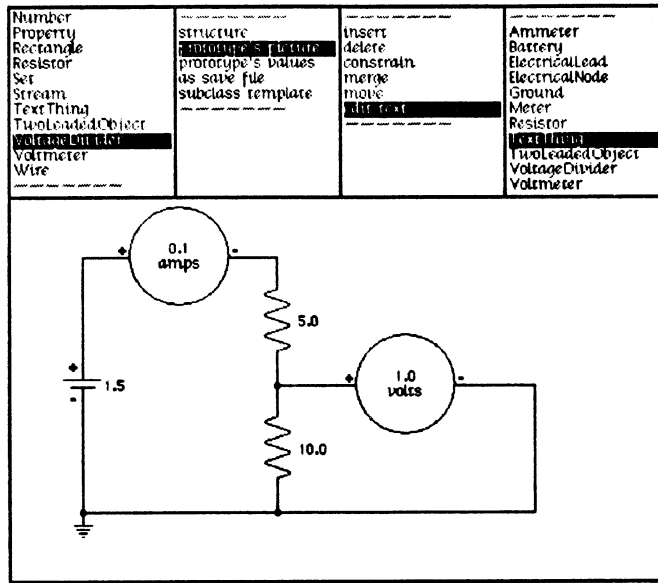


Figure 2.33 - The completed voltage divider

After the circuit has been completed (Figure 2.33), we may change its parameters and observe the results (Figure 2.34).

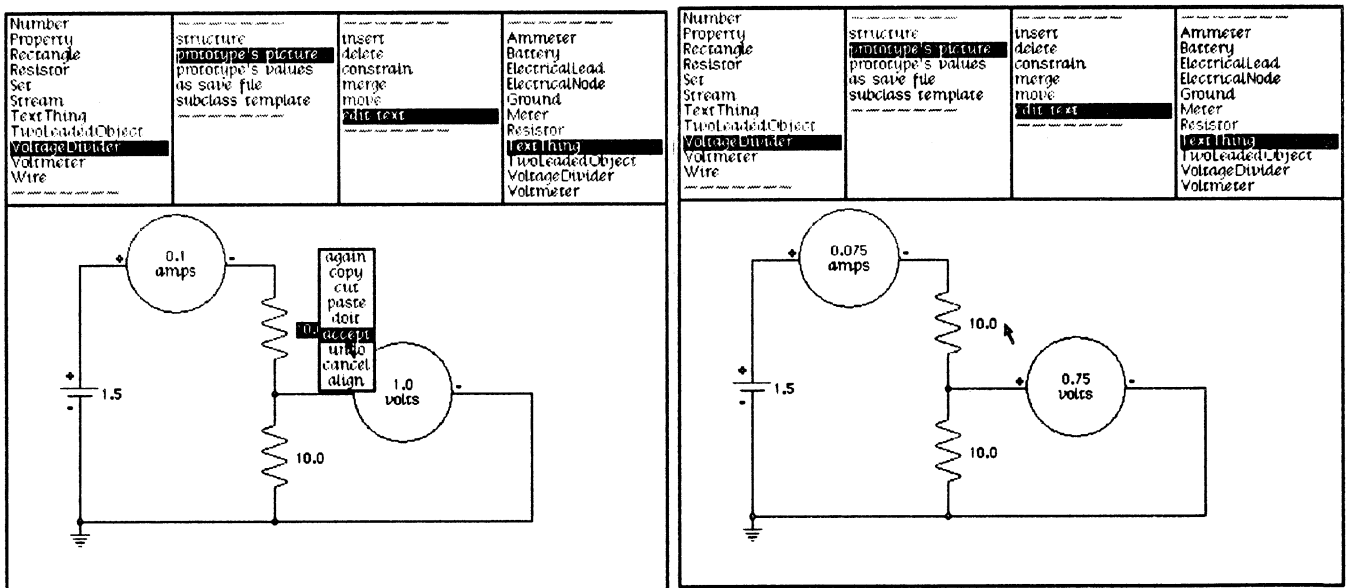


Figure 2.34 - Changing a resistance in the voltage divider

Again, in this case, the system was unable to satisfy the constraints in one pass, and has used relaxation. A discussion of constraint satisfaction for this example, and another example of the use of multiple views to eliminate the need for relaxation, may be found in Chapter 5.

Chapter 3 - Objects

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 will sometimes be referred to as the *owner* of its parts. For example, the above line owns its endpoints.

A primitive object has no parts of its own. In ThingLab, integers, real numbers, and text are taken to be primitive.

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:

name - an identifier.

constraints - the set of constraints that apply to the corresponding part of each instance.

merges - the set of merges that apply to the corresponding part of each instance.

class - the class of the corresponding part of each instance. This is more restrictive than Smalltalk, in which the class of the contents of an instance field is not declared.

Imposing this restriction makes the job of constraint satisfaction easier [see Chapter 5].

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 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.

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

A subclass of PartDescription is SuperclassDescription. Instances of this subclass are used to describe parts that represent superclasses. [See the section on *Inheritance* later in this chapter for more details.]

Insides and Outsides

One of the important features of Smalltalk is its sharp distinction between the inside and the outside of an object. The internal aspects of an object are its class, and its instance fields and their contents; its 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. [This is also related to work on data abstraction mechanisms in languages such as CLU and Alphard (Liskov *et al.* 1977; Wulf, London & Shaw 1976.)]

In ThingLab, the notion of having a part has implications for both the internal and external aspects of the object that owns the part. In the current implementation, 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 Chapter 6, is an example of the use of this separation. Such a part has all the external manifestations of a part, i.e., messages to read and write it. Internally, however, there is no corresponding field; rather, the part is 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.]

Paths

A *path* is a ThingLab object that represents a symbolic reference to a subpart. Each path consists of a list of part names that indicate a way to get from some object to one of its subparts. The path itself doesn't 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. [To get to this value, the path first sends the message *point1* to the line, and then sends the message *x* to the result.] A *full path* is a path that starts from the global symbol table, while a *relative path* is one that starts from some intermediate owner (not necessarily the global symbol table). In the latter case, the starting point is provided by the context in which the path

is used; see Chapters 4 and 5 for examples.

While the definition of a path is simple, the idea behind it has proven quite powerful. 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.

The constraint satisfaction techniques to be described in Chapter 5 all depend on noticing when one constraint interferes with 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 0 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 interfere, the system checks if any of their paths overlap.

External References

The class `PartDescription` is actually a subclass of `FieldDescription`, which is an abstract class intended to capture the notion of a description of instance fields in general. Two other

subclasses of `FieldDescription` have been defined for use in describing external references, that is, references from an object to something other than one of its own parts. External references come in two flavors: absolute and relative. The behavior of these two kinds of references is different with respect to copying. If a copy of an object is made and used as a part of another object, all absolute references will remain unchanged in the copy. On the other hand, all relative references will be changed to refer to the corresponding part of the new owner of the copy. For example, an absolute reference might be used to represent a footnote in a document *A* that refers to a section in another document *B*. If *A* is copied, the footnote in the copy will still refer to the same section in *B*. On the other hand, a relative reference should be used to represent a reference from one section of *A* to another section of the same document. If *A* were copied, the relative reference would indicate the corresponding section of the copy.

For pragmatic reasons, external references are represented as full paths, rather than direct pointers. This avoids circular structures, and makes it easy to find all the constraints that affect the referenced object.

Inheritance

As described in Chapter 1, 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 will be instances of `SuperclassDescription`, rather than `PartDescription`. These parts may have constraints and merges applied to them in the usual way. The only difference between these instances of superclasses and ordinary parts is that messages will be forwarded to them automatically (see below). [The actual implementation is a bit more arcane - see the section *Implementation of Superclasses*. However, the effect is as described, and the reader should think of it in this way.]

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 out for overlap between its parts due to merges, and must also keep all its constraints satisfied. [See Chapter 5 for an example.]

Message Behavior

Inheritance is relatively simple for a class with only one superclass. However, the situation is more complex when multiple superclasses are present: how should the system act in the presence of conflicting inherited information? To help choose among conflicting information, one of the superclasses of a class must be designated as its *primary superclass*.

When an object receives a message, it first checks the message dictionary of its own class. If a corresponding method is found, that method is used. If not, it checks the dictionary of its primary superclass, and then the primary superclass of that superclass, and so on. If the message is not understood by any of the primary superclasses, the object then checks all of its other superclasses to see if any of them understand the message. If there is no method, or if there are several conflicting inherited methods, the user is notified. If there is a single method defined for that message, then that method is used. 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 send it directly to the appropriate superclass part. If the message protocol of a class is changed, its subclasses should be notified so that they can delete obsolete message forwarders; this is not done in the current implementation.

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 class or one of its primary superclasses. An example of the use of this technique is in the default method for showing an object's picture. This default method (defined in class Object) yields a picture that combines all inherited pictures and the pictures of the object's own parts. It simply specifies that each of the object's parts should be asked to show itself (irrespective of whether or not the parts represent multiple superclasses).

In previous versions of ThingLab, the problem of conflicting inherited methods was handled in the following way. Methods were themselves objects that understood a variety of messages. If several conflicting methods were found for a given message, the inherited methods were asked to negotiate among themselves to come up with the method to be used to receive the message. These negotiations were relatively unsophisticated. If the methods were all the same, then any one of them would be used. Some methods, called default methods, would always defer to others, while other methods, such as those for depicting an object, would make a new method that invoked all the inherited methods. If negotiation failed, the user would be asked to supply the desired method. Once a method had been constructed or entered, it was saved in case it was required again. This technique worked reasonably well; the only reason that it is not in the current system is that the author never got around to re-implementing it.

As an example of the use of multiple superclasses, suppose one has available the class Triangle, and two subclasses of it: a class IsoscelesTriangle; and another class HorizontalTriangle, the class of triangles with horizontal bases.

Class Triangle

Superclasses

GeometricObject

Part Descriptions

side1: a Line

side2: a Line

side3: a Line

Merges

side1 point1 \equiv side3 point1

side1 point2 \equiv side2 point1

side2 point2 \equiv side3 point2

Class IsoscelesTriangle

Superclasses

Triangle

Constraints

side1 length = side2 length

Class HorizontalTriangle

Superclasses

Triangle

Constraints

side3 point1 y = side3 point2 y

Given the above classes of triangles, suppose one wishes to define a new class of isosceles horizontal triangles. This can be done as follows:

Class IsoscelesHorizontalTriangle

Superclasses

T: Triangle (primary superclass)

IT: IsoscelesTriangle

HT: HorizontalTriangle

Merges

T \equiv IT \equiv HT

If a different correspondence between the parts of the superclasses is desired, it can be represented by an expanded set of merges, e.g.:

Merges

T side1 point1 \equiv IT side1 point1 \equiv HT side2 point1
 T side2 point1 \equiv IT side2 point1 \equiv HT side3 point2
 T side3 point2 \equiv IT side3 point2 \equiv HT side1 point1

The use of this scheme yields a reasonable structure and behavior for the new class. The new class has all the constraints and merges of each of its superclasses. The inherited parts are all available in the new class; overlaps are represented by merges. The use of a primary superclass need not introduce an unwanted asymmetry: rather than designating either IsoscelesTriangle or HorizontalTriangle as the primary superclass, Triangle itself was so used. Note that having several inherited parts with the same name is not a problem, as one can always indicate which part is meant by prefixing the message with the name of the appropriate superclass part.

Semantically, the close relationship in this scheme between the class-subclass relation and the part-whole relation is interesting. While the scheme is fairly simple, it was arrived at only after much discussion, head-scratching, and implementation of silly ideas. Also, given the author's past experience, it probably has some as yet undiscovered deficiencies.

Use of Multiple Superclasses for Multiple Representation

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**

G: GeometricObject (primary superclass)

C: CartesianPoint

P: PolarPoint

Constraints

C = P asCartesian

C ← P asCartesian

P ← C asPolar

The constraint makes use of two auxiliary messages to CartesianPoint and PolarPoint.

Class PolarPoint**Methods****asCartesian**

[↑CartesianPoint new x: r* theta cos y: r*theta sin]

Class CartesianPoint**Methods****asPolar | dist angle**

[dist ← ((x*x) + (y*y)) sqrt.

angle ← [x≠0.0⇒[(y/x) arctan]

y=0.0⇒[0.0] y<0.0⇒[-pi/2.0] pi/2.0].

↑PolarPoint new r: dist theta: angle]

These methods in turn make use of the messages *sin*, *cos*, *arctan*, and *sqrt* to real numbers (methods not listed).

Implementation of Multiple Superclasses

For pragmatic reasons, multiple superclasses have been implemented so as to take advantage of the efficient Smalltalk subclassing mechanism. Superclasses other than the primary one are implemented as described. However, the primary superclass is represented as a standard Smalltalk superclass. Thus, for this superclass, the structure has been flattened. Rather than having an instance of its primary superclass as a part, the subclass has the parts of its primary superclass as its own parts. For example, the instance fields of the class *IsoscelesHorizontalTriangle* would actually be as follows:

field 1 - a Line (the side1 from Triangle)

field 2 - a Line (the side2 from Triangle)

field 3 - a Line (the side3 from Triangle)

field 4 - an IsoscelesTriangle

field 5 - a HorizontalTriangle

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 doesn't specify otherwise, the prototype has nil in each of its instance fields. However, if the user has defined the class by example, the prototype will hold 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 graphical editing, since every object needs *some* appearance.

More importantly, 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 will accept read-only or value-altering messages, but only prototypes will accept structure-altering messages. This is because 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.

On occasion, the user might want to alter the structure of an instance that is not a prototype. To do this, the user should first send it the message *asPrototype*, and then send the structure-altering message to the result. When an object receives the message *asPrototype*, if it isn't already a prototype, it makes a new subclass of its current class, moves its instance values into the fields of the prototype for this subclass, and returns that prototype.

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. One can also use a combination of definition by example and hand coding.

In general, there are many possible classes that could be abstracted from a given example; which one *should* be abstracted will depend on the user's purposes. The ThingLab facility for definition by example provides a reasonable default, but 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.

This facility could, however, be generalized to allow several kinds of choices as to how the class definition should be abstracted. First, the prototype could decide whether some aspect of itself, e.g. a constraint, should be a property of it alone, or of its class in general. This kind of choice does not arise in the current representation, since e.g. constraints can only be associated with classes and not particular instances; but if the representation were extended such decisions would need to be made. Also, the prototype could *recognize* some configuration of its parts as being an instance of an already defined class. In the current system, this is done only for merges -- in the user interface, if the user positions one object near another, an instance of a merge will be constructed automatically. However, the system could be extended so that various classes were waiting to recognize instances of themselves. For example, the class `HorizontalLine` might notice that the line just drawn by the user was in fact horizontal, or the class `SeriesResistors` might notice that a pair of resistors formed an instance of itself.

Dynamic Updating

If a class is edited, this change will affect all of its instances. After the edit has been made, all these objects should be notified of the change so that they can update themselves if necessary. This problem is not handled completely in the current implementation: presently, only the prototype is notified; the other affected instances first notice the change the next time they reference the class.

Dynamic updating is a knotty problem in general. One such problem arises in connection with the mixture of the generic and the particular. For example, suppose that the prototype for the class `Widget` is edited by the insertion of an additional line. Each other instance of `Widget` should also have a line added. Sometimes `Widget`'s constraints will completely determine the new line's location; but this will not always be the case. If not, what should be the line's location? The particular location used for the prototype may not be appropriate, since the parts of the other instances may be positioned quite differently.

Limitations of the ThingLab Hierarchies

One of the main deficiencies of both the part-whole and inheritance hierarchies as presently implemented is that they both are useful for describing additive sorts of knowledge only; with the exception of overriding inherited message responses, the hierarchies do not have any mechanisms for masking off, modifying, or replacing information. For example, an object

ought to be able to mask off an inherited constraint, or to replace an inherited constraint with another. More generally, there should be a way of defining all sorts of "parasitic" objects, that can be merged with a given object to change it in some specified way. For example, one should be able to define a class `LineDotter`. Any class that was a subclass of `LineDotter` would have all its inherited lines made into dotted lines. Thus, a subclass of `Triangle` and `LineDotter` would have dotted lines for its sides.

A Comparison with KRL

The representation language KRL [Bobrow & Winograd 1977a, Bobrow & Winograd 1977b] is another system that uses an object-oriented representation scheme, and it is interesting to compare the approach used in KRL with that used in `ThingLab` in the areas of overlap. In this section, KRL definitions for the various classes of triangles described above will be presented as a framework for making some comparisons. The intended domain of KRL is much broader than that of `ThingLab`, and there are hooks for many more kinds of features built into it. Also, KRL is much less "automatic" than `ThingLab`. For example, KRL has no constraint mechanism. A programmer who wants some relation to be satisfied must write procedures to satisfy it, and must make sure that the procedures are invoked at the proper times.

In KRL, knowledge is organized around entities called *units*. Each unit has a unique name, and has one or more named *slots* containing descriptions of other units associated with the given unit. Every unit has a distinguished slot named *self*; the descriptions in this slot apply to the unit as a whole. The KRL analog of a class is a unit with the particular values left undefined. Such a unit is called a prototype in KRL -- note that this is not the same as a `ThingLab` prototype. Inheritance relations may be described using the *self* slot. For example, to describe the prototype `EquilateralTriangle` as being a kind of `Triangle` and also a kind of `RegularPolygon`, its *self* slot would include the descriptions "a `Triangle`" and "a `RegularPolygon`". To provide for simple inheritance of slots and attached procedures from one parent, a prototype optionally may be described as a *further specification* of a single other prototype. This is similar to the primary superclass notion in `ThingLab`.

The KRL analog of an instance is a unit that typically has inherited slots that are filled in with specific information, but no additional slots of its own. For example, a line instance would have the description "a `Line`" in its *self* slot, perhaps with specific values for its endpoints, but would have no slots beyond those inherited from `Line`.

An important difference between a KRL prototype and a `ThingLab` or `Smalltalk` class is that a KRL prototype and an instance of it are basically the same kind of object: the only difference built into the language is that one has more information filled in than the other. [Of course, a KRL user may choose to *treat* them differently.] On the other hand, a `Smalltalk` class and an instance of it are fundamentally different: a class has the message protocol and local storage defined by

class Class; while an instance has the message protocol and local storage defined by its own class. For example, suppose one has a KRL prototype for Line that corresponds to the Smalltalk class Line. In KRL, one could find the description of an endpoint of a Line instance (this might be an explicitly stored point); in the same way, one could find the description of an endpoint of the prototype Line (this would be simply "a Point"). In Smalltalk, an instance of class Line will respond to the message *point1* by returning its first endpoint; but class Line itself won't understand this message at all. On the other hand, the Smalltalk class Line understands messages like *howMany* (how many instances of Line are there?), while this would be a somewhat odd thing to ask a KRL prototype, since it is asking for information about the set of all lines rather than the prototypical line.

There seem to be both advantages and disadvantages for each of these representations for generic concepts, but other language features reduce their importance. ThingLab prototypes can represent a typical instance of a class in the same way that a KRL prototype can represent a stereotypical individual. On the other hand, the KRL meta-description facility gives it the power to talk *about* all instances of a prototype, just as a Smalltalk class talks *about* all its instances rather than *being* a typical instance. KRL does have the significant advantage that an instance may be represented as a further description of several other prototypes. This generality is not cheap.

Turning to the example, we might describe a triangle prototype in KRL as follows:

Triangle

```

self: A GeometricObject
side1: A Line with
    point1 = The point1 from a Line thatIs My side3
    point2 = The point1 from a Line thatIs My side2
side2: A Line with
    point1 = The point2 from a Line thatIs My side1
    point2 = The point2 from a Line thatIs My side3
side3: A Line with
    point1 = The point1 from a Line thatIs My side1
    point2 = The point2 from a Line thatIs My side2.
```

The triangle is a kind of geometric object, as indicated by the description in its *self* slot. The part-whole hierarchy used in ThingLab can be represented in KRL using other named slots. Note that the descriptions of the line's endpoints are redundant. Such redundancy is allowed, even encouraged, in KRL. There are also other ways in which the triangle could be described, e.g. by including additional slots for its vertices. The contents of a KRL slot are a set of descriptions of another entity, not necessarily a direct pointer to that entity. This allows partial knowledge about an entity to be represented, which is not generally possible in ThingLab.

Given a Triangle unit, units for IsoscelesTriangle and HorizontalTriangle may be defined.

```

# IsoscelesTriangle†1           1: FurtherSpecified (\Triangle)
  self:  A Triangle
         A ConstrainedObject with
         ConstraintRule = ...

# HorizontalTriangle†1       1: FurtherSpecified (\Triangle)
  self:  A Triangle
         A ConstrainedObject with
         ConstraintRule = ...

```

Since there is no constraint mechanism built into KRL, a unit for ConstrainedObject has been posited. Note that these triangles have been described in terms of two other units (c.f. multiple superclasses). As a result of being declared as further specifications of Triangle, they will automatically inherit slots and procedures from that unit.

Finally, IsoscelesHorizontalTriangle can be described using the two preceding definitions.

```

# IsoscelesHorizontalTriangle
  self:  An IsoscelesTriangle
         A HorizontalTriangle

```

The multiple description facility of KRL provides a convenient way of handling the multiple superclass problem. In this definition, no correspondence between the sides of the IsoscelesTriangle and the HorizontalTriangle has been declared. In this case, the KRL matcher will form this correspondence when the program tries to access one of the sides. However, one could declare the correspondence in advance with suitable additional descriptions. This is in contrast to ThingLab, in which the correspondence is always part of the static instance structure. Again, the flexibility of KRL is expensive, in that a fairly complex reasoning process will be invoked for every access to a part of the horizontal isosceles triangle. In ThingLab, on the other hand, such run-time access is quite cheap.

Other Representations of ThingLab Objects

The current representation scheme is not the only one that has been used in ThingLab. In earlier versions, ThingLab objects were simulated in Smalltalk, rather than being represented directly as Smalltalk objects as at present. In these earlier versions, there was a single Smalltalk class Thing; all ThingLab objects were represented as instances of this class. ThingLab message sending and inheritance were simulated as well. [These versions were written first in Smalltalk-72 and then in Fastalk, neither of which supported subclasses.] The simulated instances were somewhat more

complex than Smalltalk instances, in that each had its own sets of constraints, merges, backpointers to owners, and so on. Also, the results of constraint analysis were held in a data structure that had to be interpreted each time the constraints needed to be satisfied. This approach was quite useful, in that it was easy to try new representation schemes. Its speed was glacial.

With the advent of Smalltalk-76, subclassing and a Smalltalk compiler became available. At that time the author decided to try to use the Smalltalk representation directly (including the use of existing Smalltalk classes such as Integer and Point), and also to compile ordinary Smalltalk methods to hold the results of constraint satisfaction. To make this possible, ways had to be found to move all the extra ThingLab information that was stored in the simulated instances into Smalltalk classes instead. This was accomplished with the use of paths, part descriptions, the elimination of backpointers, and so on. Constraints and merges could be associated with an entire class only, rather than a particular instance. [To do otherwise would require that methods be associated with an instance, which is difficult in the current Smalltalk. This restriction is not always what one would like. For example, in the current system one cannot represent the class Bridge in such a way that the connectivity of a particular bridge is a property of the instance alone. Rather, a separate subclass is needed for each kind of bridge structure.]

One result of all this is that the system is much faster -- once a method for satisfying the constraints has been planned and compiled, the response time is usually as good as if a suitable method had been hand-coded. [However, the time for planning a new method is still slow.] A much more important result is that this switch has forced the restructuring and refinement of the constraint mechanism so that it can deal with Smalltalk directly, making it realistic to consider organizing an entire programming language around constraints. See Chapter 6, *Directions for Future Research*.

Chapter 4 - Constraints

Introduction

This chapter 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. In this chapter, 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.

Organizational note: this chapter discusses constraint *representation*; constraint *satisfaction* is dealt with in Chapter 5. At the end of Chapter 5 are some remarks on constraints and the procedural-declarative controversy.

Message Plans

A message plan is an abstraction of the Smalltalk notion of sending a message. A message plan doesn't 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 0 or more *arguments*. The receiver is normally a pointer to some 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 10 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:

uniqueState - 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 the section *Relations Among*

the Parts of a Constraint.

referenceOnly - true if the action described by the message plan only references its target, rather than alters it.

compileTimeOnly - true if the message plan is used only during constraint satisfaction planning, and not in producing executable code.

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 was 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 owned by the object's class. [This table is used in generating the format pane in the ThingLab user interface.] In some previous implementations of ThingLab, the idea of methods as objects was used in the scheme for deciding how to receive a message when there were conflicting inherited methods, as was described in Chapter 3.

After a ThingLab method has been constructed, it will usually be asked to add itself to some class's 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's method dictionary.

Constraints

As described in Chapter 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 a *rule*, and a set of *methods* for satisfying the constraint. Adding or modifying a constraint is a structural change, so only prototypes will 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 will depend on the other constraints, and the user's preferences as to what should be done if the object is underconstrained (see Chapter 5). [In some cases, rather than generating a call on an already compiled method, the message plan will expand the method in-line.]

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 is a numerical equation, the test will check 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 non-numerical, the rule will be used directly to generate the test; the error will be 0 if the constraint is satisfied, and 1 if it is not. (See the examples below.) If the user wants to override these default methods, he or she can replace them with hand-coded Smalltalk methods.

Examples of Constraints

Consider the structure described by the class HorizontalLine, a subclass of Line.

Class Line

Superclasses

GeometricObject

Part Descriptions

point1: a Point

point2: a Point

Class HorizontalLine

Superclasses

Line

Constraints

point1 y = point2 y

point1 y \leftarrow point2 y

point2 y \leftarrow point1 y

The class HorizontalLine has a constraint that the y values of the endpoints of each of its instances be equal. The constraint has two ways of satisfying itself, as described by the two methods listed under the rule.

For most methods, including those listed above, the user need provide only the body of the method. The method's selector is generated automatically, and a simple parser is used to construct the corresponding message plan. Methods for a test and for an error expression are

also generated by the constraint. All these methods compile code in the class that owns the constraint, in this case `HorizontalLine`.

Class `HorizontalLine`

Methods

```

horiz-point1-y [point1 y ← point2 y]
horiz-point2-y [point2 y ← point1 y]
horiz-test [ $\uparrow$ (line point1 y - line point2 y) abs < self tolerance]
horiz-error [ $\uparrow$ (line point1 y - line point2 y)]

```

The first method is one of the ways of satisfying the constraint that was provided by the user. There is a matching message plan for this method, indicating that it alters `point1 y`, and uniquely determines the state of that subpart. The second method is analogous to the first. The test returns true if the constraint is satisfied. The message *tolerance* that is invoked in the test returns a number; for graphical objects, the default tolerance is 1 unit of resolution on the graphic display. The error expression returns a number whose value is 0 if the constraint is precisely satisfied. [The selectors for all of these methods are generated by the constraint. They all have the prefix *horiz* (supplied by the user) to distinguish them from methods for other constraints that might be applied to the class. In the method bodies, \uparrow means "return a value".]

Another example is the midpoint constraint used in the introductory example in Chapter 1.

Class `MidPointLine`

Superclasses

`GeometricObject`

Part Descriptions

```

line:      a Line
midpoint:  a Point

```

Constraints

```

(line point1 + line point2) / 2 = midpoint
midpoint ← (line point1 + line point2) / 2
line point1 ← line point2 + ((midpoint-line point2)*2)
line point2 ← line point1 + ((midpoint-line point1)*2)

```

There are three methods, one to alter the midpoint to satisfy the constraint, and the other two to alter the line's endpoints. As mentioned previously, the three methods represent alternate ways of satisfying the constraint. The user may want one way 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 will be 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. [It would be better to represent this sort of meta-constraint more explicitly -- see the section on *Meta-Constraints* that follows.]

The following Smalltalk methods are compiled in class `MidPointLine`:

Class MidPointLine**Methods**

```

inMiddle-midpoint [self midpoint← (line point1 + line point2) / 2]
inMiddle-line-point1 [line point1←
    line point2 + ((midpoint-line point2)*2)]
inMiddle-line-point2 [line point2←
    line point1 + ((midpoint-line point1)*2)]
inMiddle-test [↑((line point1 + line point2) / 2 - midpoint) abs
    < self tolerance asPoint]
inMiddle-error [↑(line point1 + line point2) / 2 - midpoint]

```

[Minor detail: the system has inserted *self* in the first method so that *midpoint←* is a message to the instance of *MidPointLine*, rather than being an assignment statement. See the section on *Assignment Statements* later in this chapter.]

Besides altering parts of the owner, a constraint may merely *reference* some of its owner's parts. The referenced parts may not be changed to satisfy the constraint, thus allowing the implementation of one-way constraints. For example, suppose the user wants a word to be the result of concatenating a prefix and a stem. When either the prefix or the stem is changed, the word should be updated; but changing the word cannot affect either the prefix or the stem. (If someone sends a message trying to change the word without changing the prefix or stem, the word will spring back to its old value.)

Class ConcatenationExample**Superclasses**

Object

Part Descriptions

```

prefix:  a String
stem:    a String
word:    a String

```

Constraints

```

word = (prefix concat: stem)
word ← prefix concat: stem
prefix reference
stem reference

```

The following methods are compiled:

Class ConcatenationExample**Methods**

```

concatenate-word [self word← prefix concat: stem]
concatenate-test [↑word = (prefix concat: stem)]
concatenate-error [word = (prefix concat: stem) ⇒ [↑0.0] ↑1.0]

```


Only the first method has generated any executable code -- the other two are used only during planning. The error will be simply 0 or 1, since either the word is correct or it isn't.

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 machine.

As has been previously discussed, Smalltalk makes a strong distinction between the insides and the outsides 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 will have 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 will satisfy the constraint, given the states of all other parts. [This is described by the Boolean variable *uniqueState* listed previously.] In all the examples given, *uniqueState* has been true.

This way of describing constraints allows the representation of relations that are not very analytically tractable. 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 will have a single method that affects the entire object that owns the constraint, and this message will not be *uniqueState*. However, in such a case, the constraint satisfier will have little to work with, and only one such constraint can be handled.

Constraints on Sets

A subclass of Constraint, namely class SetConstraint, is used to represent constraints that apply to the members of a set. The rule portion of a SetConstraint is the same as for a normal constraint. However, rather than being listed explicitly for each constrained part, the methods that the SetConstraint can use to satisfy itself are specified by a template for the method to be used to alter any one of the members of the set. (All the members are treated alike.) An example of an object that uses a SetConstraint is an ElectricalNode, as described in Chapter 2.

Class ElectricalNode**Superclasses**

ElectricalObject

Part Descriptions

voltage: a Voltage

currents: a Set

location: a Point

Constraints

currents sum = 0.0

forEach: current in: currents methods:

[current ← 0.0 - (currents excluding: current) sum]

For this class, to make it easier to find all the parts affected by the constraint, the set is stored as a set of paths to the currents, rather than as direct pointers [cf. the section on *External References* in Chapter 3]. During constraint satisfaction, the constraint on the node generates a method for each current flowing into it by substituting the appropriate path from the set for the formal name *current* in the template. For example, consider two resistors *r1* and *r2* that are connected in series with the merge *r1 lead2 node* \equiv *r2 lead1 node*. The set of currents flowing into the node that connects them would contain the paths *r1 lead2 current* and *r2 lead1 current*. When the constraint was asked for its methods, it would return:

```
r1 lead2 current ←
```

```
0.0 - (r1 lead2 node currents excluding: r1 lead2 current) sum
```

and

```
r2 lead1 current ←
```

```
0.0 - (r2 lead1 node currents excluding: r2 lead1 current) sum.
```

Two message plans are also generated to describe these methods. When one of the message plans is asked to generate code to invoke its method, it inserts that method in-line.

Who Owns the Constraints?

A constraint on several objects can be owned by any object that has all of the constrained objects as parts or subparts. For example, the horizontal constraint involves two points, both of which are parts of an instance of *HorizontalLine*. The constraint is owned by *HorizontalLine*.

Meta-Constraints

There are a number of pieces of information embedded in ThingLab's structures and code that are best regarded as *meta-constraints*, that is constraints on how other constraints are satisfied. For example, in the midpoint constraint, the user preferred that the constraint be satisfied by altering the midpoint rather than one of the endpoints of the line. Similarly, the anchors on the

constants in the temperature converter in Chapter 2 tell the constraint satisfier that these numbers should not be changed to satisfy other constraints. There is also a global meta-constraint built into ThingLab's constraint satisfaction mechanism: don't change something unless you have to.

Currently, ThingLab has no general meta-constraint facility. It would be well if these sorts of meta-constraints were explicitly represented and used. Some ideas on this may be found in Chapter 6.

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 Chapter 2. The first use is to represent connectivity, for example, to connect the sides of the quadrilateral. The other is for applying pre-defined constraints, as was done with the midpoint constraint. As with constraints, adding or modifying a merge is a structural change, so only prototypes will 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 are more efficient. [For a while, merges were always represented using equality constraints; but this was abandoned because it was too slow for typical uses of ThingLab. However, it *was* much simpler - see the section *Programming Difficulties* in this chapter.] 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.

Constructing a Merge

[Warning: heavy seas for the next six nautical paragraphs.]

The method for the *merge:* message uses another message, *mergeWith:*, which will now be described. The default method for *mergeWith:* is implemented recursively as follows. The receiver *self* and the argument *arg* must both be instances of the same class. After checking that this is the case, a new instance *result* of the receiver's class is made. Then, *result*'s parts are constructed by merging the corresponding parts of *self* and *arg*, again using the *mergeWith:* message. The process stops when primitive parts are reached. Some objects have their own interpretations of the *mergeWith:* message. For example, the result of merging two sets is the union of those sets.

The internal effects of the *merge:* message depend on what sorts of parts are being merged. There are three cases. In the first case, the parts being merged are all instances of the same class, and are ordinary parts as opposed to primitive parts. [A primitive object, e.g. an integer, is one that has no parts of its own.] In the second, the parts are again not primitives, but are instances of different classes. In the third, the parts are primitive parts (the classes don't matter in this case.)

In the first case, a new part is constructed using the *mergeWith:* message described above. A pointer to this new part is then substituted for each pointer to either of the original parts. In addition, an instance of class MergeConstraint is created that owns paths to the merged points, and is indexed in a table owned by the object's class.

In the second case, the simple strategy described above won't work, because the *mergeWith:* message can be used only with instances of the same class. [Otherwise, what would be the class of the result?] Instead, the object finds the nearest common primary superclass of the classes of each of the parts being merged. [Since all classes are subclasses of class Object, all classes have some superclass in common.] Next, the object asks the common superclass for its part names, and sends itself a series of *merge:* messages to merge the subparts of the parts being merged that were inherited from this common superclass. If the common superclass has no parts, then the merge is *vacuous*, and the error handler is invoked. [This will not happen when graphically adding a merge with the ThingLab window. A moving object will stick only to those parts with which it can merge non-vacuously.]

An example may make this second case clearer. Suppose that we have an object with two parts, a HorizontalLine and a DottedLine. Both parts are instances of subclasses of Line.

Class MergeExample

Superclasses

GeometricObject

Part Descriptions

part1: a HorizontalLine

part2: a DottedLine

We ask it to merge the two lines by sending it the message
`merge: 'part1', 'part2'.`

[The argument is a list of length 2 consisting of a path to *part1* and a path to *part2*.] It cannot use the simple method, since the parts are of different classes. Therefore, it finds the nearest common superclass of *HorizontalLine* and *DottedLine*, which is *Line*. Next, it finds the part names of *Line*, namely *point1* and *point2*. Finally, it sends itself the messages

`merge: 'part1 point1', 'part2 point1'`

and

`merge: 'part1 point2', 'part2 point2'.`

Since the endpoints are all instances of class *Point*, *MergeExample* can use the simple kind of merging in responding to these new messages.

In the third case, the parts being merged are primitives. Here, the object will automatically construct an equality constraint instead of using a merge. The reasons for this will be discussed in the section *Assignment Statements*, to follow. [This is why the horizontal line had a constraint rather than a merge to keep equal the *y* values of its endpoints.]

The complexities of the second case arise because of the way primary superclasses are represented; for other superclasses, which are represented in *ThingLab* by having an instance of the superclass as a part, the first method can be used. Indeed, before the algorithm for doing the second case was discovered, *all* subclasses in *ThingLab* were represented by including an instance of the superclass as a part.

Symmetry

Are merges symmetric with respect to the merged parts? This question has been divided into two parts: symmetry in the representation of the merge, and symmetry during the process of adding the merge. The representation is completely symmetric. The process of adding the merge could be symmetric, but often is not. In the editor, normally one of the parts being merged is moving with the cursor, while the other is stationary. The stationary part is given preference in computing the result of the merge. For example, when the line part of the moving *MidPointLine* was merged with a side of the quadrilateral, the result of the merge was a new line whose endpoints had the same locations as those of the original side.

Programming Difficulties

The most difficult parts of the *ThingLab* system to program and debug were those that deal with adding and editing merges. As an example of the sort of complexity that arises, consider the following editing operation. The object being edited has two parts, both triangles. A side of the first triangle is being merged with a side from the second.

Class TwoTriangles**Superclasses**

GeometricObject

Part Descriptions

triangle1: a Triangle

triangle2: a Triangle

Mergestriangle1 side3 \equiv triangle2 side1

When the sides are merged, a new line results. The new line must be substituted for the old *side3* of *triangle1* and for the old *side1* of *triangle2*. Further, because of the merges in the class Triangle, *point1* of the new line must be substituted for *triangle1 side1 point1* and for *triangle2 side3 point1*. Similarly, *point2* of the new line must be substituted for *triangle1 side2 point2* and for *triangle2 side2 point1*. All this is happening in the context of an interactive editing session, so it shouldn't take forever.

Perhaps all this could be simplified by viewing merges as constraints on bindings, and using the regular constraint satisfaction mechanism; but this has not been done in the current version.

Assignment Statements

[This section supplies additional detail not essential to the material in the chapters that follow.]

As should be apparent from the above description, substituting a new part for an old one can be expensive when merges are present. However, in ThingLab it is a comparatively rare operation. The usual case of an assignment statement is handled by recursively copying the values from one object into the other, rather than by smashing the pointer to the old object. A little background is needed in explaining this.

In procedure-oriented languages, a reasonable way of viewing assignment is that there is an assignment operator to which the system presents the address of a variable along with a new value to be stored into that variable. Smalltalk doesn't do it this way. Rather, the object that has the variable as a part is sent a message requesting assignment of some value to that variable. This message will invoke a method. The method may substitute a pointer to the new value for the pointer to the old one; but then again it may do something else. The sender of the message need only be concerned with its effect, not with the internal method for receiving the message. This is another example of Smalltalk's strong distinction between the outside and the inside of an object.

As described in Chapter 3, methods for reading and writing an object's parts are automatically generated and compiled by the part descriptions. The exact methods depends on the type of

the part, which will be either *ordinary* or *primitive*. Primitive objects, e.g. integers, are considered to have no parts of their own. The part description always compiles the same method for reading the field, but compiles different methods for writing it depending on whether the part is ordinary or primitive. [It finds what type it is by sending the part's class the message *isPrimitive*.] For an ordinary part, such as a line that is the side of a triangle, the following sorts of methods are compiled.

Class Triangle

side1 [*^side1*]

side1 ← *temp* [*side1 copyFrom: temp*]

For those not familiar with Smalltalk syntax, the first method is interpreted as follows:

To receive the message *side1*, return my part named *side1*.

The second method is interpreted as:

To receive the message *side1* ←, get a single argument and bind it to the temporary variable *temp*. Ask my part named *side1* to copy its values from *temp*. [The selector *side1* ← is one atom - the identifier and the arrow are grouped into a single selector by the compiler.]

Note that in this method the pointer to the side is not changed, so that none of the triangle's merges will be affected.

On the other hand, for a primitive part, such as the *x* value of a point, a method for writing this part is compiled that does change the pointer to the value.

Class Point

x [*^x*]

x ← *temp* [*x* ← *temp*]

The first method is interpreted as in the previous case. The second method is interpreted as:

To receive the message *x* ←, get a single argument and bind it to my part named *x*.

The implementation of the *copyFrom:* message is of interest. For an object composed of ordinary parts, such as a triangle, the message is interpreted as follows:

Class Triangle

copyFrom: otherTriangle

[*side1 copyFrom: otherTriangle side1*.

side2 copyFrom: otherTriangle side2.

side3 copyFrom: otherTriangle side3]

On the other hand, the interpretation is different for an object with primitive parts, such as a point.

Class Point

copyFrom: otherPoint

[*x* ← *otherPoint x*.

y ← *otherPoint y*]

For an ordinary part, the *copyFrom:* message results in recursively sending the *copyFrom:* message to that part. For a primitive part, the pointer in the field is changed. [In the actual system, each class doesn't have its own method for *copyFrom:*. Rather, there is a default implementation in class `Object` that branches on the type of the part.]

An important feature of this scheme is that the external behavior of both kinds of parts are the same. If an object has a part p , the external manifestation of this is that the object understands the messages p and $p\leftarrow$. Internally, the methods for receiving these messages depend on what type of a part is p .

Chapter 5 - Constraint Satisfaction

Overview of Constraint Satisfaction

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 will generate 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 Chapter 2 (shown again in Figure 5.1).

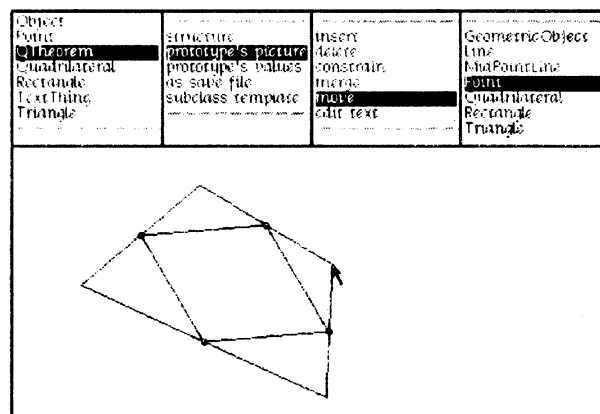


Figure 5.1 - Moving the vertex of a quadrilateral

When the user selects *mov? 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 will first attempt 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 will ask the object for a method for dealing with circularity. Currently, relaxation is the only such method available. If relaxation is used, the user will be warned, so that perhaps some other redundant constraints can be supplied that will eliminate the need for relaxation.

Constraint Satisfaction Methods

The constraint satisfaction methods used in ThingLab will now be described in more detail. To illustrate the operation of the methods, the electrical circuit example from Chapter 2 will be used. [See that chapter for descriptions of the parts involved. Additional labels have been added to the picture in Figure 5.2 to make it easier to refer to the various parts.]

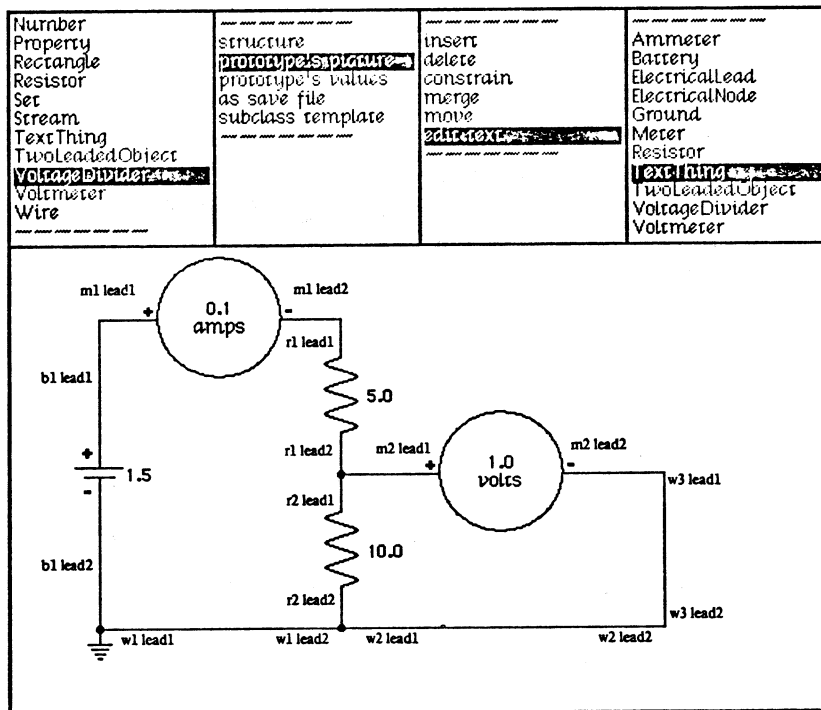


Figure 5.2 - A voltage divider

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 until no more degrees of freedom can be propagated.

Because of the difficulty of giving a precise definition of degrees of freedom for non-numeric objects, the current constraint satisfier uses a simple-minded 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 doesn't matter whether or not the constraint determines the part's state uniquely (removes all its degrees of freedom). [The power of the method could be increased by taking advantage of the cases where better descriptions can be given of the degrees of freedom available to a part.]

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 doesn't affect the line's second endpoint.

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.

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, i.e., parts that have no degrees of freedom. If such a part is found, the constraint satisfier will look 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 will be used to find its state, and run-time checks will be 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* will be known. [Actually, it is already known during planning, but the constraint satisfier doesn't use this information.] Also, by the battery's constraint, *b1 lead1 node voltage* will be known, which 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* will be known. Similarly, the voltmeter has a constraint that it draw no current, and so the current in its leads and connecting wires will be 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.

Relaxation

If there are constraints that cannot be handled by either of these techniques, the constraint satisfier will ask 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). [See Sutherland 1963 for a fuller description of the relaxation method.]

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 both the constraints on *P* itself, and also these other parts. [The heuristic for choosing *P* was adapted from that used in the EL circuit analysis program for picking an unknown when doing symbolic algebraic manipulations (Stallman & Sussman 1977)].

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 will try 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* will be relaxed.

Using Multiple Views to Avoid Relaxation

Using the method employed by Steele and Sussman [Steele & Sussman 1978], another view of the voltage divider may be added that obviates the need for relaxation. First, a new class is defined that embodies the fact that two resistors in series are equivalent to a single resistor.

Class SeriesResistors

Superclasses

ElectricalObject

Part Descriptions

rA: *a Resistor*

rB: *a Resistor*

rSeries: *a Resistor*

Constraints

rSeries resistance = rA resistance + rB resistance

rSeries resistance ← rA resistance + rB resistance

rA resistance reference

rB resistance reference

Merges

rA lead2 node ≡ rB lead1 node

rA lead1 ≡ rSeries lead1

rB lead2 ≡ rSeries lead2

The constraint specifies that the resistance of the equivalent single resistor be equal to the sum of the resistors in series. The first merge simply connects *rA* and *rB* in series. The other merges, however, apply to leads rather than nodes. The second merge states that *rA lead1* and *rSeries lead1* are identical. Hence, the currents in these leads are also identical, and *rA lead1 node* will have a single current flowing into it. [All this is handled automatically by the system -- all the user needs to do is enter the merge.] The third merge plays an analogous role.

In the picture of an instance of *SeriesResistors* (Figure 5.3), the symbol for *rSeries* is offset to the left and connected with dotted lines, indicating that it is an equivalent circuit element rather than a resistor in parallel with *rA* and *rB*.

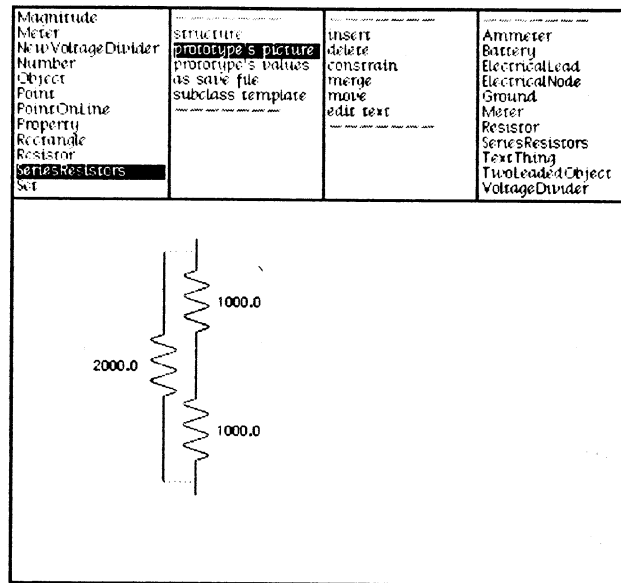


Figure 5.3 - Picture of the prototype SeriesResistors

The resistors r_A and r_B of SeriesResistors are designated as attachers. To add this new description to the voltage divider, an instance of SeriesResistors is inserted in the circuit (call it *series*), and the resistors r_A and r_B of *series* are merged with the existing resistors r_1 and r_2 in the circuit (Figure 5.4).

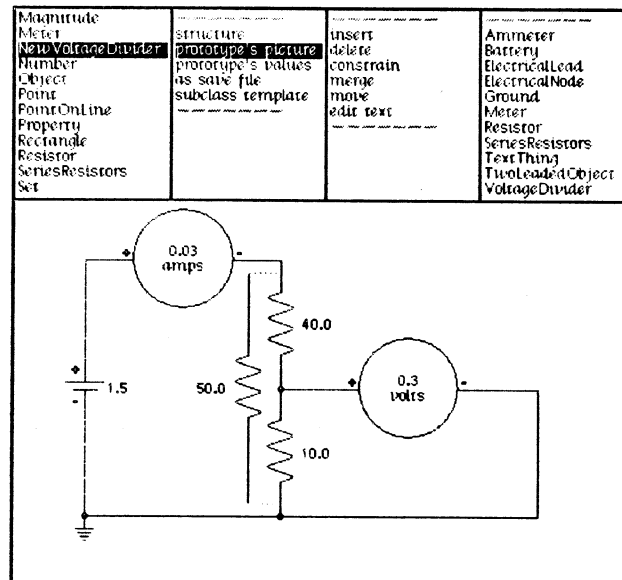


Figure 5.4 - The voltage divider with an added instance of SeriesResistors

Using this additional description, all the constraints can be satisfied in one pass. As previously described, m_1 lead2 node voltage and w_1 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.

There are many questions remaining to be explored in connection with the use of multiple views in constraint satisfaction. [See also Chapter 6.] For example, one should be able to tell the system explicitly about redundant views. This would allow the system to use propagation of degrees of freedom in the presence of such views, and would also allow it to do some checking to see that it was appropriate to apply the redundant view.

In the above example, propagation of degrees of freedom was used to postpone updating the ammeter's reading until after the voltages and currents had been found. However, suppose that there were another constraint on the ammeter that gave a redundant description of how to update the text that displays its reading. In this case, the system would not have been able to use propagation of degrees of freedom, since it would not know that a value for the text could always be found that would satisfy both constraints simultaneously. If these two constraints were explicitly described as being redundant descriptions, however, the method could still be used.

In regard to checking whether it is appropriate to apply a redundant view, consider the process described above of viewing two resistors in series as being equivalent to a single resistor. The parts *rA* and *rB* of an instance of *SeriesResistors* should only be merged with a pair of resistors that are already connected, and there should be no significant other current flowing from the center node of the resistors. In one sense, the system already ensures that these conditions will be satisfied. If the existing resistors are not already connected, the act of merging the two resistors from an instance of *SeriesResistors* will cause them to be connected. Also, if there is a significant other current flowing from the center node of the resistors in series, not all the constraints will be satisfied at run-time. However, the system could do a better job of informing the user of unsatisfiable constraints if it knew about the use of a redundant description.

The Process of Transforming a Message Plan

During planning, an object is presented with a message plan that describes a Smalltalk message that might be sent to it at run-time. The object constructs a method for receiving such messages, and returns a new message plan that gives directions for invoking the method. This is called *transforming* the message plan. In the simple case, when no constraints are present, the result of transforming the message plan is the same as the original plan. However, a quite different plan may be returned if there are constraints that would be affected by the message

described by the original plan.

As a simple example of this, consider moving an endpoint of a horizontal line. During planning, the horizontal line is presented with the message plan

```
horizline point1 moveby: delta.
```

This message plan represents the actual request that might be made of the line at run-time. The horizontal line constructs a Smalltalk method for receiving a message that moves its *point1*, while still keeping itself horizontal. A selector is chosen for this method, say *point1Moveby*. Finally, the horizontal line returns a new message plan that provides directions for invoking this method, namely

```
horizline point1Moveby: delta.
```

It is important to remember that a message is a request to an object, not an operator that can freely manipulate the object that receives it. Thus, if an object is asked to transform a message plan, one of the things it can do is to indicate that it will refuse to receive such messages. In this case, either the sender of the rejected message plan is notified, or if no provision was made for this, the Smalltalk error handler is called.

A Detailed Example

A more detailed example of transforming a message plan will now be given. The following classes will be used in this example:

Class Line

Superclasses

GeometricObject

Part Descriptions

point1: a Point

point2: a Point

Class HorizontalLine

Superclasses

Line

Constraints

point1 y = point2 y

point1 y ← point2 y

point2 y ← point1 y

Class VerticalLine**Superclasses**

Line

Constraints

point1 x = point2 x
 point1 x \leftarrow point2 x
 point2 x \leftarrow point1 x

Class DemoRectangle**Superclasses**

GeometricObject

Part Descriptions

side1: a VerticalLine
 side2: a HorizontalLine
 side3: a VerticalLine
 side4: a HorizontalLine

Merges

side1 point1 \equiv side2 point1
 side2 point2 \equiv side3 point1
 side3 point2 \equiv side4 point2
 side4 point1 \equiv side1 point2

The object used in the example will be an instance r of DemoRectangle (Figure 5.5). [The class DemoRectangle is not the same as the normal Smalltalk class Rectangle, which has as its parts an upper-left hand corner and a lower right-hand corner. Constraint satisfaction would be trivial for instances of Rectangle.]

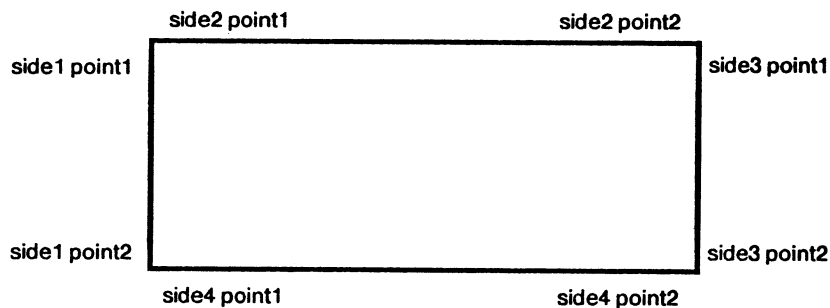


Figure 5.5 - An instance of DemoRectangle

Suppose that the user wishes to pick up and move the right-hand side of r with the cursor. In the ThingLab window, this is accomplished by repeatedly asking r to move its side by some increment $delta$, where $delta$ is a point. [Strictly speaking, $delta$ ought to be an instance of PointIncrement or some such class, rather than Point.] First, a message plan m is constructed and presented to r .

This plan is:

r side3 moveby: delta

Here, the receiver is *r*, the path is *side3*, the action is *moveby:*, and the symbolic argument is *delta*. The rectangle is asked to transform *m* into a new message plan that gives directions for invoking a method for moving *side3*, while also satisfying all the rectangle's constraints. Once a transform of *m* has been found, the window can use this information in moving the side.

The steps in transforming a message plan are outlined below. In the following sections, each of these steps is described in more detail.

First see if a transform of the message plan has already been constructed. If so, use it.

If not:

Check whether a message described by the plan would affect any of the receiver's constraints. If not, then forward the message plan to the part, and use the transform returned by that part.

Otherwise:

Build a queue of message plans describing methods that might need to be invoked to satisfy the constraints. First, do a compile-time expansion of the original plan.

Add to the queue other message plans by checking the constraints and merges of the receiver and its parts.

Process the message plans in the queue. If possible, find a one-pass ordering for invoking at run-time the methods described by the plans; otherwise, compile a call on the relaxation method. Include run-time checks if needed.

Using the ordering found in the previous step, compile a Smalltalk method.

Finally, construct a transform of the message plan, whose action is to invoke the Smalltalk method. Index the plan and its transform in a dictionary for future use.

The Dictionary of Message Plans and Transforms

When *r* is asked to transform the message plan *m*, it first checks in its dictionary of transforms to see if such a transform has already been constructed. This dictionary consists on the name side of incoming message plans, and on the value side of transforms of these plans. In this case, suppose a transform for *m* is not in the dictionary.

If the prototype *DemoRectangle* were edited by adding or deleting parts, or its constraints or merges were changed, some of the transforms might become obsolete. Currently, all transforms are forgotten when any such change occurs. An improvement would be to check which ones are affected by the change, and to forget only these entries. [This would amount to keeping track of dependencies on a class-wide basis.]

Checking the Message Plan for Interaction with the Receiver's Constraints

The first name on the path of the message plan is *side3*. If *r* had no constraints or merges involving *side3*, then the transform would be constructed using a transform obtained from *VerticalLine* as a "subroutine". However, *r* does have merges involving *side3*, so this cannot be done. An example of using the transform obtained from a part will be presented later in this chapter.

Compile-time Expansion of Message Plans

The next step is to check if the message plan has a compile-time expansion. A message queue is created, and *r* asks its part, pointed to by the path *side3*, to expand the original plan into this queue. The vertical line uses the default expansion method inherited from *Object*, and expands the *moveby:* message plan into two *moveby:* message plans, one for each of its endpoints. The endpoints are in turn asked to do a compile-time expansion of *these* message plans. Points, however, are themselves willing to receive *moveby:* messages at run-time, and the expansion stops. The message queue is now:

MessageQueue

- 1: *side3 point1 moveby: delta (alters side3 point1)*
- 2: *side3 point2 moveby: delta (alters side3 point2)*

More information about compile-time expansion may be found in the *Leftovers* section.

Adding Message Plans from Constraints to the Queue

Each message plan in the queue is checked to see if its path overlaps the paths of any of the receiver's constraints or merges. If a constraint overlaps a message plan, then copies of the constraint's message plans are added to the queue. On the other hand, if a merge overlaps one of the message plans, the message plan is checked for each distinct path to the merged part.

The rectangle itself has no constraints. However, two of its merges do overlap the message plans in the queue. The paths *side3 point1* and *side2 point2* both refer to the same point. Therefore, the first message plan will be checked using both of these paths. The other message plan is treated in an analogous fashion.

Each message plan in the queue must now be checked against the constraints and merges of *r*'s parts. The first message plan with the first of its merged paths is

side3 point1 moveby: delta.

The first name on the path is stripped off, and the message plan

point1 moveby: delta

is forwarded to *side3*. This part is a vertical line, and its constraint overlaps the path of the forwarded message plan. Consequently, copies of the message plans that describe the methods

this constraint can use to satisfy itself are added to the queue. The queue is now:

MessageQueue

- 1: *side3 point1 moveby: delta (alters side3 point1)*
- 2: *side3 point2 moveby: delta (alters side3 point2)*
- 3: *side3 vert-point1-x (alters side3 point1 x)*
- 4: *side3 vert-point2-x (alters side3 point2 x)*

Message plan 3 describes one of the methods for satisfying the vertical line's constraint, namely the method

vert-point1-x

[*point1 x ← point2 x*]

Message plan 4 describes the other method. Note that the paths of the constraint's message plans have been prefixed by the name *side3*, since these messages are now for *r*.

The vertical line *side3* strips off the next name on the path of message plan 1, and forwards the message plan

moveby: delta

to its first endpoint. The point has no constraints or merges of its own. The plan's path is now empty, and so there are no more subparts to be checked.

The other path to the upper right-hand corner, namely *side2 point2*, is now checked. This causes the constraint on the top horizontal line to add message plans 5 and 6.

MessageQueue

- 1: *side3 point1 moveby: delta (alters side3 point1)*
- 2: *side3 point2 moveby: delta (alters side3 point2)*
- 3: *side3 vert-point1-x (alters side3 point1 x)*
- 4: *side3 vert-point2-x (alters side3 point2 x)*
- 5: *side2 horiz-point1-y (alters side2 point1 y)*
- 6: *side2 horiz-point2-y (alters side2 point2 y)*

The other *moveby:* message plan is treated in a similar fashion. The message plans added by the constraints are checked as well, since they might overlap some other constraint or merge. There is a mechanism for preventing infinite recursion: before adding copies of its message plans, a constraint checks to see if it has already added copies to the queue. If so, no new copies are added.

The final state of the queue is:

MessageQueue

- 1: *side3 point1 moveby: delta (alters side3 point1)*
- 2: *side3 point2 moveby: delta (alters side3 point2)*
- 3: *side3 vert-point1-x (alters side3 point1 x)*
- 4: *side3 vert-point2-x (alters side3 point2 x)*
- 5: *side2 horiz-point1-y (alters side2 point1 y)*
- 6: *side2 horiz-point2-y (alters side2 point2 y)*
- 7: *side4 horiz-point1-y (alters side4 point1 y)*
- 8: *side4 horiz-point2-y (alters side4 point2 y)*

Message plans 3 and 4 were added by the vertical constraint on *side3*, message plans 5 and 6 by the horizontal constraint on *side2*, and message plans 7 and 8 by the horizontal constraint on *side4*. None of the various paths of the message plans in the queue overlap the vertical constraint on *side1*, and so that constraint added no plans.

Processing the Message Plans in the Queue

The first step in processing the message plans is to look for preferences. The system distinguishes preferences from relations that must hold. Requests made by the user to change a value are considered preferences: the system will attempt to honor them, but if the request violates a constraint, it will be overridden. Message plans 1 and 2, which represent the user's request to move a side, are preferences of this sort. The messages described by these plans will be sent first. The system will attempt not to undo these edits, but may do so if pressed. After processing the preferences, message plans 1 and 2 are removed from the queue, and put in the list of plans describing messages to be sent first. Also, the constraint satisfier notes that the points that these message plans affect should not be changed unless necessary.

Messages to be Sent First

- 1: *side3 point1 moveby: delta (alters side3 point1)*
- 2: *side3 point2 moveby: delta (alters side3 point2)*

Prefer not to Alter

side3 point1
side3 point2

Propagation of degrees of freedom is used next. In using this method, the constraint satisfier looks for a message plan such that its target has enough degrees of freedom to satisfy all its constraints. It turns out that all six remaining message plans meet this requirement, since each of the numbers that are their targets has only one constraint that affects it. Thus, any of these numbers could be chosen so as to satisfy its constraint. However, the constraint satisfier would prefer not to alter *side3 point1* or *side3 point2*. The constraint satisfier notes that, after taking merges into account, these two paths overlap all the message plans' paths except for those of message plans 5 and 7. The constraint satisfier then decides that the messages described by

these two plans can be sent last. Since these messages will take care of satisfying the two horizontal constraints, the remaining two message plans added by the horizontal constraints can be discarded.

Messages to be Sent First

- 1: side3 point1 moveby: delta (alters side3 point1)
- 2: side3 point2 moveby: delta (alters side3 point2)

Messages to be Sent Last

- 5: side2 horiz-point1-y (alters side2 point1 y)
- 7: side4 horiz-point1-y (alters side4 point1 y)

Discarded Messages

- 6: side2 horiz-point2-y (alters side2 point2 y)
- 8: side4 horiz-point2-y (alters side4 point2 y)

Prefer not to Alter

side3 point1
side3 point2

[Incidentally, if the system were running on a multi-processor machine, the constraint satisfier could have included as part of the plan the information that messages 5 and 7 could be sent in parallel. See Chapter 6.]

Propagation of degrees of freedom is tried again. This time, the only message plans remaining do affect *side3*. So the constraint satisfier decides to send the message described by one of these plans, say message plan 3. [The constraint satisfier is not smart enough to realize that if the vertical constraint were satisfied before moving the line, it would be satisfied after the entire line is moved. Thus, the message described by plan 3 will actually be sent, even though it won't change anything.] Sending this message will take care of satisfying the vertical line constraint, so message plan 4 may be discarded. The queue is now empty, and this phase of constraint satisfaction is over.

Messages to be Sent First

- 1: side3 point1 moveby: delta (alters side3 point1)
- 2: side3 point2 moveby: delta (alters side3 point2)

Messages to be Sent Last

- 3: side3 vert-point1-x (alters side3 point1 x)
- 5: side2 horiz-point1-y (alters side2 point1 y)
- 7: side4 horiz-point1-y (alters side4 point1 y)

Discarded Messages

- 6: side2 horiz-point2-y (alters side2 point2 y)
- 8: side4 horiz-point2-y (alters side4 point2 y)
- 4: side3 vert-point2-x (alters side3 point2 x)

Prefer not to Alter

side3 point1
side3 point2

Compiling Smalltalk Code

A plan has been constructed for moving *side3* while still satisfying all the rectangle's constraints. The system now embeds this plan in a Smalltalk method understood by *DemoRectangle*. A name for this method is generated by ingloriously squishing together the path and action from the message plan. Code for the method is accumulated by asking each of the message plans to add code to a stream. The method compiled by the system is:

```
side3Moveby: delta
  [side3 point1 moveby: delta.
   side3 point2 moveby: delta.
   side3 vert-point1-x.
   side2 horiz-point1-y.
   side4 horiz-point1-y.]
```

At run-time, this method will in turn invoke methods belonging to the horizontal and vertical constraints, namely

```
Class Horizontalline
  horiz-point1-y
  [point1 y ← point2 y]
Class Verticalline
  vert-point1-x
  [point1 x ← point2 x]
```

Constructing the Transform

The rectangle *r* is now able to return a transform of the original message plan

```
side3 moveby: delta.
```

The transform is simply

```
side3Moveby: delta.
```

In other words, the transform has an empty path, and its action is to invoke the newly compiled method. The message-transform pair is indexed in *DemoRectangle*'s dictionary of transforms. At this point, the reader who has waded through all the steps will appreciate the virtues of saving this transform for future use.

Leftovers

Parts of the transformation process not exercised by the above example will now be described.

Using Transforms Obtained from Parts

In the last section, it was mentioned that in the absence of interactions with the receiver's constraints, the transform obtained from a part could be used as a "subroutine". Consider an object consisting simply of two rectangles.

Class TwoBoxes**Superclasses**

GeometricObject

Part Descriptions

box1: a DemoRectangle

box2: a DemoRectangle

Suppose that the user wants to move the right-hand side of *box2*. The instance of TwoBoxes is asked to transform a message plan *m*, namely

box2 side3 moveby: delta.

Further, suppose that there is no entry for *m* in TwoBoxes' dictionary of transforms. The next step is to check the first name on *m*'s path for interactions with TwoBoxes' parts. The first name is *box2*. There are no constraints or merges involving *box2* (in fact there are none at all). Therefore, the transform obtained from DemoRectangle may be used. TwoBoxes strips the first name off *m*'s path, and asks DemoRectangle to transform the message plan

side3 moveby: delta.

DemoRectangle looks up this plan in its dictionary, and quickly returns the transform *side3Moveby: delta.*

TwoBoxes inserts the name *box2* at the head of the path, and returns as *its* transform *box2 side3Moveby: delta.*

Another way of describing this process is that when there is no interference, plans obtained from an object's parts are re-used. The result of this is that plans for receiving a message are stored as far down in the part-whole hierarchy as possible. There are much more powerful ways in which the idea of plans obtained from subparts could be used. See Chapter 6.

More about Compile-time Expansion

Compile-time expansion is used with message plans that describe such things as translation and scaling. Most message plans, however, simply expand to themselves.

There are two reasons for using the technique of compile-time expansion. First, it is not easy to write, say, an efficient but general *moveby:* message. A first cut might be to have a default *moveby:* message that simply asked each part to move, with points overriding the default. This handles such things as lines, but fails in general. Consider a triangle.

Class Triangle**Superclasses**

GeometricObject

Part Descriptions

side1: a Line

side2: a Line

side3: a Line

Mergesside1 point1 \equiv side3 point1side1 point2 \equiv side2 point1side2 point2 \equiv side3 point2

If the triangle received a *moveby:* message, it would in turn ask each of its sides to move. Each side would then ask its endpoints to move. But because of the merges, each vertex would be moved twice! It would be possible to keep track at run-time of all the objects that have already been moved. However, by using compile-time expansion, this can all be done during planning. The original *moveby:* message plan would be expanded to:

MessageQueue

- 1: side1 point1 moveby: delta
- 2: side1 point2 moveby: delta
- 3: side2 point1 moveby: delta
- 4: side2 point2 moveby: delta
- 5: side3 point1 moveby: delta
- 6: side3 point2 moveby: delta

The system then notices which of the message plans refer to the same object, and eliminates the duplicates. The following code would be compiled for Triangle:

moveby: delta

```
[side1 point1 moveby: delta.
 side1 point2 moveby: delta.
 side2 point2 moveby: delta.]
```

One way of looking at the process of eliminating duplicate message plans is to notice that the duplications arise because of merges. Merges are just an efficient way of representing an equality constraint. If the merged parts had not actually been collapsed internally, there would have been an appropriate number of messages to each part.

The benefits of doing compile-time expansion during planning are greater with more complex message plans. Another message plan that undergoes compile-time expansion is the "move the *i*-th attacher" message. (See Chapter 2 for examples of attachers.) The expansion here is
 fix the locations of attachers 1 to *i*-1, and move attachers *i* to *n*
 (where *n* is the total number of attachers)

This message is used when inserting a new part into the object being edited. For example, suppose the three vertices of an instance of Triangle have been designated as attachers. When inserting a triangle, the user positions the first vertex (with the entire triangle following), then the second vertex (moving the second and third vertices), and then the third vertex. During planning, the message plan

```
moveAttacher: 2 by: delta
```

is expanded to:

MessageQueue

```
1:  side1 point1 fix
2:  side1 point2 moveby: delta
3:  side2 point2 moveby: delta
```

[Note: the message *fix* means "Add this path to *Unalterable*, but don't generate any code".] Again, all this *could* be done at run-time, but much less efficiently.

The other reason for using compile-time expansion is that it is a way to avoid waking up constraints needlessly. For example, consider the voltage divider again. Suppose the user wants to move *r1*. This involves asking the voltage divider to transform the message plan

```
r1 moveby: delta.
```

Using the constraint satisfaction scheme described above, but without compile-time expansion, any constraint overlapping the path *r1* would add message plans to the queue. In other words, *r1*'s electrical as well as graphical constraints would need to be checked. With compile-time expansion, the plan is expanded to

```
r1 lead1 node location moveby: delta
r1 lead2 node location moveby: delta
r1 label frame origin moveby: delta
r1 label frame corner moveby: delta
```

and only the graphical constraints on *r1* are checked.

More about Preferences

In the above examples, the user's desires could be completely met, and all the object's constraints satisfied as well. This is not always possible. Consider a horizontal line with one end anchored. If the user tries to move the other end with the cursor, the moving end will in general not be able to follow the cursor exactly -- the *x* value of the endpoint will match that of the cursor, but the *y* value will remain constant (see Figure 5.6).

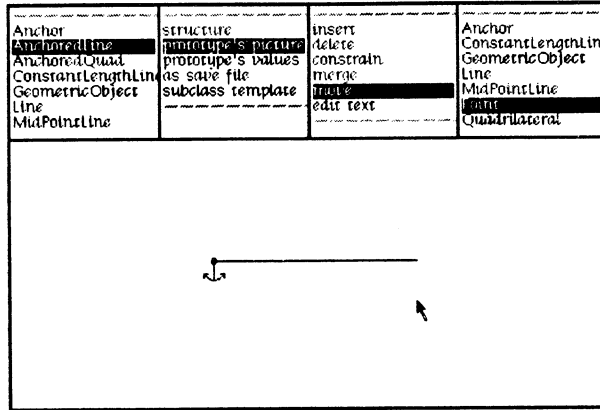


Figure 5.6 - An anchored horizontal line

The classes for this example are as follows.

Class HorizontalLine
 (as previously defined)

Class AnchoredPoint
Superclasses
 Point
Constraints
 self fixed

Class AnchoredLine
Superclasses
 GeometricObject
Part Descriptions
 line: a HorizontalLine
 anchor: an AnchoredPoint
Merges
 line point1 \equiv anchor

The compiled method for moving *point2* of an instance of AnchoredLine is:

```
AnchoredLine
  linePoint2Moveby: delta
    [line point2 moveby: delta.
     line horiz-point2-y]
```

In other words, the anchored line will first make the change to its *point2*, but will partially undo it to satisfy its constraint.

Constraints and the Procedural-Declarative Controversy

The last section of this chapter moves up from a detailed discussion of constraint satisfaction to a more general level.

Several years ago, a major debate in artificial intelligence circles was occurring between advocates of *procedural* and *declarative* representations of knowledge. Although indulgence in this debate is no longer as fashionable as it once was ["Anyone who hasn't figured this 'controversy' out yet should be considered to have missed his chance, and be banned from talking about it." McDermott 76], the issues raised by it still remain. It is interesting to view the current ThingLab constraint mechanism and its evolution in the light of this debate.

Superficially, the debate is between researchers who want to embed the knowledge that a system has in its procedures, and researchers who want to represent knowledge as a set of facts together with general-purpose programs for manipulating these facts. On the procedural side, an extreme case is a monolithic LISP program; the corresponding extreme on the declarative side is a system that represents all its knowledge in first-order predicate calculus, and uses a resolution theorem prover to manipulate these facts.

Constraints are an amalgam of both declarative and procedural information. On the declarative side, a constraint states what the relation is, which subparts of the object are affected by the constraint, and which subparts may be altered to satisfy it. On the procedural side, it describes procedures for making that relation be true. Notice, however, that the procedural parts of the constraint are rather tightly controlled. These methods are invoked by the system, not the programmer. Each of the methods, if invoked, must cause the constraint to be satisfied. The side effects of the methods are described by message plans: each plan has a path to the part of the constraint's owner that the method alters; the method may alter no other parts. There is also a relationship that must hold among the constraint's methods. For every part that is referenced by one of the methods, there must either be another method that alters it, or a dummy method referencing it.

In his essay *Frame Representations and the Declarative/Procedural Controversy* [Winograd 1975], Terry Winograd writes

... At this point it is tempting to look for a synthesis -- to say "You need both. Some things are better represented procedurally, others as declarative facts, and all we need to do is work on how these can be integrated." This reaction misses what I believe is the fundamental ground for the dispute. It is not simply a technical issue of formalisms, but is an expression of an underlying difference in attitude towards the problems of complexity. Declarativists and proceduralists differ in their approach to the duality between modularity and interaction, and their formalisms are a reflection of this viewpoint. ...

If we look at our debate between opposing epistemologies, we see two metaphors at opposite poles of the modularity/interaction spectrum. Modern symbolic mathematics makes strong use of modularity at both a global and a local level. Globally, one of the

most powerful ideas of logic is the clear distinction between *axioms* and *rules of inference*. ... Locally, axioms represent the ultimate in decomposition of knowledge. Each axiom is taken as true, without regard to how it will interact with the others in the system. ...

Programming, on the other hand, is a metaphor in which interaction is primary. The programmer is in direct control of just what will be used when, and the internal functioning of any piece (subroutine) may have side effects which cause strong interactions with the functioning of other pieces. ...

Viewed in this light, the constraint mechanism is more than the result of simply pasting together facilities for representing facts and procedures. An individual constraint provides a way of integrating the declarative description of a relation with procedures for achieving it, and is thus a more powerful tool than a simple statement of a fact. More globally, constraints achieve the same sort of modularity as declarative systems, in which each fact can be stated independently. Indeed, it is in the area of interaction that the current constraint mechanism falls down. Within it, one cannot represent such things as the masking off of a constraint, or a suggestion as to how to satisfy a set of constraints.

One can view the evolution of the constraint mechanism in ThingLab as starting with a rather strictly declarative representation, and moving toward the inclusion of more procedural sorts of knowledge -- richer ways of describing the interaction of a constraint with its environment. In the first implementations, constraints were represented using only an error expression, and relaxation was the only means of satisfying them. In contrast, the current system attempts to use the local procedures provided by the constraint for satisfying itself. Some further steps in this evolution are proposed in Chapter 6, such as constraints involving time, and meta-constraints.

Chapter 6 - Directions for Future Research

Introduction

This chapter is an exploration of the idea of building a programming language around the notions of constraints and hierarchy that have been used in ThingLab. At this point, a complete design does not exist, although many of the pieces are there. The reader is therefore warned that the proposals discussed in this chapter are incomplete and untested, and some of them will doubtless turn out to be unworkable.

The Next Version of Smalltalk

Discussion regarding the next major version of Smalltalk is currently under way in the Learning Research Group. An important goal in the design of the new Smalltalk is to integrate many of the language metaphors that LRG has investigated and implemented as subsystems over the past several years, including search and information retrieval, simulation, constraints, objects as editable documents, and others.

The constraint language described here is a trial design -- a step towards integrating constraints into a full language. In the present document, the author has adopted an evolutionary approach that starts with the current Smalltalk messages and methods and ThingLab constraints, and works forward from that.

Entities of the Constraint Language

Like Smalltalk and ThingLab, the constraint language would be object-oriented. As a starting point, the language can use the current ThingLab inheritance and part-whole hierarchies. Constraints would be built into the language in a fundamental way. A generalization of the current constraint representation is proposed that allows constraints to be active or inactive, and to involve time and sequencing. Later in this chapter, a way of representing Smalltalk methods as constraints is described. This is an important step, as it serves to unify constraints with standard programming constructs.

The reader should think of programming in the constraint language as being like building up networks of constraints, as in the calculator example in Chapter 2. The syntax of the language can involve linear text, as in other languages. However, when the expression $1.8 * C + 32.0$ appears in a piece of code, it would have the same *semantics* as the equivalent network of constants, variables, and arithmetic operators. Thus, $+$ is not merely an uninterpreted atomic

symbol; rather, it is the name of a complete object in its own right. This object would have attachers that seek to merge with other objects (numbers), just like the Plus object in the calculator examples. Also like the Plus object, it would know about its inverses.

Derivation of Procedural from Declarative Information

In ThingLab, the user must enter into the system both the declarative and procedural parts of a constraint, where the declarative part is the constraint's rule, and the procedural part is the set of methods that can be invoked to satisfy the constraint. It is the user's responsibility to ensure that these parts correspond, and that the methods do what is expected.

In the constraint language, normally the user would enter the rule portion of a constraint only. For example, to enter a constraint that a Centigrade temperature C corresponded to a Fahrenheit temperature F , the user would need only enter the rule $1.8 * C + 32.0 = F$. The system would take care of constructing the methods that the constraint could use to satisfy itself, namely $F \leftarrow 1.8 * C + 32.0$ and $C \leftarrow (F - 32.0) / 1.8$.

In this case, the problem of constructing these methods may be solved using the one-pass constraint satisfaction techniques. The network implied by the above constraint rule corresponds to the network constructed by the user in the temperature converter example in Chapter 2. In that example, the constraint satisfier generated a plan for satisfying the constraints when the user changed C or F by using the one-pass constraint satisfaction techniques and the information owned by the arithmetic operators. These same techniques would yield the desired constraint methods in the new language.

In other cases, the system would not be able to find a one-pass ordering for solving the constraint satisfaction problem. Improved constraint satisfaction techniques would help with this to some extent. However, there will doubtless be cases for which the compiler would be unable to derive the procedural parts of the constraint from its rule. For example, unless it were equipped with some rather sophisticated algebraic problem-solving techniques, the constraint language compiler will probably be unable to derive Newton's method for finding a square root given only the constraint $y = x * x$.

Therefore, the language should include facilities that allow the user to provide the system with the procedural parts of the constraint. It would be nice if the system could check to see that these procedures did what was expected of them. In general, this is the program verification problem, and is hence quite hard. However, less ambitious kinds of checking could be accomplished more readily. First, there could be limited compile-time checking. For read-only methods, e.g. a *print* method, the compiler could check that the method didn't alter any parts of the object. For methods that did affect the object, the compiler could check that it altered only the appropriate parts. Second, the compiler could have a debugging mode, in which it would

insert a run-time check to see that a constraint was satisfied after a user-supplied method had been invoked. After a constraint had been used for a while, the checks could be removed.

Virtual Parts

As discussed in Chapter 3, Smalltalk makes a strong distinction between the insides and the outsides of an object. The internal aspects of an object are its class, and its instance fields and their contents; its external aspects are the messages that it understands and its responses. In Smalltalk one can thus have *virtual fields*. From the outside, it appears that an object has some local storage that can be read and written via appropriate messages; but internally there is no corresponding instance field.

To allow ThingLab to take advantage of this in describing part-whole hierarchies, a facility for defining virtual parts should be added. It would be rather simple to implement. In analogy to the class PartDescription, there would be a class VirtualPartDescription that would keep track of the virtual part's constraints. When a virtual part was involved in some constraints to be satisfied, the object that owned the part would inform the constraint satisfier of its existence. In generating code, the constraint satisfier would allocate a temporary variable for the virtual part and initialize it; whenever the virtual part was referenced in a method, the constraint satisfier would substitute a reference to the temporary for messages to the object asking for its part. As described in Chapter 3, an example of such a virtual part might be the center of a rectangle, which need not be explicitly stored in the instance, but can always be computed from the rectangle's corners.

Virtual parts will be used in the proposed way of dealing with constraints on time. However, they would have other applications as well. First, they would provide a convenient way for handling multiple representations of objects. For example, the multiply represented point described in Chapter 3 could have its Cartesian representation as an actual part, and the corresponding polar representation as a virtual part. From the outside, both parts would look the same. Also, virtual parts would be useful in storing data more efficiently when so desired. As an extreme example, consider the class HorizontalIsoscelesTriangle described in Chapter 3. As represented there, it has three superclass parts: a triangle, a horizontal triangle, and an isosceles triangle. Each triangle has pointers to the three sides (which are shared). The sides in turn have pointers to the endpoints, which finally hold the x and y coordinates. This object could be stored very efficiently by having six integers as its actual parts, with all the other parts being *virtual*.

Time

As a basis for a programming language, the most important missing aspect of the current constraint mechanism is that it has no facilities for dealing with *time*. To remedy this, a number of new mechanisms are proposed. These mechanisms are intended to be general enough to describe both constraints on time in simulations, and constraints on sequencing and iteration such as one might use in ordinary programming.

The semantics of constraints would be generalized. A constraint would have the following form:

when condition [body]

The condition would be a Boolean-valued expression that determines when the constraint is active. The body would consist of further constraints. The condition could be omitted; if so, the condition *when true* would be implied.

As in Simula, an object would have active phases, separated by periods of inactivity. An active phase of an object is called an *event*. An event happens in zero time (at least as far as any temporal constraints are concerned). In the new language, one way in which an event could occur would be for the value of the condition of one of the object's constraints to become true. Such events could be triggered either by external stimuli, such as the user's pressing a button, or by constraints whose conditions referenced a changing part, such as the value of a clock.

The object would have to satisfy all the other constraints in the body of the active constraint as long as the constraint's condition remained true. Using constraints as they have been defined thus far, as soon as the condition became true, the object might have to change its state to satisfy its constraints; but until the value of one of the conditions of its constraints changed, the object's state would not alter. However, a constraint could itself trigger a succession of events with the use of *temporal adverbs*. A preliminary list of such adverbs is:

first - the object's state when the constraint first becomes active

current - the object's current state

next - the object's state after the next event

last - the object's state when the constraint first becomes inactive

These adverbs would be messages understood by all objects. Thus, *next* is a message meaning "return your state after the next event". *next* would be something like a virtual part ("virtual successor" might be a better description). Temporal adverbs could be used in paths, e.g. *poolGame eightBall location first*. As long as the constraint's condition held, the object would take on the successive states that satisfied the *next* conditions described in the constraint's body. Thus, with the use of *current* and *next*, constraints could specify implicit looping. For example, as long as the constraint

i next = i current + 1

were active, the value of *i* would continue to be incremented. [This technique for defining iterations is

much like that used in the language Lucid (Ashcroft & Wadge 1977).]

If an object had several constraints that were active simultaneously, the *next* state of the object would have to satisfy all of the constraints. Otherwise, if two objects that were not related by constraints were active simultaneously, the events of each object would occur asynchronously.

There are various problems that need to be worked out in this formulation. For example, the constraint satisfaction techniques would need to be augmented so that they could deal with induction. Also, there are some timing issues in connection with asynchronous events. At the moment, the author does not regard these problems as insoluble. However, none of this has been implemented, so it remains to be seen if this is the case.

An Example of Constraints Involving Time

As a simple example of constraints involving time, a clock for use in building digital logic simulations will be described.

Class LogicClock

Superclasses

LogicDevice

Part Descriptions

state: a LogicState

active: a Boolean

Constraints

when active=true

[state first = off;

state next = state current inverse]

The driving force behind a logic simulation is the clock. As long as *active* is true, the logic clock will provide a steady stream of timing pulses. [Note: *inverse* is a message understood by instances of LogicState that means "if you are *on*, return *off*; if you are *off*, return *on*." The semicolon is used here to separate a series of constraints that must hold simultaneously.] One can picture an instance of LogicClock in the following way:

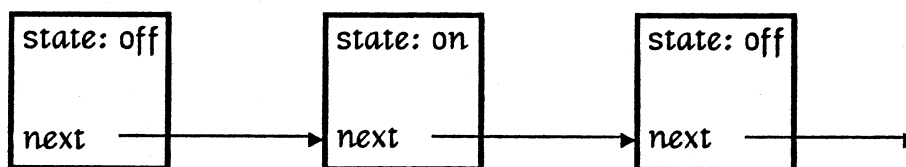


Figure 6.1 - Successive states of a logic clock

The initial state of the clock is *off*. This state has a virtual successor (that object obtained by sending it the message *next*), whose state is *on*, and so forth.

Smalltalk Methods as Constraints

The above mechanisms for virtual parts and for constraints dealing with time provide a way in which Smalltalk methods can be viewed as constraints. This is important, since it serves to unify the constraint metaphor with the metaphor of Smalltalk programming. However, the formulation to be presented is *not* the way that Smalltalk programs should eventually be pictured; rather, it is an interim formulation to help make a bridge between the continuous nature of constraints and the step-by-step nature of methods.

In Smalltalk-76, when an object receives a message, the message selector is matched against the selectors of the object's methods. If a match is found, the actual arguments of the message are bound to the formal arguments of the method, and space is allocated for the temporary variables. The code is then executed, causing further messages to be sent or assignments to be made. A method may be viewed as a constraint in the following way: the object that has the method should have a number of virtual parts, one with the same name as the selector, and others with names corresponding to those of the formal parameters. There is a constraint whose condition is that the virtual part that is the selector have the value true. The rule for this constraint should accomplish the same thing as the old method. Also, it should specify that the *next* value of the virtual part that is the selector be false, so that when the constraint becomes active, it causes the desired effects and then turns itself off. To send a message to an object, the actual arguments are placed in the corresponding virtual parts, and its virtual part that corresponds to the desired selector is set to true. [Small point: these virtual parts should really be regarded as virtual parts of the particular activation of an object, so that methods can be recursive.]

This formulation can handle the current message-method behavior, but is actually somewhat more general. Interesting results may be obtained by not automatically turning the constraint off, thus allowing a number of methods to be active simultaneously. This notion of control should be thought of in analogy to digital logic circuits: there is a "control line" (i.e., the virtual part that represents the selector) that can be turned on and off. As long as the control line is on, a device (method) is active. Many devices can be connected to a control line; a device may manage the control lines of other sub-devices.

The logic clock definition can be re-written as follows:

Class LogicClock

Superclasses

LogicDevice

Part Descriptions

state: a LogicState

Methods

active

[state first = off;
state next = state current inverse]

The LogicClock should be thought of as having a virtual part *active*. The clock is turned on by someone invoking the method named *active*. [In these examples, *active* will be a method that starts up an object's default activity. Thus in this case, the default activity of a clock is to tick.]

A Rocket Ship

As another example of the use of this scheme, consider a simulated rocket. The rocket will have a local time frame, represented by a Clock object. [Many of the parts and constraints of the rocket ought to be inherited from a more general class such as PhysicalObject, but that is not the point of this example.]

Class Clock

Superclasses

Object

Part Descriptions

time: a Time

dt: a TimeChange

Methods

active

[time first = 0;
time next = time current + dt]

Class Rocket**Superclasses**

Object

Part Descriptions

location: a Point
 velocity: a Velocity
 acceleration: an Acceleration
 heading: a Direction
 mass: a Mass
 thrust: a Force
 clock: a Clock

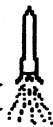
Methods**coast**

[thrust = 0.0]

fullSpeedAhead

[thrust = 1000.0]

show | icon

[icon = [thrust>0 ⇒ []];
 screen showIcon: icon at: location]

active

[clock active;
 location next = location current + (velocity*clock dt);
 velocity next = velocity current + (acceleration*clock dt);
 acceleration = heading*thrust/mass]

testFlight

[location first = screen center;
 heading first = up;
 time first = 0.0;
 velocity first = 0.0;
 mass = 1.0e6;
 clock dt = 0.001;
 self show;
 self fullSpeedAhead;
 self active]

The *active* method applies the laws of physics to the rocket, and simultaneously starts the clock ticking. The *testFlight* method sets up constraints that will initialize the rocket's state, and also invokes *active*.

An instance of Rocket could be created and tested by executing:

```
r ← Rocket new.  
when button down  
  [r testFlight].
```

As long as the button was held down, the rocket would fly.

Meta-Constraints

ThingLab currently uses certain kinds of constraints that specify how other constraints are to be processed; however, they are represented and used in an *ad hoc* manner. In the new language, meta-constraints would be constraints in their own right. The methods of these constraints would send messages to the satisfier, rather than to the object whose constraints were being satisfied. For example, a meta-constraint that a part be fixed would tell the constraint satisfier to leave that part alone. Other things that could be expressed in a natural way with the use of meta-constraints include specifying which method should be used to satisfy a constraint (if there is a choice); masking off a given constraint; indicating that a constraint is only a preference rather than a requirement; establishing partial orderings on preferences; explicitly representing the additional information used in constructing a method that does not uniquely determine the state of a part; and specifying heuristics as to what to try next in planning a constraint satisfaction method.

Since meta-constraints would themselves be constraints, there could be meta-meta-constraints, and so on. For example, given an inherited meta-constraint deleting another inherited (ordinary) constraint, one could include a meta-meta-constraint that deleted the deletion. However, use of this facility should probably be kept to a minimum in the interest of the sanity of the programmer.

Views

ThingLab has some facilities that support multiple views of an object. These facilities fall into two categories. First, the ThingLab user interface includes a menu of the different ways in which an object can depict itself. Second, the constraint mechanism allows the user to define two different representations of an object, and put a constraint on them that keeps them in coordination. For example, one can represent a point in both Cartesian and polar coordinates, and use a constraint to keep the two representations consistent. Similarly, one can define multiple views of an object to aid in constraint satisfaction, as in the quadratic example in Chapter 2 or the voltage divider in Chapter 5. [See also the discussion of the constraint language of Steele and Sussman in Chapter 1, and the *Views* example in Chapter 2. Picture transforms are a kind of view in common use in graphics systems. An early AI system that used multiple views was Merlin (Moore & Newell 1973). Views also play a prominent role in KRL (Bobrow & Winograd 1977a).]

There are, however, many questions that remain to be explored in this area. As presently implemented, the two kinds of multiple views described above are rather different. An object depicts itself on the screen procedurally -- there is a sequence of messages that it sends that causes bits to be changed on the bitmap display, but there is no separate object that corresponds to the picture as such. In the case of the point, however, there are really two objects. To make these cases fundamentally the same, ways of making pictures be actual objects should be investigated. Work on this problem is currently under way in the Learning Research Group. [Eventually, one might want to make the picture be a virtual object, in analogy with virtual parts; but its *semantics* should be as described.]

Another area for research concerns views that need not be always in coordination. Consider two views on a flip flop: as a logic device, and as an electronic circuit. Viewed as a logic device, the flip flop is fairly simple. When describing its state, one talks in terms of things being *on* or *off*. Viewed as an electronic circuit, it is more complex. It is composed of transistors, resistors, and so forth; in this case, its state will involve voltages, currents, and such. One can introduce constraints that relate the two representations -- for example, a constraint that a logic state of *off* must correspond to a voltage between 0 and 2, while for a logic state of *on* the voltage must be between 3 and 5. However, one might not want both views to be active at all times. When simulating the logic behavior of a device, the electrical representation could be turned off (perhaps by employing *when* conditions on the constraints that relate the two views). However, if one became interested in timing considerations, these electrical properties could be turned on again. These questions are also related to the use of part-whole hierarchies. For example, one might have the amplifier module of a radio described in terms of a simple relation between its input and output signals that holds under normal conditions, and also in terms of the details of the circuit. When appropriate, the simple model could be used; but when its conditions of applicability were violated (e.g., a drop in the power supply voltage or an input signal that was too large), the detailed representation could be used.

Such investigations might also yield a much cleaner view of what happens when a picture is edited and the depicted object is updated correspondingly. Consider the bar chart example in Chapter 2. When the user was in the midst of editing the text of one of the numbers, the constraint relating the number to the height of the bar was temporarily violated; only when the *accept* command was issued was the bar's height updated. Permission to temporarily violate the constraint is currently embedded in the methods of the text editor; the proposed investigations may provide ways of describing and reasoning about such things more formally. Another problem of this sort concerns one process that provides a view on another dynamically changing process. What happens when the viewing process can't keep up?

Another interesting area to be explored concerns cascaded views. In 3-d graphics work, picture transforms are often represented as 4x4 matrices. When a picture is to be subjected to a series of transforms, typically the transformation matrices are multiplied together in advance. In the

same way, successive views could be cascaded; regarding a view as a kind of constraint, this amounts to converting a network of constraints into a single constraint.

Views and Hierarchy

In his essay *The Architecture of Complexity* [Simon 1969], Herbert Simon describes what he calls *nearly decomposable systems*.

In hierarchic systems, we can distinguish between the interactions *among* subsystems, on the one hand, and the interactions *within* subsystems--that is, among the parts of those subsystems--on the other. The interactions at the different levels may be, and often will be, of different orders of magnitude. In a formal organization there will generally be more interaction, on the average, between two employees who are members of the same department than between two employees from different departments. In organic substances, intermolecular forces will generally be weaker than molecular forces, and molecular forces weaker than nuclear forces.

In a rare gas, the intermolecular forces will be negligible compared to those binding the molecules--we can treat the individual particles, for many purposes, as if they were independent of each other. We can describe such a system as *decomposable* into the subsystems comprised of the individual particles. As the gas becomes denser, molecular interactions become more significant. But over some range, we can treat the decomposable case as a limit and as a first approximation. We can use a theory of perfect gases, for example, to describe approximately the behavior of actual gases if they are not too dense. As a second approximation, we may move to a theory of *nearly decomposable* systems, in which the interactions among the subsystems are weak but not negligible.

At least some kinds of hierarchic systems can be approximated successfully as nearly decomposable systems. The main theoretical findings from the approach can be summed up in two propositions: (a) in a nearly decomposable system, the short-run behavior of each of the component subsystems is approximately independent of the short-run behavior of the other components; (b) in the long run, the behavior of any one of the components depends in only an aggregate way on the behavior of the other components.

In terms of the constraint language, there should be ways of viewing a complex part as a much simpler object. Dually, from within a part, there should be ways of viewing the influences of the environment in an aggregate way. As a very simple example, consider the thermometers example in Chapter 2. The Thermometers object has a part that is an instance of TemperatureConverter. From the outside, it should be possible to view the TemperatureConverter as a unit: the Times and Plus constraints, the constants, and their connectivity can all be collapsed into a single constraint. There is no need for the Thermometers object to know anything about the internal construction of its part. From within the TemperatureConverter, the outside environment impinges only at the two attachers, which in this case are connected to the thermometers. The TemperatureConverter could plan a way of satisfying its internal constraints for any state of its attachers, and synthesize a single constraint that embodies this view.

As a more complex example, in simulating planets orbiting the sun, from the outside each

planet can be approximated as a point mass. From the point of view of an individual planet, the influence of its environment can be summarized as a set of forces. As a good approximation, the environment is simply the sun; the attractions of the other planets are much weaker by comparison.

Some Implementation Considerations

Initially, the constraint language (and the next version of Smalltalk as well) could be implemented in Smalltalk-76. Eventually, however, the underlying environment could be changed to take advantage of the features of the new language. Some considerations of this sort are discussed in this section.

Storage Management

Suppose that the only kinds of direct pointers allowed in the system were pointers from a whole to its parts, or from an object to its successor (in a list, or its temporal successor as in the *next* message described previously). Further, suppose that all sharing of direct pointers had to be described by merges. Any other sort of reference would be stored as a path. If this were done, then the system could easily determine how many direct pointers there were to a given object. Storage management could be done without reference counting or garbage collection.

The disadvantage of this scheme is of course that the system would have to spend more time following paths. However, Smalltalk currently spends a considerable portion of its time on reference counting; and in the end, the new scheme might come out ahead. Also, specialized hardware could speed things up greatly. Aside from these considerations, this scheme has significant advantages in terms of control for constraint satisfaction purposes.

Parallelism

As noted in Chapter 5, in the current ThingLab the constraint satisfier could easily notice at compile-time when it is permissible for several constraints to be satisfied in parallel. In the proposed language, parallelism (or at least pseudo-parallelism) would be needed when two or more independent objects were active simultaneously. Therefore, one of the promising directions for future research would be to experiment with running the new language on a multiprocessor machine. The parallelism would involve an interesting mixture of compile-time and run-time techniques. Some of the parallelism could be completely planned at compile-time, with the constraint satisfier automatically generating code that told the machine which steps could be done in parallel, and when the various parallel forks had to rejoin. Other sorts of parallelism could be unravelled only at run-time, for example, the parallelism implied by several *when* conditions that depend on user-controlled input devices.

Bibliography

- [Ashcroft & Wadge 1977] Ashcroft, E. A., and W. W. Wadge, "Lucid, a Nonprocedural Language with Iteration," *Communications of the ACM*, July 1977, pp. 519-526.
- [Bobrow & Winograd 1977a] Bobrow, Daniel G., and Terry Winograd, "An Overview of KRL, A Knowledge Representation Language", *Cognitive Science*, V. 1, No. 1, 1977.
- [Bobrow & Winograd 1977b] Bobrow, Daniel G., Terry Winograd, and the KRL research group, "Experience with KRL-0: One Cycle of a Knowledge Representation Language", *Proceedings of the Fifth International Joint Conference on Artificial Intelligence*, 1977, pp. 213-222.
- [Borning 1977] Borning, Alan, "ThingLab -- An Object-Oriented System for Building Simulations Using Constraints", *Proceedings of the Fifth International Joint Conference on Artificial Intelligence*, 1977, pp. 497-498.
- [Brachman 1976] Brachman, Ronald, *What's in a Concept: Structural Foundations for Semantic Networks*, BBN Report No. 3433, Bolt Beranek and Newman, Cambridge, Mass, October 1976.
- [Bundy 1978] Bundy, Alan, "Will it Reach the Top? Prediction in the Mechanics World," *Artificial Intelligence* 10 (1978), pp. 129-146.
- [Dahl, Myhrhaug, & Nygaard 1970] Dahl, Ole-Johan, Bjorn Myhrhaug, and Kristen Nygaard, *Common Base Language*, Norwegian Computing Center Publication S-22, Oslo, Norway, October 1970.
- [Dahl & Nygaard 1966] Dahl, Ole-Johan, and Kristen Nygaard, "SIMULA -- An ALGOL-Based Simulation Language," *Communications of the ACM*, September 1966, pp. 671-677.
- [de Kleer 1975] de Kleer, Johan, *Qualitative and Quantitative Knowledge in Classical Mechanics*, MIT AI Lab Technical Report AI-TR-352, December 1975.
- [de Kleer & Sussman 1978] de Kleer, Johan, and Gerald J. Sussman, *Propagation of Constraints Applied to Circuit Synthesis*, MIT AI Lab Memo 485, September 1978.
- [Doyle 1977] Doyle, Jon, *Truth Maintenance Systems for Problem Solving*, M.S. Thesis, MIT, Cambridge, Mass., 1977.
- [Elcock *et al.* 1971] Elcock, E. W. *et al.*, "ABSET, a Programming Language Based on Sets:

- Motivation and Examples," in B. Meltzer and D. Michie (eds.), *Machine Intelligence 6*, Edinburgh: Edinburgh University Press, 1971, pp. 467-492.
- [Fikes 1970] Fikes, Richard, "REF-ARF: A System for Solving Problems Stated as Procedures", *Artificial Intelligence*, V. 1 No. 1, Spring 1970, pp. 27-120.
- [Goldberg & Robson 1979] Goldberg, Adele, and Dave Robson, "A Metaphor for User Interface Design," *Proceedings of the Twelfth Hawaii International Conference on System Sciences*, V. I, 1979, pp. 148-157.
- [Hewitt 1976] Hewitt, Carl, *Viewing Control Structures as Patterns of Passing Messages*, MIT AI Lab Memo 410, December 1976.
- [Ingalls 1978] Ingalls, Daniel H. H., "The Smalltalk-76 Programming System: Design and Implementation," *Conference Record of the Fifth Annual ACM Symposium on Principles of Programming Languages*, Tucson, Arizona, January 1978, pp. 9-16.
- [Kahn 1978] Kahn, Kenneth M., *Dynamic Graphics using Quasi Parallelism*, MIT AI Lab Memo 480, June 1978.
- [Kay 1972a] Kay, Alan, "A Personal Computer for Children of all Ages", *Proceedings of the ACM National Conference*, August 1972.
- [Kay 1972b] Kay, Alan, "A Dynamic Medium for Creative Thought", *Proceedings of National Council of Teachers of English Conference*, November 1972.
- [Kay & Goldberg 1977] Kay, Alan, and Adele Goldberg, "Personal Dynamic Media", *IEEE Computer*, March 1977, pp. 31-41.
- [Kay 1977] Kay, Alan, "Microelectronics and the Personal Computer", *Scientific American*, September 1977, pp. 230-244.
- [Liskov *et al.* 1977] Liskov, B. H., A. Snyder, R. Atkinson, and C. Shaffert, "Abstraction Mechanisms in CLU," *Communications of the ACM*, August 1977.
- [Moore & Newell 1973] Moore, J., and Newell, A., "How Can Merlin Understand?", in L. Gregg (ed.), *Knowledge and Cognition*, Baltimore, Md.: Lawrence Erlbaum Associates, 1973.
- [Newell & Simon 1972] Newell, Allen, and Herbert Simon, *Human Problem Solving*, Englewood Cliffs, N.J.: Prentice-Hall, 1972.
- [Rieger & Grinberg 1977] Rieger, Chuck, and Milt Grinberg, "The Declarative Representation and Procedural Simulation of Causality in Physical Mechanisms", *Proceedings of the Fifth International Joint Conference on Artificial Intelligence*, 1977, pp. 250-256.

- [Sacerdoti 1975] Sacerdoti, Earl D., "The Nonlinear Nature of Plans", *Proceedings of the Fourth International Joint Conference on Artificial Intelligence*, 1975, pp. 206-214.
- [Simon 1969] Simon, Herbert A., "The Architecture of Complexity," in *The Sciences of the Artificial*, Cambridge: MIT Press, 1969.
- [Stallman & Sussman 1977] Stallman, Richard M., and Gerald J. Sussman, "Forward Reasoning and Dependency-Directed Backtracking In a System for Computer-Aided Circuit Analysis," *Artificial Intelligence* 9 (1977), pp. 135-196.
- [Steele & Sussman 1978] Steele, Guy L., and Gerald J. Sussman, "Constraints," MIT AI Lab Memo 502, November 1978.
- [Sussman & Stallman 1975] Sussman, Gerald J., and Richard M. Stallman, "Heuristic Techniques in Computer-Aided Circuit Analysis," *IEEE Transactions on Circuits and Systems*, Vol. CAS-22 (11), November 1975.
- [Sutherland 1963] Sutherland, Ivan E., *Sketchpad: A Man-Machine Graphical Communication System*, Ph.D. thesis, MIT, Cambridge, Mass., 1963.
- [Tate 1977] Tate, Austin, "Generating Project Networks", *Proceedings of the Fifth International Joint Conference on Artificial Intelligence*, 1977, pp. 888-893.
- [Wilkes 1964] Wilkes, M. V., "Constraint-Type Statements in Programming Languages," *Communications of the ACM*, V. 7 No. 10, October 1964, pp. 587-588.
- [Winograd 1975] Winograd, Terry., "Frame Representations and the Declarative/Procedural Controversy," in Bobrow & Collins (eds.), *Representation and Understanding: Studies in Cognitive Science*, New York: Academic Press (1975).
- [Woods 1975] Woods, W. A., "What's in a Link: Foundations for Semantic Networks," in Bobrow & Collins (eds.), *Representation and Understanding: Studies in Cognitive Science*, New York: Academic Press (1975).
- [Wulf, London & Shaw 1976] Wulf, W. A., R. London, and M. Shaw, "An Introduction to the Construction and Verification of Alphard Programs," *IEEE Transactions on Software Engineering*, SE-2, 1976, pp. 253-264.
- [Yonezawa & Hewitt 1977] Yonezawa, Akinori, and Carl Hewitt, "Modelling Distributed Systems", *Proceedings of the Fifth International Joint Conference on Artificial Intelligence*, 1977, pp. 370-376.