A General Framework for Visualizing Abstract Objects and Relations

TOMIHISA KAMADA and SATORU KAWAI
University of Tokyo

Pictorial representations significantly enhance our ability to understand complicated relations and structures, which means that information systems strongly require user interfaces that support the visualization of many kinds of information with a wide variety of graphical forms. At present, however, these difficult visualization problems have not been solved. We present a visualization framework for translating abstract objects and relations, typically represented in textual forms, into pictorial representations, and describe a general visualization interface based on this framework. In our framework, abstract objects and relations are mapped to graphical objects and relations by user-defined mapping rules. The kernel of our visualization process is to determine a layout of graphical objects under geometric constraints. A constraint-based object layout system named COOL has been developed to handle this layout problem. COOL introduces the concept of rigidity of constraints in order to reasonably handle a set of conflicting constraints by use of the least squares method. As applications of our system, we show the generation of kinship diagrams, list diagrams, Nassi–Shneiderman diagrams, and entity-relationship diagrams.

Categories and Subject Descriptors: D.2.2 [**Software Engineering**]: Tools and Techniques—*user interfaces*; I.3.4 [**Computer Graphics**]: Graphics Utilities—*picture description languages*; I.3.6 [**Computer Graphics**]: Methodology and Techniques—*languages*

General Terms: Algorithms, Design, Theory

Additional Key Words and Phrases: constraints, constraint-based systems, graph drawing, graphics systems, layouts, pictorial representations, user interfaces, visualization

1. INTRODUCTION

It has been said that a picture is worth a thousand words; pictorial representations of information have played an important role in human communication and recognition since time immemorial. Humans can grasp the content of a picture much faster than they can scan and understand a text sentence because they have the ability to recognize the spatial configuration of elements in a picture and find the relationships between such elements quickly. Today, visual communication is also indispensable for the interface between humans and computers. Recent rapid progress in the technology of high-performance workstations

Authors' current addresses: T. Kamada, Research and Development, ACCESS Co., Ltd., Jinbocho Kyowa-Bld., 1-64 Kanda Jinbocho, Chiyoda-ku, Tokyo, 101 Japan; S. Kawai, Department of Graphic and Computer Science, College of Arts and Sciences, University of Tokyo, 3-8-1 Komaba, Meguro-ku, Tokyo, 153 Japan.

and personal computers with high-resolution bitmap displays has opened a new age of visual human–computer interaction. It is essential that the user interface of such information systems be able to present many kinds of information in a wide variety of graphical forms. This paper describes research on the visualization of information handled by information systems, which is a fundamental problem of visual communication.

It is usually expensive to implement a graphical presentation module (program) because a "good" pictorial representation depends heavily both on the data handled by the information system and on the purpose of the presentation. It is, therefore, challenging to realize general tools for data visualization. For numeric data, the visualization problem has been studied intensively, and business graphics packages have already spread widely. Recently, visualization in scientific computing has also been studied by many researchers. However, currently available graphics tools are as yet unable to provide high-level interfaces sufficient to visualize complicated abstract relations easily. When users want to represent data of a specific format in a specific graphical format, they must write a program for reading the data and for drawing the pictures they want by using a graphics library or a standard graphics package.[1] Existing graphics languages and systems do not help us translate original abstract relations into pictorial representations systematically, essentially because they only provide drawing instructions. This is very inefficient and error prone. The total visualization process, not merely the graphics system, should be studied. The goal of this research is to fill the gap between applications and graphics systems, and to explore a general approach to the visualization of abstract objects and relations.

A set of pictures (diagrams) can be viewed as a language, in the sense that picture elements (graphical objects) such as boxes, circles, and text strings are arranged under certain rules. For example, some rules are based on positional relationships such as listing order, listing direction, and distance. Others are indicated by lines and arrows that connect graphical objects. An essential criterion for good pictures is whether viewers find such layout rules intuitive and easily understand the underlying structures. Our idea for visualization is derived from this fact. That is, we generate a picture by mapping a relational structure to simple layout rules. This mapping, which we call *visual mapping*, is controlled by users.[2] In particular, we focus on diagrams structured by horizontal/vertical listings and explicit line connections.

We regard the visualization process as the translation from textual languages into two- or three-dimensional visual languages. We call this process *translation into pictures*. The translation process works as follows. First, the data generated or processed by an application is translated into a relational structure representation. What is to be visualized is the relational structure. In our approach, a relational structure (a semantic network) is regarded as a set of abstract relations

[1] For example, GKS (Graphical Kernel System) [24] and PHIGS (Programmer's Hierarchical Interactive Graphics System) [2] are available.

[2] We have already presented a graphics system (GRIP) for the visualization of shielding relations among 3D objects [32, 34]. The picturing function in GRIP maps shielding relations to drawing attributes. The picturing function can therefore be regarded as a visual mapping in a wide sense, although it handles geometric objects instead of abstract ones.

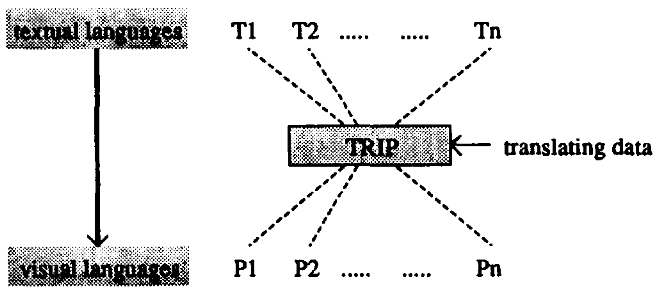


Fig. 1. TRIP, a general platform for visualization.

among abstract objects. Second, abstract objects are mapped to graphical objects, and abstract relations among them are mapped to graphical relations among the corresponding graphical objects. The graphical relations treated here are geometric relations, connection relations, and attribute relations. Finally, an actual layout of graphical objects is computed by solving graphical constraints and a picture is generated. In order to handle this layout problem, we have developed a layout system named COOL (COnstraint-based Object Layout system). COOL can handle over-constrained equation systems, which have been treated as errors by existing constraint-based graphics systems. In COOL, over-constrained systems are solved approximately by the least squares method.

In addition, we must take into account cases in which it is desired that network structures be literally visualized as network diagrams. As a special visual mapping, a technique for visualizing network structures generally is incorporated into our scheme.[3] In order to realize this, we have developed an algorithm for drawing general undirected graphs [31, 35]. We have implemented a visualization system named TRIP (TRanslation Into Pictures) as a prototype based on our visualization framework. TRIP is independent of both input textual languages and output visual languages (as Figure 1 illustrates) and can handle a wide range of visualization problems.

The remainder of this paper is organized as follows. Section 2 surveys work related to our research. Section 3 presents a general visualization framework. Section 4 describes our constraint-based object layout system in detail. Section 5 gives four practical applications of TRIP with the actual mapping rules and the pictorial results. We describe future work in Section 6 and conclude in Section 7.

2. RELATED WORK

Although the study of visualizing information has a long history, a general approach to visualization has not yet been developed. No study has directly influenced our whole work. However, some aspects of our research are related to previous and current work in various research areas. We categorize related work

[3] Displaying general views of various data handled in information systems besides network structures is necessary for the construction of good user interfaces. We have already handled and proposed a method for displaying general pictures of three-dimensional objects [33].

into the following three areas: information presentation, constraint-based systems and languages, and automatic graph drawing.

Information processed in computers is so various that information presentation problems have been studied in domain-dependent ways. For numeric data, many systems that automate business and statistical graphs have been developed [14, 60]. In [68], an interactive system that allows users to produce Cartesian plots, histograms, network diagrams, and pie charts is presented. Mackinlay presents research on automating the design of effective graphical presentations (such as bar charts, scatter plots, and connected graphs) of relational information [44]. Scientific visualization that enables scientists to observe their simulations and computations is a hot topic in computer graphics [48, 69]. However, general visualization tools for abstract relations have not yet been developed.

In what follows, we describe the research on information presentation specific to some areas. In the area of databases, there is some work on drawing Bachman diagrams and entity-relationship diagrams automatically [66, 73]. A graphical browser for entity-relationship databases is presented in [13]. The study of visualizing programs began with drawing flowcharts by computers [37]. Since then, many improved diagrams such as Nassi–Shneiderman diagrams [51], Chapin charts [17], Ferstl charts [25], and PAD (Problem Analysis Diagram) [26] have been developed. Recent advances in interactive graphics have encouraged programming in graphical environments, which is called visual or graphical programming. The existing work in this area is surveyed in [15, 16, 56, 61]. Algorithm animation is also related to visualization [4, 10, 11]. The design of information presentation is an important function of UIMSs (User Interface Management Systems). The work of Peridot [50] and Coral [65] should be noted as the systems that introduced constraints. In addition, we take note of knowledge-based information presentation systems [3, 75], which may provide the means of building a knowledge base of presentation data.

GRAFLOG is an interactive graphics interface in which drawings receive linguistic interpretations [53, 54, 55]. In GRAFLOG, graphical structures are mapped to logical structures by the translation function. In this respect, we can find a resemblance between TRIP and GRAFLOG. However, GRAFLOG is, if anything, a kind of visual language, and does not include the notion of constraints and layout facilities. Many of the above information presentation problems belong to the application domain of our system, TRIP. In fact, we handle the problem of generating such diagrams in Section 5.

Our research involves the design of a high-level graphical layout system called COOL, which uses constraints to specify the relationships among picture elements. The use of constraints in computer graphics is not a new idea. In the early 1960s, Sutherland's Sketchpad [64] pioneered the use of constraints in graphics systems. Borning's ThingLab [5, 6, 7] is a well-known graphical simulation laboratory based on constraints, and is similar to Sketchpad. In these systems, constraints are solved by local propagation, if possible, and otherwise by relaxation. ThingLab is extended to animation (Animus) by introducing temporal constraints [22]. Sussman and Steele describe another type of constraint system, which uses symbolic techniques for solving constraints, in [63]. The following are other representative constraint-based systems. Knuth's METAFONT [38] uses linear equations among the variables representing x- and

y-coordinates of points in the design of character shapes. Van Wyk's IDEAL [71] is a language for typesetting graphics which can solve "slightly nonlinear" equation systems. The constraint solver of IDEAL [20] attempts to reduce a set of constraints to a linear system by a variant of Gaussian elimination. Gosling's Magritte [27] is an interactive graphical layout system whose constraint-satisfaction mechanism is based on the algebraic transformation of constraints. Nelson's Juno [52] is an interactive constraint-based graphics system which uses Newton–Raphson iteration for solving constraints. Leler's Bertland [43] is a general-purpose constraint language based on augmented term rewriting. Vander Zanden proposes techniques for incrementally resatisfying constraints and applies them to graphical interfaces [72]. TK!Solver [39] is a commerically available system which uses constraints for general-purpose problem solving. Note that the idea of constraints has been introduced into spreadsheet systems like VisiCalc and its descendents. Our layout system is also related to the symbolic layout in VLSI design. Some constraint-based layout systems [46, 47, 58] have also been developed. In this area, a constraint refers to establishing a geometric relationship among the attributes of layout elements such as cells and wires. The most distinctive function of COOL, compared with the above systems, is that it can solve over-constrained linear equations by the least squares method. And, recently, constraint solving was introduced into traditional logic programming languages [19, 42, 59, 70]. As this trend shows, constraint-based programming is becoming a new programming paradigm.

In the abstract, information structures can be viewed as graphs. Basic algorithms for drawing nice graphs are required in many fields of computer science. The state of the art in automatic graph drawing is surveyed comprehensively in [23, 49, 67]. Many drawing algorithms have been developed for several classes of graphs—trees, planar graphs, acyclic digraphs, and general graphs. We have developed a spring algorithm for drawing general undirected graphs [31, 35]. The algorithm has many good properties: symmetric drawings of symmetric graphs, almost-congruent drawings of isomorphic graphs, uniform distribution of vertices, and a relatively small number of edge crossings. We have integrated this spring algorithm into TRIP.

3. TRANSLATION INTO PICTURES

We regard, and discuss, the visualization problem as the translation from textual into visual languages. As we have already mentioned, our purpose is to develop a general translation framework (Figure 1). In order to meet the requirement of generality, various kinds of translations must be taken into account. Indeed, there are an infinite number of textual languages, from simple syntactic languages to natural ones. Each application system usually has its own textual language, suitable for its particular objectives. These languages must be translated into a variety of visual languages (e.g., many variations of trees and networks).

3.1 Translation Process

In order to realize the independence of both original textual representations and target pictorial representations, we have introduced two additional representations intermediate between them. Figure 2 illustrates our visualization model.

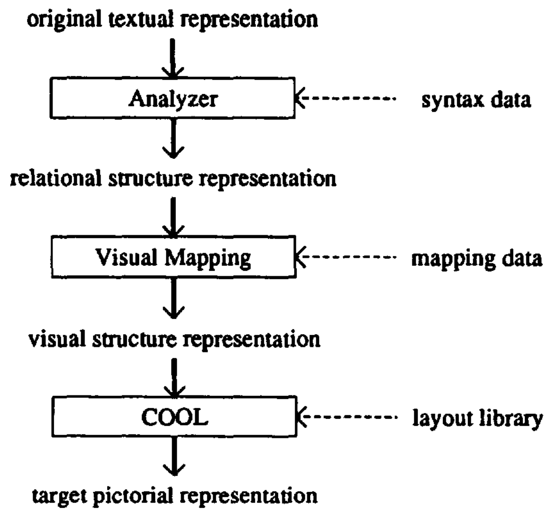

original textual representation
Analyzer
syntax data
relational structure representation
Visual Mapping
mapping data
visual structure representation
COOL
layout library
target pictorial representation

Fig. 2. Proposed visualization model.

The translation process proceeds through the following four representations.

(1) *Original textual representation.* Application programs or users deal with this representation of data. No specific representation is assumed. The input data for TRIP is a set of sentences of a textual language.

(2) *Relational structure representation.* This level of representation comprises a relational (semantic) structure derived from the textual representation. A relational structure is represented by abstract objects and relations. Different textual representations are translated into this universal representation.

(3) *Visual structure representation.* A visual structure expresses the relationships among the constituents in a picture, and is represented by graphical objects and relations. A relational structure is mapped to a visual structure by the use of user-defined mapping data.

(4) *Target pictorial representation.* This is the actual pictorial representation that users view. A pictorial representation is generated from the specification of a visual structure.

However, the original data may not be represented in textual form, but be stored in some internal form by an application. In this case we must transform the internal representation into the relational structure representation.

An analyzer parses a sentence and analyzes its semantics, and then generates the relational structure representation. Note that a set of sentences is translated into a set of underlying semantic relations, which can be regarded as a (semantic) network.[4] The introduction of the relational structure representation enables us to handle general structures so that we are not restricted to tree structures. Since

[4] We tried a grammar-directed approach based on constraint grammars by which a sentence is translated directly into a visual structure. Although this approach is an effective way of visualizing tree structures, it has a serious limitation in that it can essentially handle only tree structures [31].

analyzers are, of course, dependent on the original textual languages, different parsers are necessary for different languages. We can make use of parser generators such as YACC [30] to produce the parser for a given language. Moreover, if we obtain a parser for a natural language, we can translate the natural language into pictures. In Section 5 we show an example of the pictorial interpretation of English sentences. Since the responsibility for preparing an appropriate analyzer is, for the present, left to users we do not discuss this analysis phase further.

We have adopted Prolog to represent relational structures. In general, the relational structures we treat here are network structures similar to semantic networks. Prolog has sufficient expressive power to describe network structures and process patterns on networks [40]. Abstract objects and relations are represented by Prolog predicates. Graphical objects and relations are defined as special predicates that are the entries to COOL. The mapping from a relational structure to a visual one is also described as a Prolog program. For the present, users actually specify a visual mapping in the programming environment of Prolog. We plan to develop a high-level user interface for specifying the visual mapping easily. A set of predicates for a visual structure can be viewed as a program for COOL. The kernel of COOL, which is outside of Prolog, solves constraints and generates a picture.

We can build an analogy between our visualization and the machine translation of natural languages [62]. The usual translation process between natural languages consists of three successive phases: analysis (parsing), transfer, and synthesis (generation). The source language is analyzed into the underlying semantic representation of a sentence. Then it is mapped to a semantic representation specific to the target language. Finally, the corresponding target language sentence is generated from the representation. Our visualizing translation process consists of the phases corresponding exactly to analysis, transfer, and synthesis. This analogy justifies our claim that visualization is a sort of translation. In our model, the transfer phase, which we call visual mapping, plays the role of mapping a relational structure to a visual structure. It is one of the most important characteristics in our model that the transfer phase is separated both from the analysis phase and from the synthesis phase, and is designed to be controlled by users. COOL realizes the powerful synthesis phase in our model.

3.2 An Introductory Example of Visual Mapping

In our framework, visual mapping mediates between relational structures and visual ones. The conceptual model of visual mapping is presented in [31]. We realize this model by using Prolog and COOL. An abstract object is represented by a Prolog term or rule (predicate). An abstract relation is represented by a Prolog rule. These rules are asserted in the Prolog database as a result of the analysis phase. A visual structure is represented by the predefined special predicates, which generate constraints and drawing instructions.

We provide an example in order to give some guidelines for specifying the visual mapping. Consider a simple tree structure in which the mother is *diagram* and the daughters are *element*, *layout*, and *connection*. Figure 3 shows four pictures (generated by TRIP) representing this structure. Let the tree structure

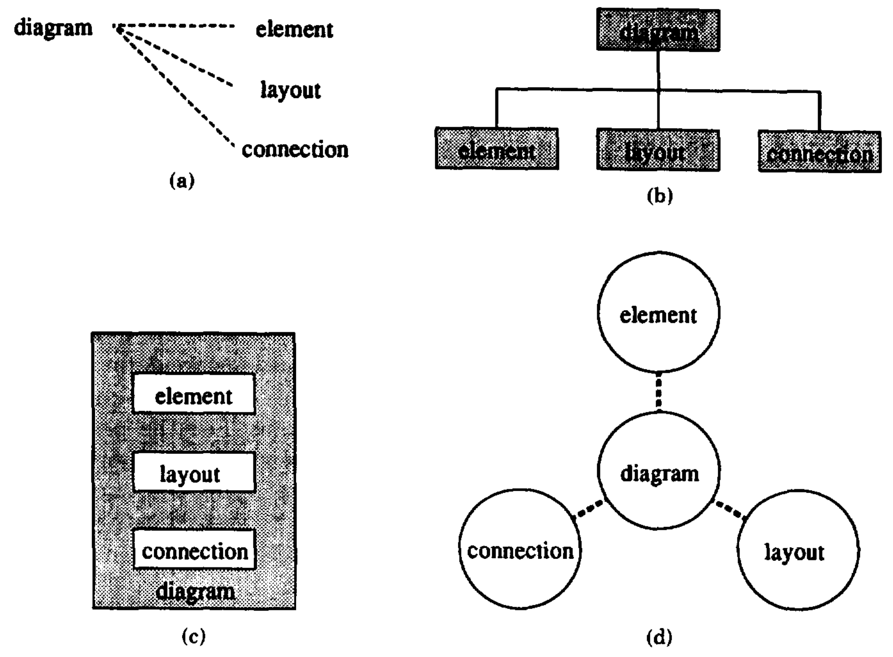


Fig. 3. Variations in drawing tree structures.

be described as the following Prolog predicates after the analysis phase:

consist_of(*diagram*, [*element*, *layout*, *connection*]).
is_object(*diagram*).
is_object(*element*).
is_object(*layout*).
is_object(*connection*).

The visual mapping for Figure 3a is explained as follows. The object expressed by the pattern

is_object(*X*)

is mapped to the box enclosing a text string, expressed by

boxwithlabel(*X*, *Width*, *Height*, *X*, [*invisible*]).

The five arguments of *boxwithlabel* are object name, box width, box height, box label, and drawing modes, respectively. The drawing mode *invisible* means that the box is not drawn. The relation expressed by the pattern

consist_of(*A*, [*H*/*L*])

is mapped to the following graphical relations:

horizontallisting([*A*, *H*], *Xgap*, [*rigid*]),
verticallisting([*H*/*L*], *Ygap*, [*rigid*]),
multi_connect([*A*], [*H*/*L*], *right*, *left*, [*dashed*, *straight*]).

The first relation places the object corresponding to *diagram* on the left of the object corresponding to *element* at an interval of *Xgap*. The next relation arranges the objects *element*, *layout*, and *connection* vertically from top to bottom at intervals of *Ygap*. The last relation connects the right side of the object *diagram* to the left sides of the objects *elements*, *layout*, and *connection* by dashed straight lines. We can obtain the picture in Figure 3b by changing the above mapping. The graphical object is to be changed to

boxwithlabel(X, Width, Height, X, [*visible*]).

The graphical relations are to be changed to

above([*A*], [*H/L*], *Ygap*, [*rigid*]),
horizontallisting([*H/L*), *Xgap*, [*rigid*]),
multi_connect([*A*], [*H/L*], *bottom*, *top*, [*solid*, *orthogonal*]).

These graphical objects and relations are supported by COOL, which computes a layout of objects from graphical relations and then generates a picture. Figures 3c and 3d are obtained by similar visual mappings. In Figure 3c, the mother-to-daughter relation is translated into the containment relation between boxes. In Figure 3d, three daughters are placed circularly around the mother with connecting lines.

In order to generate these pictures, users of TRIP have only to specify such translation rules as relate the patterns of network structures to graphical objects and relations. All that users have to do to change a pictorial form is to change the visual mapping.

4. A CONSTRAINT-BASED OBJECT LAYOUT SYSTEM

The picture-generation stage follows visual mapping in our visualization process. At this stage, a visual structure is translated into a real picture. We have developed a constraint-based object layout system named COOL in order to handle the problem of determining a graphical layout from the visual structure representation.

4.1 The COOL System

In conventional graphics systems, application programs or users generate a picture by placing graphics primitives at certain positions and then translating, rotating, and scaling them [2, 24]. However, in many cases, it is troublesome and nonessential to specify the exact positions of graphics primitives; users are often more interested in the relationships among primitives than in the exact positions and orientations of individual primitives. The graphical layout problems treated here conform to such cases. In these cases, graphical relations among graphical objects should be used to determine a layout. In COOL, the positions of graphical objects are computed automatically from the specified graphical relations. Graphical relations are divided into geometric (positional) relations and drawing relations. Geometric relations are expressed as algebraic constraints among the variables that characterize the graphical objects. Line connections between objects and label drawings for objects and for connecting lines are supported as drawing relations.

In the course of specifying the visual mapping, users should concentrate on a global layout scheme rather than local or component-wise constraints. In other words, the problem of over- and/or under-constrained systems should be handled not by users but by the visualization system. Particularly conflicting constraints are induced when multidimensional relations are to be represented in a two-dimensional restricted space. COOL is designed to be able to handle such an over-constrained system. In COOL, constraints are divided into two types, those that must be satisfied exactly and those that may be satisfied approximately. We call the former "rigid" and the latter "pliable" constraints. Pliable constraints are solved approximately by the least squares method. Users can specify a complicated layout flexibly by associating important layout rules with rigid constraints and less important ones with pliable constraints.

Borning et al. [8] propose constraint hierarchies for handling a problem of over- or under-constrained systems. In the constraint hierarchy system, one level in the hierarchy completely dominates the next level. That is, a constraint hierarchy represents which constraints to be satisfied first in a dynamic environment. Our classification is, on the other hand, used to weight the errors of constraints. Weighted constraints are also introduced into later versions of the constraint hierarchy theory [9]. Although we have currently only two extreme classes (rigid constraints and pliable constraints), it is possible to divide pliable constraints into several classes of different weights. This could be implemented by multiplying the error of a constraint by its weighting factor in the least squares summation.

In addition, COOL introduces a picture hierarchy. A set of graphical objects related by graphical relations may be bracketed as a subpicture, which is a box enclosing them. Since a subpicture is treated as a graphical object, it can belong to another subpicture. The system solves constraints hierarchically by traversing the picture hierarchy bottom-up.

COOL consists of three parts: constraint solver, language interface, and graphics interface. The constraint solver is designed to be independent of the other two parts and to be extensible. Currently, constraints are restricted to be linear equalities. The language interface for Prolog is used in TRIP. The combination of logic programming with constraint-based layout techniques is powerful. The pattern-matching, backtracking, recursion, and multiple rules that Prolog provides make the layout specification concise. In the current implementation, the constraint solver in COOL is invoked after the Prolog program completely terminates, which means there is no feedback from the constraint solver to Prolog. Device drivers have been prepared for various graphics devices. The pictures in this paper were generated through a PostScript interface.[5]

4.2 Picture Generation Specification

Here, we outline picture-generation specification in COOL. We use Prolog's programming environment to specify picture generation. The users' task, when

[5] The coordinate system we use here is the default user space in PostScript [1]. The origin is located at the lower left corner, with the positive x axis extending horizontally to the right and the positive y axis extending vertically upward. The length of a unit along the x and y axes is $\frac{1}{72}$ of an inch, although the figures in this paper may be reduced.

generating a picture, is to generate graphical objects and to generate graphical relations among them by using special predicates for COOL. They have only to specify relative relations among objects, without concern for the order of relations, because the system computes a whole layout automatically from relative relations.

Before describing the functions of COOL, we present the fundamental principles of specifying layouts. We define several reference points for each graphical object and connect objects by relations among their reference points. Our layout specification is based on the following principles.

(1) Constrain the reference points of graphical objects to lie on a geometric curve (line, circle, ellipse, arc, spline) at equal intervals.
(2) Connect the reference points of graphical objects by a geometric curve (line, arc, spline) with or without an arrowhead.

For the present, we have implemented horizontal, vertical, diagonal, and circular arrangements for principle (1). In order to realize the arrangements along higher-order curves, we must introduce nonlinear constraints into COOL. As for principle (2), we have implemented line connections. Both straight line drawings and orthogonal line drawings are supported.

Graphical Objects

A graphical object is conceptually defined to consist of the following three sets:

—a set of variables,
—a set of constraints,
—a set of drawing instructions.

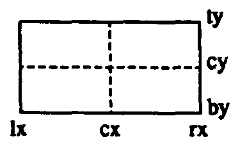
Variables express the geometric attributes (such as position and size) of an object. Constraints express algebraic relationships among these variables. Drawing instructions express how to draw an object. They are sent to the graphics interface with the actual values of variables after the constraints are solved. Graphical objects in COOL are similar to boxes in IDEAL [71].

As examples, graphical objects BOX and CIRCLE are defined as Prolog predicates in Figure 4. A term is mapped to a box of given width and height by *box*, and is also mapped to a circle of given radius by *circle*. BOX and CIRCLE have six internal variables, respectively, which are related to one another by intra-constraints. These variables express representative x- or y-coordinates of an object, as illustrated in Figure 4.

The predicates *map*, *constraint*, and *drawing*, which appear in the definitions of BOX and CIRCLE, play the role of interface between Prolog and the COOL kernel. A term is mapped to a unique graphical object by *map*. Constraints and drawing instructions are generated by *constraint* and *drawing*, respectively. For example, *box*(*a*, 40, 20, [*bound*]) generates the following constraints:

$$\begin{aligned} &a.cx = (a.lx + a.rx)/2 \\ &a.cy = (a.by + a.ty)/2 \\ &a.lx + 40 = a.rx \\ &a.by + 20 = a.ty \end{aligned}$$

```
% BOX( term, width, height, mode )
%     variables:   lx, rx, by, ty, cx, cy
%     available modes:  invisible/bound/fill/visible(default)
%
box( Term, Width, Height, Mode ) :- map( Term, Object ),
      constraint( "$1.cx = ($1.lx + $1.rx)/2", [Object] ),
      constraint( "$1.cy = ($1.by + $1.ty)/2", [Object] ),
      constraint( "$1.lx + $2 = $1.rx", [Object, Width] ),
      constraint( "$1.by + $2 = $1.ty", [Object, Height] ),
      member( invisible, Mode ) -> map( Term, Object ),
          drawing( "box( $1.id, $1.lx, $1.rx, $1.by, $1.ty, invisible)", [Object] );
      member( fill, Mode ) -> map( Term, Object ),
          drawing( "box( $1.id, $1.lx, $1.rx, $1.by, $1.ty, fill)", [Object] );
      member( bound, Mode ) -> map( Term, Object ),
          drawing( "box( $1.id, $1.lx, $1.rx, $1.by, $1.ty, bound)", [Object] );
                              map( Term, Object ),
          drawing( "box( $1.id, $1.lx, $1.rx, $1.by, $1.ty, visible)", [Object] ).
```



```
% CIRCLE( term, radius, mode )
%     variables:   lx, rx, by, ty, cx, cy
%     available modes:  invisible/bound/fill/visible(default)
%
circle( Term, Radius, Mode ) :- map( Term, Object ),
      constraint( "$1.lx = $1.cx - $2", [Object, Radius] ),
      constraint( "$1.rx = $1.cx + $2", [Object, Radius] ),
      constraint( "$1.by = $1.cy - $2", [Object, Radius] ),
      constraint( "$1.ty = $1.cy + $2", [Object, Radius] ),
      member( invisible, Mode ) -> map( Term, Object ),
          drawing( "circle( $1.id, $1.cx, $1.cy, $2, invisible)", [Object, Radius] );
      member( fill, Mode ) -> map( Term, Object ),
          drawing( "circle( $1.id, $1.cx, $1.cy, $2, fill)", [Object, Radius] );
      member( bound, Mode ) -> map( Term, Object ),
          drawing( "circle( $1.id, $1.cx, $1.cy, $2, bound)", [Object, Radius] );
                              map( Term, Object ),
          drawing( "circle( $1.id, $1.cx, $1.cy, $2, visible)", [Object, Radius] ).
```

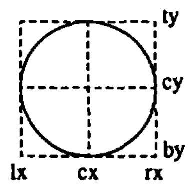


Fig. 4. Definitions of graphical objects BOX and CIRCLE. The predicates *constraint* and *drawing* put the first argument (string) on the output stream. The special symbol $\$n$ in the string is replaced by the nth item in the list specified as the second argument.

and generates the following drawing instruction:

$$box(a.id,\ a.lx,\ a.rx,\ a.by,\ a.ty,\ bound).$$

It is easy to add new objects to COOL by using these predicates, although only five objects BOX, DIAMOND, CIRCLE, ELLIPSE, and POINT are supported in the current implementation.

Geometric Relations

A geometric relation among graphical objects is expressed as extra-constraints among the variables of objects. COOL provides a variety of geometric relations, such as alignment and ordering relations for x- or y-coordinates.

A relation should be generic in the sense that it can be uniformly applied to all kinds of graphical objects. In order to realize the generic property of relations, we introduce some geometric attributes that are recognized to be common to all graphical objects. They are represented by variables of the same name so that they can be referred to by graphical relations in an object-independent way.[6] For example, consider the relation HORIZONTAL, which constrains objects to lie on a horizontal line in Figure 5. It refers to the variables *ty*, *by*, or *cy* depending on the alignment mode. Thus, *horizontal*([*a*, *b*, *c*], [*top_align*]) generates the following constraints:

$$a.ty = b.ty$$
$$b.ty = c.ty.$$

Note that a HORIZONTAL relation can be applied both to BOX and to CIRCLE, and in fact to any object with the variables *ty*, *by*, and *cy*. Figure 5 illustrates a HORIZONTAL relation between a box and a circle in three alignment modes.

Other geometric relations such as vertical, circular, and diagonal listings are defined in the same way. The two modes *rigid* and *pliable*, which represent the strength of the constraints, are commonly available for all geometric relations. We show later how to use these modes. In addition, COOL provides a special geometric relation, HIDE, to deal with the overlapping problem that arises when drawing pictures in a two-dimensional plane. A hiding object obscures the other in the picture.

Drawing Relations

A drawing relation among graphical objects generates a drawing instruction that refers to the variables of the objects. Drawing relations do not influence the procedure for determining the positions of objects because they generate no constraints. The drawing relations currently available are label drawings and line connections. These relations are defined to be generic.

A predicate *label* is prepared for drawing a textual label associated with a graphical object. A text string is placed inside the specified object depending on the mode (the third argument of *label*), which takes one of *top*, *bottom*, *left*, *right*, and *center*. The real position of a label is determined according to the variables of an object (e.g., *lx* and *cy* in *left* mode). Figure 6a illustrates the five cases for labeling a box. A textual label can also be drawn associated with a connecting line. A label is placed near the midpoint of a line and on the upper side of it (on the left side in the case of a vertical line). As illustrated in Figure 6b, *connectwithlabel* draws a line with a label at its midpoint. In this example, POINT objects are also labeled.[7]

[6] There are, of course, object-specific relations such as adjoining relations in an arbitrary direction. Although these relations are required in some kinds of layouts, we have not added them to the system so far.

[7] It may seem tricky that the label specified in *left* mode is placed on the right of the point. The label is placed on the right of the point (lx, cy) in *left* mode irrespective of object types. In the case of a POINT object, (lx, cy) coincides with the POINT itself because the variables of a POINT object are constrained to satisfy equations $lx = rx = cx$ and $ty = by = cy$.

```
% HORIZONTAL( [term, term, ... ], mode ).
%     available modes:  top_align/bottom_align/center_align(default)
%                       pliable/rigid(default)
horizontal( [_], _ ).
horizontal( [Left, Right | List], Mode ) :-
        hor2( Left, Right, Mode ), horizontal( [Right | List], Mode ).
hor2( Left, Right, Mode ) :-
        member( top_align, Mode ) -> hor2t( Left, Right, Mode );
        member( bottom_align, Mode ) -> hor2b( Left, Right, Mode );
        hor2c( Left, Right, Mode ).
hor2t( Left, Right, Mode) :- maplist( [Left, Right], Objects ),
        constraint( "$1.ty = $2.ty", Objects, Mode ).
hor2b( Left, Right, Mode) :- maplist( [Left, Right], Objects ),
        constraint( "$1.by = $2.by", Objects, Mode ).
hor2c( Left, Right, Mode) :- maplist( [Left, Right], Objects ),
        constraint( "$1.cy = $2.cy", Objects, Mode ).
```

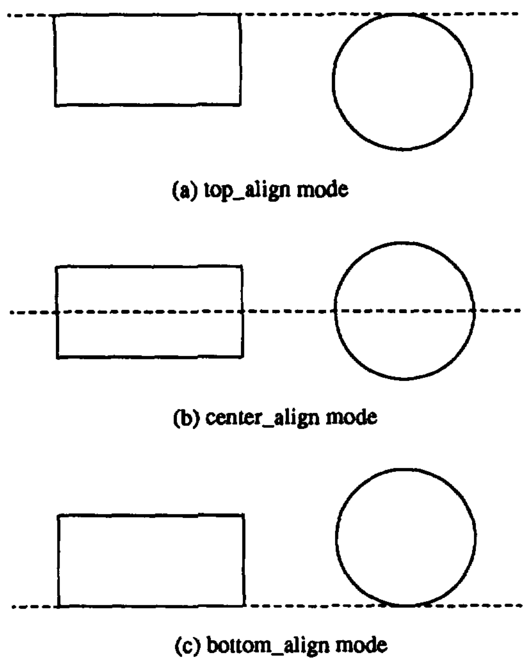
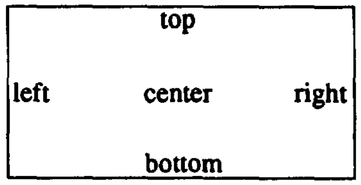
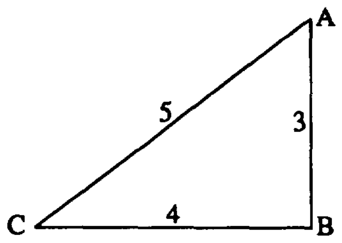


Fig. 5. Definition of a HORIZONTAL relation.

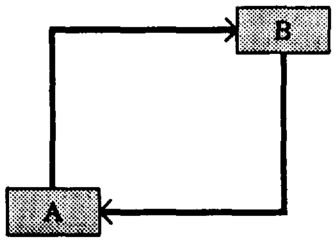
Graphical objects can be connected by a line (*connect*) or an arrow (*arrow*). The connecting point of an object is specified as one of *top*, *bottom*, *left*, *right*, and *center*. The real position of a connecting point is determined by referring to the variables of an object as the position of a label is determined. The drawing styles *straight* and *orthogonal* are provided. In addition, two modes for line width (*thin* and *thick*) and three line types (*solid*, *dashed*, and *dotted*) are available. The default mode is *straight*, *thin*, and *solid*. In orthogonal drawings, the route



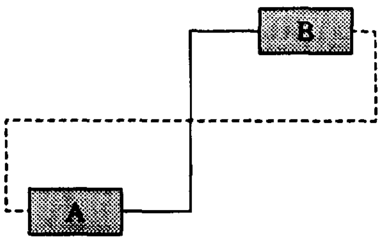
```
(a)  box( object, 150, 75, [bound] ),
     label( object, 'top', [top] ),
     label( object, 'bottom', [bottom] ),
     label( object, 'left', [left] ),
     label( object, 'right', [right] ),
     label( object, 'center', [center] ).
```



```
(b)  point( a ), label( a, 'A', [left] ),
     point( b ), label( b, 'B', [left] ),
     point( c ), label( c, 'C', [right] ),
     connectwithlabel( a, b, center, center, [straight], 3 ),
     connectwithlabel( b, c, center, center, [straight], 4 ),
     connectwithlabel( a, c, center, center, [straight], 5 ).
```



```
(c)  box( a, 40, 20, [visible] ),
     label( a, 'A', [center] ),
     box( b, 40, 20, [visible] ),
     label( b, 'B', [center] ),
     arrow( a, b, top, left, [orthogonal, thick] ),
     arrow( b, a, bottom, right, [orthogonal, thick] ).
```



```
(d)  box( a, 40, 20, [visible] ),
     label( a, 'A', [center] ),
     box( b, 40, 20, [visible] ),
     label( b, 'B', [center] ),
     connect( a, b, right, left, [orthogonal, solid] ),
     connect( a, b, left, right, [orthogonal, dashed] ).
```

Fig. 6. Label drawings and line connections.

of a polygonal line or arrow is selected automatically depending on the connecting points of objects, as illustrated in Figures 6c and 6d.

A graphical layout is specified as a set of such objects and relations. Figure 7 lists the basic predicates for COOL in the current version. To make the specification concise, useful high-level relations are provided. They are defined as

1. Graphical Objects.

box(term, width, height, mode)	% BOX object
diamond(term, width, height, mode)	% DIAMOND object
circle(term, radius, mode)	% CIRCLE object
ellipse(term, xradius, yradius, mode)	% ELLIPSE object
point(term)	% POINT object

2. Geometric Relations.

x_relation(t1, t2, ref1, ref2, gap, mode)	% t1.(ref1) + gap = t2.(ref2)
y_relation(t1, t2, ref1, ref2, gap, mode)	% t1.(ref1) + gap = t2.(ref2)
x_order([t1, t2, ...], xgap, mode)	% t1.rx + xgap = t2.lx .....
y_order([t1, t2, ...], ygap, mode)	% t1.by - ygap = t2.ty .....
x_average(t1, [t2, ...], mode)	% t1.cx = (t2.cx + ...) / n
y_average(t1, [t2, ...], mode)	% t1.cy = (t2.cy + ...) / n
horizontal([t1, t2, ...], mode)	% t1.cy(ty,by) = t2.cy(ty,by) .....
vertical([t1, t2, ...], mode)	% t1.cx(lx,rx) = t2.cx(lx,rx) .....
contain(t1, t2, gap, mode)	% t1 contains t2
hide(t1, t2)	% t1 hides t2

3. Drawing Relations.

label(term, label, mode)	% label drawing
connect(t1, t2, from, to, mode)	% t1 is connected to t2 by a line
arrow(t1, t2, from, to, mode)	% t1 is connected to t2 by an arrow
connectwithlabel(t1, t2, from, to, mode, label)	% connect with a label
arrowwithlabel(t1, t2, from, to, mode, label)	% arrow with a label

4. High-level Relations.

boxwithlabel(term, width, height, label, mode)	% box & label
diamondwithlabel(term, width, height, label, mode)	% diamond & label
circlewithlabel(term, radius, label, mode)	% circle & label
ellipsewithlabel(term, xradius, yradius, label, mode)	% ellipse & label
pointwithlabel(term, label, mode)	% point & label
horizontallisting([t1, ...], gap, mode)	% horizontal & x_order
verticallisting([t1, ...], gap, mode)	% vertical & y_order
circularlisting(t1, [t2, ...], mode)	% t2, ... are on a circle around t1
diagonallisting([t1, ...], xgap, ygap, mode)	% x_relation & y_relation
above([t1, ...], [t2, ...], gap, mode)	% x_average & y_order
below([t1, ...], [t2, ...], gap, mode)	% x_average & y_order
leftof([t1, ...], [t2, ...], gap, mode)	% y_average & x_order
rightof([t1, ...], [t2, ...], gap, mode)	% y_average & x_order
between(t1, t2, t3, mode)	% x_average & y_average
multi_connect([t1, ...], [t2, ...], from, to, mode)	% connect t1, ... to t2, ...
multi_arrow([t1, ...], [t2, ...], from, to, mode)	% connect t1, ... to t2, ...
multi_hide([t1, ...], [t2, ...])	% t1, ... hide t2, ...

Fig. 7. List of predicates for COOL.

combinations of low-level relations. Figure 8 shows a diagram generated by COOL. Many BOX objects are laid out under various graphical relations. In this example, 106 constraints among 106 variables are solved. COOL is implemented on a UNIX® workstation (CPU, MC68020). The CPU time needed to solve the

® UNIX is a trademark of AT&T Bell Laboratories.

```
%     diagram, text, structure, connection, layout, and element
boxwithlabel( o, 60, 20, 'diagram', [fill] ), boxwithlabel( c1, 60, 20, 'text', [fill] ),
boxwithlabel( c2, 60, 20, 'structure', [fill] ), boxwithlabel( c3, 60, 20, 'connection', [fill] ),
boxwithlabel( c4, 60, 20, 'layout', [fill] ), boxwithlabel( c5, 60, 20, 'element', [fill] ),
circularlisting( o, [c1, c2, c3, c4, c5], [ ] ), verticallisting( [c1, o], 60, [ ] ),
multi_connect( [o], [c1, c2, c3, c4, c5], center, center, [dashed] ),
arrow( c1, c5, left, top, [orthogonal, dashed, thick] ), arrow( c3, c4, left, right, [solid, thick] ),
%     tree, planar graph, and general graph
boxwithlabel( s1, 80, 20, 'tree', [bound, bottom] ),
boxwithlabel( s2, 100, 40, 'planar graph', [bound, bottom] ),
boxwithlabel( s3, 120, 60, 'general graph', [bound, bottom] ),
contain( s3, s2, 0, [top_align, left_align] ), hide( s2, s3 ),
contain( s2, s1, 0, [top_align, left_align] ), hide( s1, s2 ),
horizontallisting( [c2, s3], 30, [ ] ), connect( c2, s3, right, left, [solid] ),
%     straight line and orthogonal line
boxwithlabel( v1, 70, 20, 'straight line', [invisible] ),
boxwithlabel( v2, 80, 20, 'orthogonal line', [invisible] ),
horizontallisting( [c3, v1], 30, [ ] ), verticallisting( [v1, v2], 10, [left_align] ),
multi_connect( [c3], [v1, v2], right, left, [orthogonal, solid] ),
%     horizontal, vertical, and circular
boxwithlabel( h1, 60, 20, 'horizontal', [invisible] ),
boxwithlabel( h2, 60, 20, 'vertical', [invisible] ),
boxwithlabel( h3, 60, 20, 'circular', [invisible] ),
horizontallisting( [h1, h2, h3], 20, [ ] ), above( [c4], [h1, h2, h3], 40, [ ] ),
multi_connect( [c4], [h1, h2, h3], bottom, top, [orthogonal, solid] ),
%     box, circle, and point
boxwithlabel( e1, 40, 20, 'box', [bound] ),
boxwithlabel( e2, 40, 20, 'circle', [bound] ),
boxwithlabel( e3, 40, 20, 'point', [bound] ),
verticallisting( [e1, e2, e3], 20, [ ] ), rightof( [c5], [e1, e2, e3], 30, [ ] ),
multi_connect( [c5], [e1, e2, e3], left, right, [dotted] ).
```

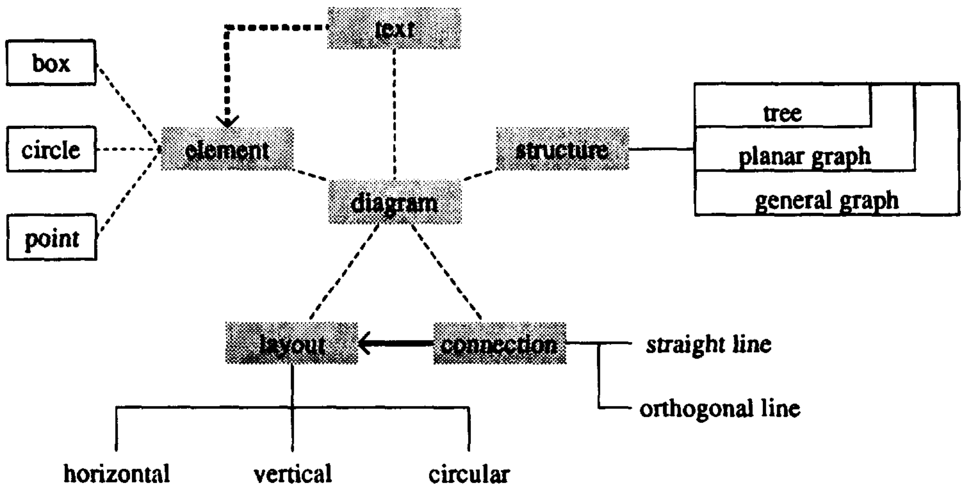


Fig. 8. A diagram generated by COOL.

above 106 constraints is about 0.8 second. Thus, with respect to running time, the bottleneck is the Prolog interpreter.

4.3. Advanced Features

COOL has some advanced features in addition to the basic layout facilities described above. In this section we explain how to use pliable constraints and the subpicture structure. Integration of a graph layout system into COOL is also described.

Pliable Constraints

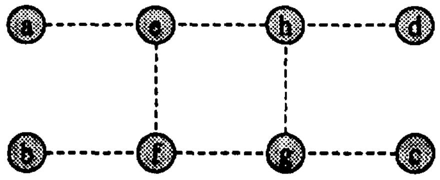
We introduce pliable geometric relations, expressed as pliable constraints, into COOL in order to solve conflicting constraints. Pliable relations can be mixed with rigid ones. Whether a geometric relation is rigid or pliable is specified as the mode *rigid* or *pliable*. (The default mode is *rigid*.) In COOL, conflicting pliable constraints are satisfied approximately by the least squares method, while rigid constraints are satisfied exactly. Pliable relations are very useful for our purposes. Over-constrained systems sometimes occur in our layout problems because it is difficult to map several abstract relations to nonconflicting two-dimensional geometric relations. In such cases we can make the constraint systems solvable by converting some of the rigid relations, corresponding to less important relations, into pliable ones.

Figure 9 illustrates an example of pliable geometric relations. Eight CIRCLE objects are constrained by four geometric relations that conflict with one another. Obviously, the constraints cannot be solved if all relations are rigid. Figures 9a, 9b and 9c show the cases in which some of the relations are pliable. In these cases the errors of pliable constraints are distributed only over the pliable constraints. Figure 9d shows the case in which all relations are pliable. The resultant layout is balanced by distributing the constraint errors over the whole picture.

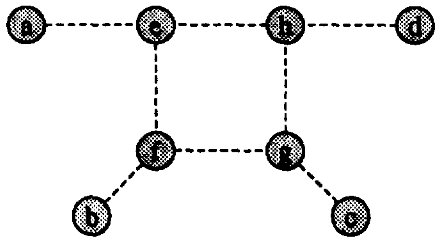
Picture Hierarchy

A common way of building up a picture or a layout is to divide it into parts, construct each subpart, and combine them. For example, many graphics systems provide a structure or segmentation facility by which graphics primitives are defined in a local coordinate space and are then transformed collectively by coordinate transformations [2, 24]. For example, a "block" in PIC [36] provides such a facility. A set of graphical objects in a block can be placed by referring to their bounding box in PIC. We introduce such a bounding box hierarchy into our constraint-based object layouts. In COOL, a set of graphical objects and a set of graphical relations among them can be treated as a subpicture. A subpicture is regarded as a bounding box that encloses its component objects. As a subpicture itself is also a graphical object, subpictures may be nested.

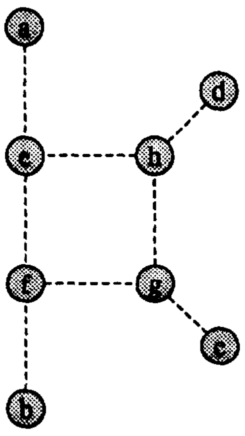
Two special predicates *pstart*(*name*) and *pend* are provided for defining a subpicture. The objects and relations specified between *pstart* and *pend* are collected in a subpicture. A subpicture can be referred to in graphical relations by its name. Figure 10 shows an example in which nested *tree* patterns are visualized as nested subpictures by the recursively defined visual mapping. The



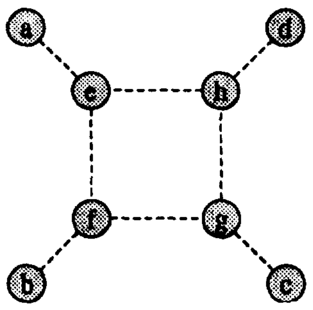
(a)
```
horizontallisting( [a, e, h, d], 40, [rigid] ),
horizontallisting( [b, f, g, c], 40, [rigid] ),
verticallisting( [a, e, f, b], 40, [pliable] ),
verticallisting( [d, h, g, c], 40, [pliable] ).
```



(b)
```
horizontallisting( [a, e, h, d], 40, [rigid] ),
horizontallisting( [b, f, g, c], 40, [pliable] ),
verticallisting( [a, e, f, b], 40, [pliable] ),
verticallisting( [d, h, g, c], 40, [pliable] ).
```



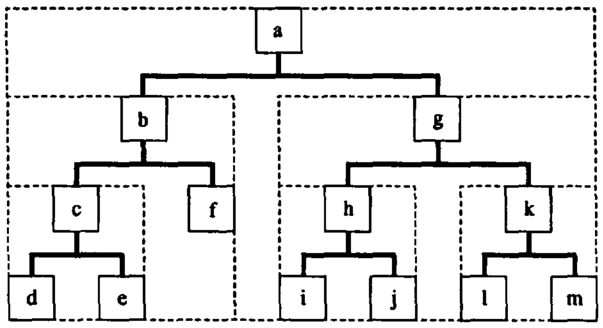
(c)
```
horizontallisting( [a, e, h, d], 40, [pliable] ),
horizontallisting( [b, f, g, c], 40, [pliable] ),
verticallisting( [a, e, f, b], 40, [rigid] ),
verticallisting( [d, h, g, c], 40, [pliable] ).
```



(d)
```
horizontallisting( [a, e, h, d], 40, [pliable] ),
horizontallisting( [b, f, g, c], 40, [pliable] ),
verticallisting( [a, e, f, b], 40, [pliable] ),
verticallisting( [d, h, g, c], 40, [pliable] ).
```

Fig. 9. An example of pliable constraints.

```
%  A hierarchical visual mapping for the tree structure.
v( tree( X, Y, Z ) ) :-
    pstart( tree( X, Y, Z ) ),
    v( Y ),
    v( Z ),
    v( X ),
    horizontallisting( [Y, Z], 25, [top_align] ),
    above( [X], [Y, Z], 25, [ ] ),
    multi_connect( [X], [Y, Z], bottom, root, [thick, orthogonal] ),
    reference( tree( X, Y, Z ), root, X, top ),
    pend.

v( X ) :-
    boxwithlabel( X, 25, 25, X, [bound] ),
    reference( X, root, X, top ).
```



v(tree(a, tree(b, tree(c, d, e), f), tree(g, tree(h, i, j), tree(k, l, m)))).

Fig. 10. A tree visualized by the picture hierarchy. The boxes bounded by dashed lines represent subpictures.

boxes bounded by dashed lines represent subpictures. In each subpicture, its subordinate subpictures are related geometrically by *horizontallisting* and *above* relations. A predicate *reference* is a special predicate for declaring a reference point of a subpicture. In this example, a reference point *root* is declared as *top* of a box in each subpicture. In COOL, such a recursive layout can be specified easily by the picture hierarchy.

Integration of a Graph Layout System

We have integrated a graph layout system into COOL; the positions of graphical objects can be determined by the use of our algorithm for drawing general undirected graphs. Geometric relations and drawing relations can be used together with this graph layout. It may be difficult, however, to use geometric relations in addition to graph layout constraints because users can hardly forecast the resultant layout generated by the graph layout system.

A special predicate *adjacent(obj, obj, weight)*, which represents the adjacency relation between two graphical objects, is provided as the entry to the graph layout system. The graph layout system is invoked before the constraint solver runs. The system collects these relationships among object and computes a layout. Then it generates positional data in the form of linear constraints of *cx* and *cy* variables of objects. The constraint solver in COOL solves these constraints together with the constraints generated by geometric relations. Figure 11 shows a network diagram whose layout is computed by the graph layout system. Many CIRCLE objects are placed in an esthetically pleasing way by the system. As this example shows, drawing relations such as label drawings and line connections can be used with the graph layout.

4.4 Constraint Satisfaction

It is difficult to handle constraint satisfaction generally. For example, even in the restricted case of linear constaints only, the constraint satisfaction problem is as difficult as linear programming if inequalities are included. Symbolic manipulation systems such as MACSYMA [45] and REDUCE [28] can be used as general tools for solving algebraic constraints.[8] However, these systems work less efficiently, owing to their generality. Accordingly, constraint-based systems usually have domain-specific techniques. At present, several constraint solving techniques are known [43]: local propagation, relaxation, graph transformation, a simple equation solver, and a technique based on augmented term rewriting.

Our technique is based on a simple equation solver [20, 71]. The basic idea of our algorithm follows. First, all constraints are converted into a normal form, called an ordered linear combination. Then, rigid constraints are eliminated in order by using Gaussian elimination.[9] Next, pliable constraints are solved approximately by the least squares method. Let $\mathbf{Ax} = \mathbf{b}$ be the matrix form of pliable constraints, when $\mathbf{A}$, $\mathbf{x}$, and $\mathbf{b}$ are the coefficient matrix, the variable vector, and the constant vector, respectively. It is known that the least squares solution of these simultaneous equations satisfies ${}^t\mathbf{AAx} = {}^t\mathbf{Ab}({}^t\mathbf{A}$ is the transposed matrix of $\mathbf{A}$) [57], which can be solved by Gaussian elimination. In this way the solution that satisfies the least squares of pliable constraints and that satisfies rigid constraints exactly is obtained. The above process is performed for each subpicture.

The constraint-solving process proceeds by traversing the picture hierarchy in post-order. After the constraints in a subpicture are solved, the width and height of the subpicture are computed. The size of a subpicture, that is, the size of the corresponding BOX object, is referred to when the constraints in its superordinate picture are solved. In this way subpictures are processed from the leaves to the root. After this first traversal, the local solution of constraints in each subpicture is already computed. Next, the final positions of objects in the picture are computed by translating subpictures in a top-down manner. The whole process

[8] There are some symbolic manipulation systems (of recent invention) that can solve nonlinear polynomial equations by calculating Gröbner bases [12].

[9] We could extend this process so that it would solve slightly nonlinear simultaneous equations by using the technique in [71]. However, we have not yet implemented it.

```
%           Graphical Objects
Radius is 7,
circlewithlabel( a, Radius, 'A', [visible] ),
circlewithlabel( g, Radius, 'G', [visible] ),
circlewithlabel( h, Radius, 'H', [visible] ),
circlewithlabel( i, Radius, 'I', [visible] ),
circlewithlabel( m, Radius, 'M', [visible] ),
circlewithlabel( n, Radius, 'N', [visible] ),
circlewithlabel( b, Radius, 'B', [bound] ),
circlewithlabel( c, Radius, 'C', [bound] ),
circlewithlabel( d, Radius, 'D', [bound] ),
circlewithlabel( e, Radius, 'E', [bound] ),
circlewithlabel( f, Radius, 'F', [bound] ),
circlewithlabel( j, Radius, 'J', [bound] ),
circlewithlabel( k, Radius, 'K', [bound] ),
circlewithlabel( l, Radius, 'L', [bound] ),
%           Graphical Relations
adjacent( a, b, 1 ),     connect( a, b, center, center, [thick, dashed] ),
adjacent( b, m, 1 ),     connect( b, m, center, center, [thick, dashed] ),
adjacent( e, h, 1 ),     connect( e, h, center, center, [thick, dashed] ),
adjacent( e, i, 1 ),     connect( e, i, center, center, [thick, dashed] ),
adjacent( f, g, 1 ),     connect( f, g, center, center, [thick, dashed] ),
adjacent( f, n, 1 ),     connect( f, n, center, center, [thick, dashed] ),
adjacent( b, c, 1 ),     connect( b, c, center, center, [thick, solid] ),
adjacent( c, d, 1 ),     connect( c, d, center, center, [thick, solid] ),
adjacent( c, e, 1 ),     connect( c, e, center, center, [thick, solid] ),
adjacent( c, f, 1 ),     connect( c, f, center, center, [thick, solid] ),
adjacent( c, l, 1 ),     connect( c, l, center, center, [thick, solid] ),
adjacent( d, f, 1 ),     connect( d, f, center, center, [thick, solid] ),
adjacent( e, f, 1 ),     connect( e, f, center, center, [thick, solid] ),
adjacent( j, k, 1 ),     connect( j, k, center, center, [thick, solid] ),
adjacent( j, l, 1 ),     connect( j, l, center, center, [thick, solid] ),
adjacent( k, l, 1 ),     connect( k, l, center, center, [thick, solid] ).
```

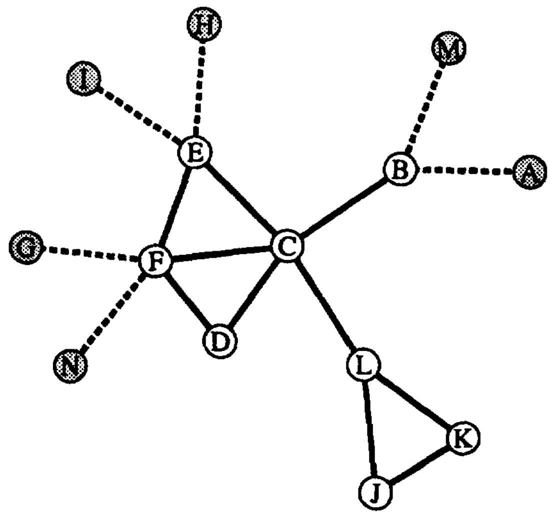


Fig. 11. A network diagram.

of solving constraints in the picture hierarchy is summarized in a programming langauge form, as follows.

(1) Bottom-up phase: solve(rootpicture)

```
solve(sp)
sp is a subpicture;
{
    let $sp_1, sp_2, \ldots, sp_m$ be the subordinate subpictures of sp;
    let $o_1, o_2, \ldots, o_n$ be the objects in sp;
    solve($sp_1$);
    solve($sp_2$);
    ...
    solve($sp_m$);
    convert all constraints into the form of ordered linear combinations;
    eliminate rigid constraints;
    add the following constraints if necessary:
        $(o_1 \cdot cx + o_2 \cdot cx + \cdots + o_n \cdot cx)/n = center\_x$;---(a)
        $(o_1 \cdot cy + o_2 \cdot cy + \cdots + o_n \cdot cy)/n = center\_y$;---(b)
    solve pliable constraints;
    compute the width of sp:
        $\max_i o_i \cdot rx - \min_i o_i \cdot lx$;
    compute the height of sp:
        $\max_i o_i \cdot ty - \min_i o_i \cdot by$;
}
```

(2) Top-down phase: translate (rootpicture).

```
translate(sp)
sp is a subpicture;
{
    let $sp_1, sp_2, \ldots, sp_m$ be the subordinate subpictures of sp;
    let $o_1, o_2, \ldots, o_n$ be the objects in sp;
    translate $o_1, o_2, \ldots, o_n$;
    translate($sp_1$);
    translate($sp_2$);
    ...
    translate($sp_m$);
}
```

Consider the procedure **solve**(*sp*) in more detail. The system reports an error if the rigid constraints are contradictory. It is not a problem if constraints are specified redundantly. The positions of graphical objects usually have the freedom of translation, because a set of geometric relations describes only relative positions of graphical objects. In order to solve this under-constrained problem, COOL adds the above constraints (a) and (b) to the equation system if necessary. These constraints place the average position of all objects at the center of a subpicture. However, COOL cannot handle other under-constrained problems. It should be noted that the system executes *max* and *min* operations to compute the size of a subpicture while solving constraints subpicture-by-subpicture. If we try to solve all constraints in the picture hierarchy simultaneously, we must solve the *max* and *min* constraints with other constraints at the same time. Our solution technique avoids this difficult problem, and achieves the same effect by bottom-up constraint solving. However, it is limited in that the internal layout of a subpicture must be computable on its own. For example, one cannot have a subpicture whose layout depends on the amount of space allocated for it by the containing picture. To realize such influences in the current version, one would simply collapse the subpictures into one picture.

5. APPLICATIONS

In this section we present four applications of TRIP. First, we consider the visualization of simple relations described in English, and generate kinship diagrams from English sentences. Since a natural language is, of course, the best-understood textual representation for us, the translation from natural language representations into pictorial ones is of great importance. Next, we consider the visualization of list structures as a typical problem of displaying data structures. We then try to generate Nassi–Shneiderman diagrams from source programs of a programming language. Finally, we discuss the problem of generating entity-relationship diagrams as a visualization problem in the area of databases.

5.1 Kinship Diagrams

It is often difficult to understand complicated kinship relations expressed in English sentences. We must read them several times over and then imagine the relations in a chart-like form. A pictorial representation, however, is so powerful that we can understand the overall structure by glancing at it. Here we describe an application in which kinship relations described in English are translated into conventional kinship diagrams. This application is a good example of the use of pliable constraints. We explain the translation process by giving an example. The input sentences are given as follows:

B is the wife of A
C, D, and E are the children of A.
D is the husband of H.
I and J are the children of H.
E is the father of G.
F is the mother of G.

Several expressions are allowed to represent the relations among husband, wife, and child (children), though the vocabulary is limited.[10] The above sentences represent three families, each of which consists of parents and a child or children. These sentences are first translated into their semantic structures based on the case grammar framework [74]. We use a special parser[11] in TRIP which translates a sentence into the underlying case structure. The family relations are inferred from these semantics. Then, we can regard the relational structure as the

[10] Strictly speaking, the sentences allowed here are pseudo-English ones about kinship. In general, kinship sentences can be as complicated as any other part of English, which is still an open problem for machine understanding.
[11] This parser was originally implemented in Franz Lisp by Nobuo Satake. It is based on active chart parsing [74].

following family relations:

```
% family (Father, Mother, Children).
family ('A', 'B', ['C', 'D', ['E']).
family ('D', 'H', ['I', 'J']).
family ('E', 'F', ['G']).
person ('A').
person ('B').
person ('C').
person ('D').
person ('E').
person ('F').
person ('G').
person ('H').
person ('I').
person ('J').
```

We visualize the kinship relations by using the visual mapping in Figure 12. An object expressed by *person* is mapped to a box bounding its name (see *objectmap* in Figure 12). As for relation mapping (see *relationmap* in Figure 12), first, the two boxes corresponding to father and mother are arranged horizontally with a thick dotted connecting line. Next, these boxes are placed above the box(es) corresponding to a child or children, being connected to each child by a thin solid line. Finally, x-coordinate constraints between children (see *hor_relation* in Figure 12) are divided into four cases, depending on whether a child is married or not. Such a distinction is necessary because we represent both the relation between parents and the relation between children as x-coordinate constraints. Note that y-coordinate constraints between parents and children and x-coordinate constraints between children are specified as pliable.

Figure 13a shows the resultant picture. In this case, pliable constraints are satisfied exactly. We add one more family to the above data:

K is the child of J.
G is the mother of K.

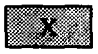
Now x-coordinate constraints become over-constrained. Figure 13b shows the picture generated by computing the least squares solution. We add a final family:

L and M are the children of C.
C is the husband of I.

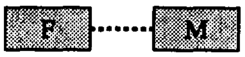
The new relations[12] prevent the exact satisfaction of y-coordinate constraints, as was already the case for x-coordinates. Figure 13c shows the updated picture. To generate this picture, 99 rigid constraints and 13 pliable constraints among 108 variables are solved. As these pictures show, satisfactory diagrams are obtained even when the kinship relations form a network structure. However, there may be more complicated cases that the visual mapping in this example could not handle. In such cases a more elaborate visual mapping, including exchanges of the positions of children, might be necessary.

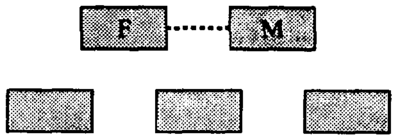
[12] Since the pair C and I have the third degree of kinship, they are legally prohibited at this time from getting married in Japan.

```
objectmap :-
      person( X ),
      boxwithlabel( X, 40, 20, X, [visible] ),
      fail.
```

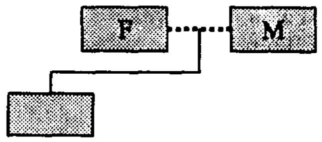


```
relationmap :-
      family( F, M, C ),
      % relation between father and mother
      point( [F, M] ),
      horizontallisting( [F, [F, M], M], 20, [rigid] ),
      connect( F, M, right, left, [thick, dotted] ),
      % relation between parents and children
      x_average( [F, M], C, [rigid] ),
      ver_map( [F, M], C ),
      % relation among children
      hor_map( C ),
      fail.
```

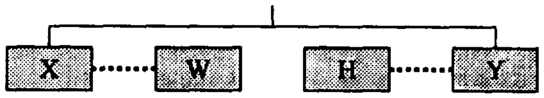




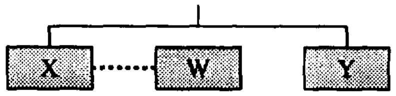
```
ver_map( _, [] ).
ver_map( P, [C | L] ) :-
      y_order( [P, C], 50, [pliable] ),
      connect( P, C, bottom, top, [orthogonal, thin, solid] ),
      ver_map( P, L ).
```



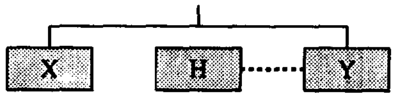
```
hor_map( [_] ).
hor_map( [X, Y | L] ) :-
      hor_relation( X, Y ), hor_map( [Y | L] ).

hor_relation( X, Y ) :-
      family( X, W, _ ),
      family( H, Y, _ ),
      !,
      x_order( [W, H], 40, [pliable] ).
```



```
hor_relation( X, Y ) :-
      family( X, W, _ ),
      !,
      x_order( [W, Y], 40, [pliable] ).
```



```
hor_relation( X, Y ) :-
      family( H, Y, _ ),
      !,
      x_order( [X, H], 40, [pliable] ).
```



```
hor_relation( X, Y ) :-
      x_order( [X, Y], 40, [pliable] ).
```

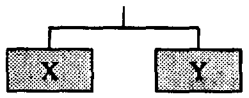
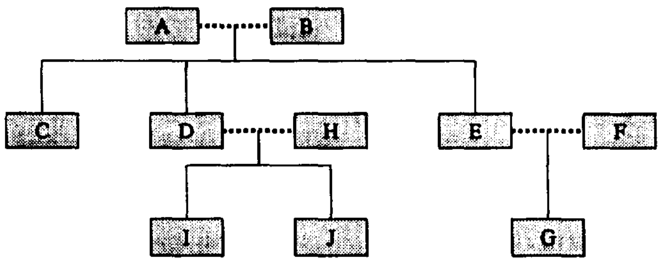


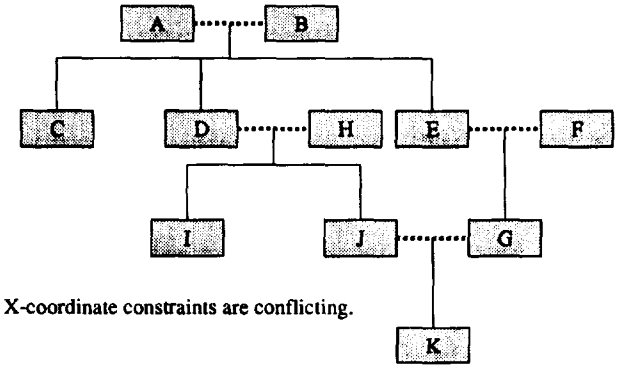
Fig. 12. Visual mapping for kinship diagrams.

5.2 Diagrams for List Structures

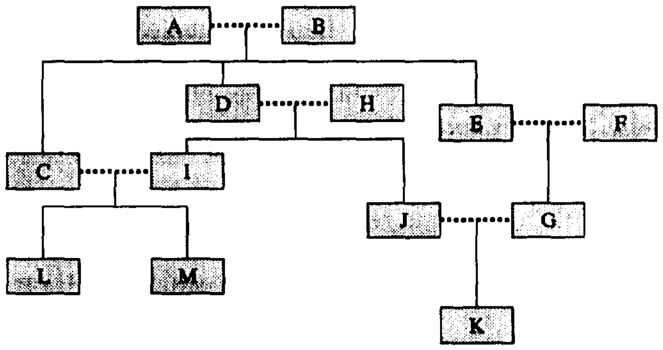
We present a simple application which uses the picture hierarchy mechanism of COOL. We try to generate diagrams for illustrating list structures that may be found in textbooks for the LISP language. A list is visualized as cells and pointers.



(a) All constraints are satisfied exactly.



(b) X-coordinate constraints are conflicting.



(c) Both x-coordinate constraints and y-coordinate constraints are conflicting.

Fig. 13. Kinship diagrams.

Such a diagram was used to animate LISP programs in [21]. The input is an S-expression, as follows:

$$((A(B))(((C)D)(E)F)(GH)).$$

Since it is the *cons* structure of a given list that we want to visualize, we translate the original data into the relational structure representing the *cons* structure explicitly. Using a simple parser, the above data is translated into

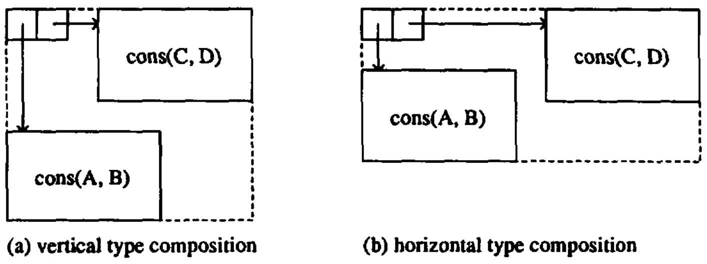

cons(C, D)
cons(A, B)
(a) vertical type composition
cons(C, D)
cons(A, B)
(b) horizontal type composition

Fig. 14. Composite rules for cells. The bounding box represents a subpicture corresponding to cons (*cons* (*A*, *B*), *cons* (*C*, *D*)).

the following predicate:

cons(*cons*('*A*', *cons*(*cons*('*B*', *nil*), *nil*)), *cons*(*cons*(*cons*(*cons*('*C*', *nil*), *cons*('*D*', *nil*)), *cons*(*cons*('*E*', *nil*), *cons*('*F*', *nil*))), *cons*(*cons*('*G*', *cons*('*H*', *nil*)),*nil*))).

Now we concentrate on visualizing this hierarchical relational structure. We map each *cons* (X, Y) pattern to a subpicture which includes a cell. If X or Y is also a *cons* pattern, it is mapped to a subordinate subpicture. In this case the subpicture corresponding to the car-part is placed below the cell, and the one corresponding to the cdr-part is placed to the right of it. The pointers are represented by vertical and horizontal arrows. As illustrated in Figure 14, we can select one of two rules for laying out these two subpictures. Figure 15 shows the visual mapping, including these composite rules. A predicate *diagonal* draws a diagonal line in the specified box, which represents a null pointer. To connect a cell to the subpictures by vertical and horizontal arrows, reference points *topleft* and *lefttop* are declared as *top* and *left* of the left box of the cell in a subpicture (see *cell* in Figure 15).

Figures 16a and 16b show the resultant pictures of vertical type composition and of horizontal type composition, respectively. To avoid imbalance, we have tried a mixed type composition in which compositions of these two types are applied in turn. Figures 16c and 16d show the pictures generated by the use of this mixed version. In order to produce more balanced layouts, the visual mapping must choose the appropriate composite rules, depending on the given structures.

5.3 Diagrams for Program Structures

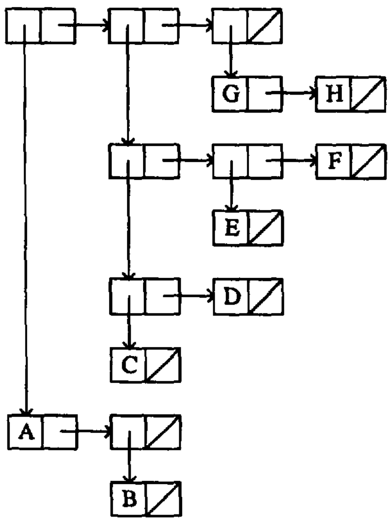
We apply TRIP to display the structures of programs. We use the structured charts known as Nassi–Shneiderman (NS) diagrams [51]. We present a general system by which a source program written in a programming language is translated into the corresponding NS diagram. In this visualization problem, a programming language is the original textual representation, and an NS diagram is the target pictorial representation in our definition. As the relational structure representation, we introduce an intermediate universal representation independent of a specific language. This representation is mapped to a diagram based on

```
v( cons( cons( A,  B ), cons( C, D ) ) ) :-
        P = cons( cons( A,  B ), cons( C, D ) ),
        pstart( P ),
        v( cons( A, B ) ),
        v( cons( C, D ) ),
        cell( P ),
        vertical( [[P, car], cons( A, B )], [left_align] ),
        horizontal( [[P, cdr], cons( C, D )], [top_align] ),
        % vertical type composition                          % horizontal type composition
        y_order( [cons( C, D ), cons( A, B )], 17, [ ] ),     y_order( [[P, car], cons( A, B )], 17, [ ] ),
        x_order( [[P, cdr], cons( C, D )], 17, [ ] ),         x_order( [cons( A, B ), cons( C, D )], 17, [ ] ),
        arrow( [P, car], cons( A, B ), center, topleft, [ ] ),
        arrow( [P, cdr], cons( C, D ), center, lefttop, [ ] ),
        pend.

v( cons( cons( A,  B ), nil ) ) :-
        P = cons( cons( A,  B ), nil ),
        pstart( P ),
        v( cons( A, B ) ),
        cell( P ),
        diagonal( [P, cdr] ),
        verticallisting( [[P, car], cons( A, B )], 17, [left_align] ),
        arrow( [P, car], cons( A, B ), center, topleft, [ ] ),
        pend.

v( cons( A, cons( B, C ) ) ) :-
        P = cons( A, cons( B, C ) ),
        pstart( P ),
        v( cons( B, C ) ),
        cell( P ),
        label( [P, car], A, [bound] ),
        horizontallisting( [[P, cdr], cons( B, C )], 17, [top_align] ),
        arrow( [P, cdr], cons( B, C ), center, lefttop, [ ] ),
        pend.

v( cons( A,  nil ) ) :-
        P = cons( A, nil ),
        pstart( P ),
        cell( P ),
        label( [P, car], A, [bound] ),
        diagonal( [P, cdr] ),
        pend.

cell( P ) :-
        box( [P, car], 17, 17, [bound] ),
        box( [P, cdr], 17, 17, [bound] ),
        horizontallisting( [[P, car], [P, cdr]], 0, [ ] ),
        reference( P, topleft, [P, car], top ),
        reference( P, lefttop, [P, car], left ).
```

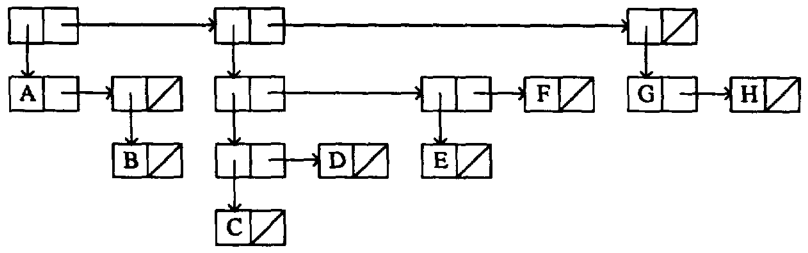
Fig. 15. Visual mapping for list diagrams.

boxes. The parsers for translating a source program into the intermediate representation are prepared for the respective language.

The visual mapping used for this application is shown in Figure 17. We visualize iterative and alternative commands as the special boxes. Figure 18 shows the NS



(a) vertical type composition

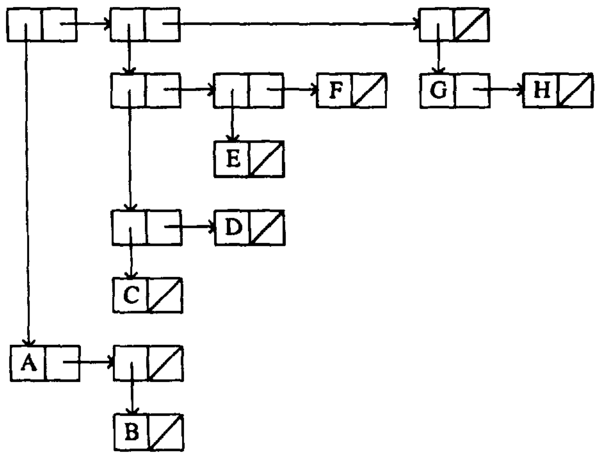


(b) horizontal type composition

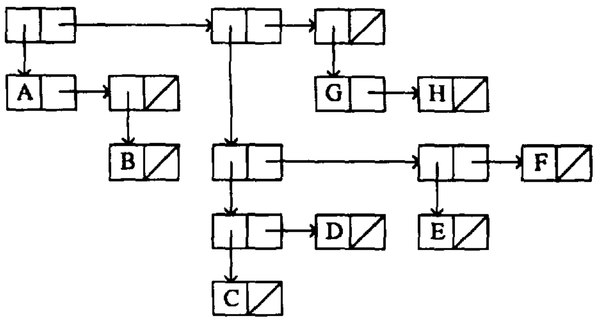
Fig. 16. List diagrams. Data: ((*A* (*B*)) (((*C*) *D*) (*E*) *F*) (*GH*)).

diagram generated for a binary search program written in C. The program is translated into the following intermediate representation:

```
[header('int binarysearch(int x, int v[], int n)'),
declaration('int low, high, mid;'),
statement('low = 0;'),
statement('high = n - 1;'),
iteration('while (low <= high)',[
  statement('mid = (low + high)/2;'),
  alternation('x < v[mid]',
    [statement('high = mid - 1;')],
    [alternation('x > v[mid]',
      [statement('low = mid + 1;')],
      [statement('return mid;')])])]),
statement('return - 1;')]
```



(c) mixed type (vertical and horizontal) composition



(d) mixed type (horizontal and vertical) composition

Fig. 16. *Continued.*

The visual mapping is applied to this list. We can extend this system to a more advanced version by incorporating the special handling of variable declarations, SWITCH-statements, and procedure calls.

5.4 Entity-Relationship Diagrams

The entity-relationship (ER) model has been widely used to represent the schema of a database. Entity-relationship diagrams are very useful to visualize the structure of data based on this model [18]. Some algorithms for automatic layout of ER diagrams have been studied (for example, [66]). Recently, database systems with a visual interface, whose users can manipulate databases by editing their graph representations directly, have been developed [13, 41]. We present a system for drawing ER diagrams from the nongraphical ER schema description as an application of TRIP. The layout of an ER diagram is computed by the graph layout system.

```
%    v( ProgramList, WidthOfDiagram ).
v( [H], WIDTH ) :-
        pstart( [H], [bound] ),
        vline( H, WIDTH ),
        pend.

v( [H | L], WIDTH ) :-
        pstart( [H | L], [bound] ),
        v( L, WIDTH ),
        vline( H, WIDTH ),
        verticallisting( [H, L], 0, [ ] ),
        pend.

vline( header( S ), WIDTH ) :-
        boxwithlabel( header( S ), WIDTH, 20, S, [left, visible] ).

vline( statement( S ), WIDTH ) :-
        boxwithlabel( statement( S ), WIDTH, 20, S, [left, invisible] ).

vline( declaration( S ), WIDTH ) :-
        boxwithlabel( declaration( S ), WIDTH, 20, S, [left, invisible] ).

vline( iteration( S, BODY ), WIDTH ) :-
        O = iteration( S, BODY ),
        pstart( O, [bound] ),
        NEWWIDTH is WIDTH - 40,
        v( BODY, NEWWIDTH ),
        boxwithlabel( [O, iteration], WIDTH, 20, S, [left, invisible] ),
        verticallisting( [[O, iteration], BODY], 0, [right_align] ),
        pend.

vline( alternation( S, TRUE, FALSE ), WIDTH ) :-
        O = alternation( S, TRUE, FALSE ),
        pstart( O, [bound] ),
        NEWWIDTH is WIDTH / 2,
        v( TRUE, NEWWIDTH ),
        v( FALSE, NEWWIDTH ),
        boxwithlabel( [O, alternation], WIDTH, 20, S, [top, bound] ),
        separateline( [O, alternation] ),
        label( [O, alternation], 'T', [left] ),
        label( [O, alternation], 'F', [right] ),
        verticallisting( [[O, alternation], TRUE], 0, [left_align] ),
        verticallisting( [[O, alternation], FALSE], 0, [right_align] ),
        pend.

% vline( alternation( S, [ ], FALSE ), WIDTH ) and vline( alternation( S, TRUE, [ ] ), WIDTH )
% are defined in the same way.
```

Fig. 17. Visual mapping for NS diagrams.

We describe an ER schema in Prolog. We can use another textual language for describing an ER schema, if we prepare a translator from the language into the Prolog representation. A predicate *entity* defines an entity set and its attributes. A predicate *relationship* defines a relationship between entity sets.

INPUT DATA:

```
int binarysearch(int x, int v[], int n)
{
int low, high, mid;
low = 0;
high = n - 1;
while (low <= high) {
mid = (low + high) / 2;
if (x < v[mid])
high = mid - 1;
else if(x > v[mid])
low = mid + 1;
else
return mid;
}
return -1;
}
```

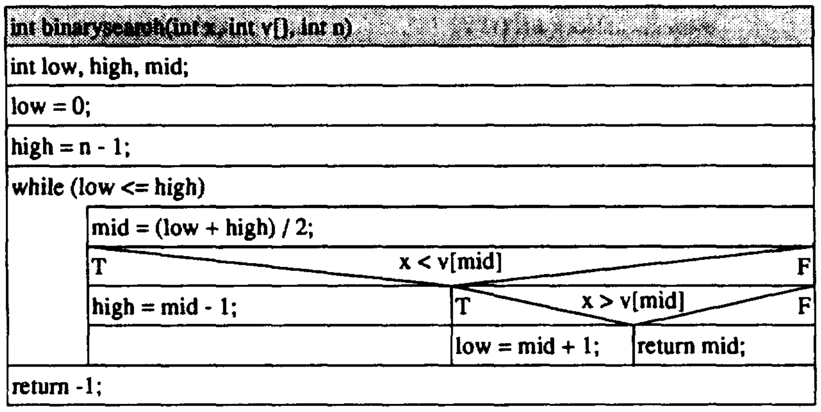


Fig. 18. An NS diagram for binary search program (C).

We assume that relationships are binary. As an example, we visualize an ER schema for an educational institute, which is described as follows:

```
% entity(ENTITY, ATTRIBUTES).
entity(institute, [name]).
entity(course, [code, name]).
entity(student, [number, name, grade]).
% relationship(RELATIONSHIP, ENTITY1, ENTITY2, #1, #2).
relationship('C-I', course, institute, n, 1).
relationship('S-I', student, institute, n, 1).
relationship('S-C', student, course, m, n).
```

The fourth and fifth arguments of *relationship* represent the cardinality of a relationship. For example, the *C-I* relationship is of cardinality $n : 1$.

According to the convention in drawing ER diagrams, we visualize entity sets, attributes, and relationships as boxes, ellipses, and diamonds, respectively. As

```
%    mapping for entity predicates
entitymap :-
        entity( ENTITY, ATTRIBUTES ),
        boxwithlabel( ENTITY, 50, 20, ENTITY, [bound] ),
        attributemap( ENTITY, ATTRIBUTES ),
        fail.

attributemap( _, [ ] ).
attributemap( ENTITY, [H | L] ) :-
        ellipsewithlabel( [ENTITY | H], 20, 10, H, [bound] ),
        adjacent( ENTITY, [ENTITY | H], 1 ),      % entry to the graph layout system
        connect( ENTITY, [ENTITY | H], center, center, [straight] ),
        attributemap( ENTITY, L ).

%    mapping for relationship predicates
relationshipmap :-
        relationship( R, ENTITY1, ENTITY2, M, N ),
        diamondwithlabel( [R], 50, 30, R, [visible] ),
        adjacent( ENTITY1, ENTITY2, 3 ),        % entry to the graph layout system
        between( [R], ENTITY1, ENTITY2, [ ] ),
        connectwithlabel( ENTITY1, [R], center, center, [straight], M ),
        connectwithlabel( ENTITY2, [R], center, center, [straight], N ),
        fail.
```

Fig. 19. Visual mapping for ER diagrams.

for line connections, we adopt straight-line connections between a box and an ellipse and between a box and a diamond. Figure 19 shows the visual mapping for ER diagrams. The positions of boxes and ellipses are computed by the graph layout system. A diamond is placed between the boxes to which the diamond is connected. The cardinality of a relationship is visualized as the labels associated with the connecting lines. Figure 20a shows the resultant picture. The layout is well balanced. Next, we add the following data to the above ER schema:

entity(*teacher*, [*name*, *salary*]).
relationship('*I-T*', *institute*, *teacher*, 1, *n*).
relationship('*C-T*', *course*, *teacher*, *m*, *n*).

The ER diagram for the updated data is shown in Figure 20b. The graph layout system incorporated into TRIP can be used separately for automatic generation of such network diagrams. We could build a visual database browser by combining this layout algorithm with a direct manipulation interface.

6. FUTURE RESEARCH DIRECTIONS

In the previous section, we applied our system to four visualization problems which look quite different. As these examples show, our visualization technique has much flexibility, which is mainly derived from the power of COOL. However, the current version of COOL can still be extended. First, we must add other sets of graphical objects and relations (e.g., pie objects and circular layout relations and flowchart symbols and format relations). In addition, more reference points should be added to a graphical object. Second, we should provide high-level

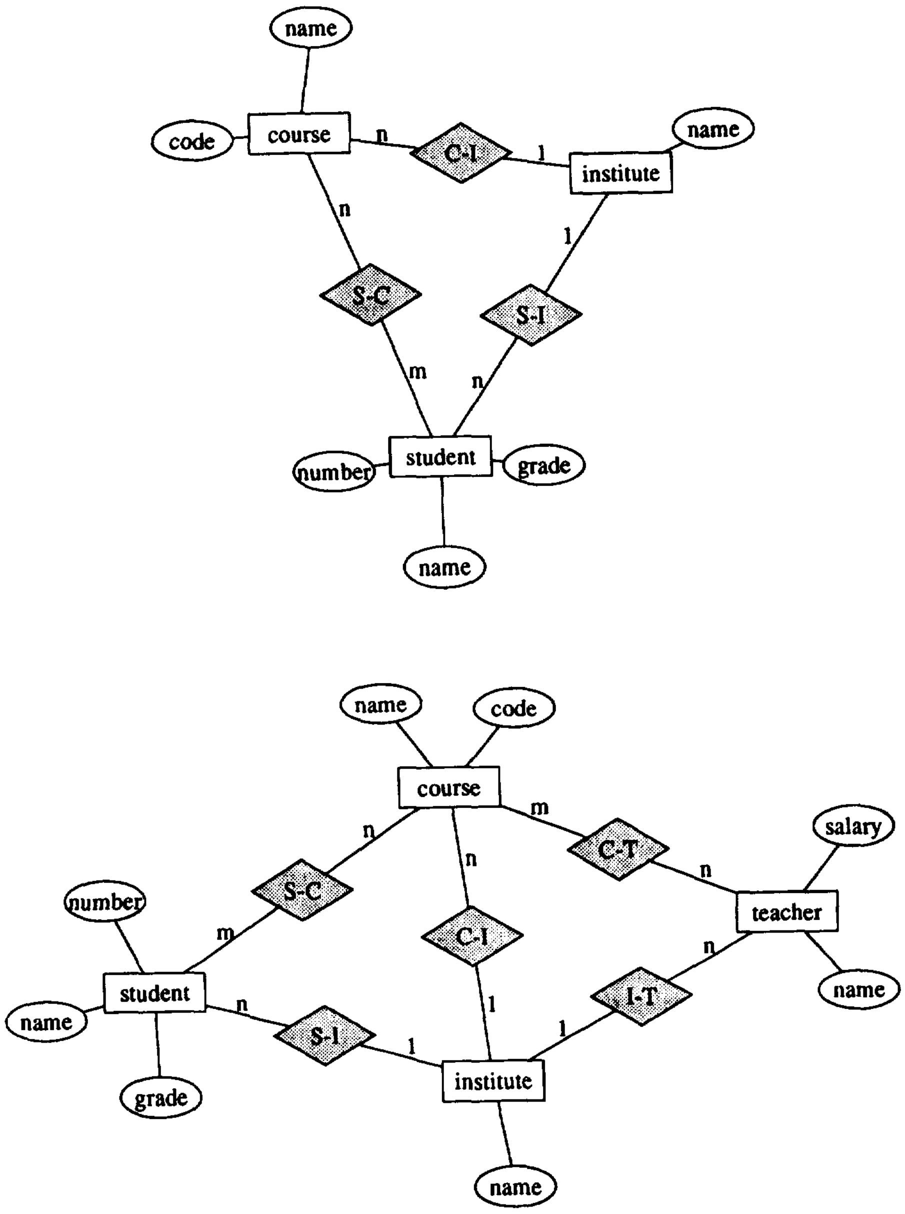


Fig. 20. Entity-relationship diagrams.

drawing relations that draw labels and lines depending on the positions of objects. For example, when a relation draws a line between two objects, it would be better in many cases for the system to automatically select their nearest reference points and choose an appropriate route. Further, it would be a challenging extension to draw labels and lines so that they could not overlap other graphical objects. Third, we desire to introduce nonlinear constraints, including inequalities. It is difficult and less efficient to solve these constraints generally. The constraint solver adopted in CLP(R) would be a good approach to this problem. It divides constraints into several classes, such as linear equalities, linear inequalities, and nonlinear equalities, and invokes distinct solution modules for each class [29].

So far we have focused on the visualization of information, that is, translation from textual or internal representations into pictorial representations. The translation from pictorial representations into textual or internal ones is also an important part of the realization of visual communication. The visualization framework presented is so general that it can be applied to this inverse translation. For example, it can be applied to the interaction model of direct manipulation systems, in which visual structures are distinctly separated from relational ones. Users manipulate only visual structures, and the consistency between visual and relational structures is maintained by the translation mappings between them. Such a general framework is important to systematically generate interactive user interface modules. Image recognition is another application of this inverse translation. In this case the inverse translation process works as follows. First, graphical objects are recognized from an input image and graphical relations among the objects are detected. An efficient spatial parsing technique is necessary in this phase. Then, graphical objects are mapped to abstract ones, and the graphical relations among them are mapped to abstract relations among the corresponding abstract objects by inverse visual mapping. Finally, a target textual or internal representation is synthesized. We could build a general tool based on this framework, whose users could obtain the desired image recognition system by specifying the inverse visual mapping.

We have not discussed the problem of how to choose an appropriate and effective pictorial representation. For the present, how to use graphical objects and relations in specifying the visual mapping in our system is the users' responsibility. However, we must solve this problem in order to build a visualization database and automate visual mapping. Researchers in cognitive science and human factors are studying this problem intensively. We will note and use their results in this area.

7. CONCLUDING REMARKS

We have presented a general visualization framework in which abstract objects and relations are visualized as graphical objects (such as boxes and circles) and graphical relations (such as horizontal/vertical listings and line connections) among graphical objects according to user-defined translation rules. We have realized a constraint-based object layout system (COOL), which plays a central role in our visualization framework. It can be also used separately as a high-level graphics system. In fact, all the illustrations in this paper were generated by COOL. The visualization of information involves many issues other than the problems described here. We are going to extend our visualization framework and introduce new ideas in order to handle a wider range of visualization problems. Our long-term goal is to construct an integrated system in which the textual and visual worlds are connected to each other by bidirectional translation.

ACKNOWLEDGMENTS

We would like to thank Nobuo Satake for his program for parsing English sentences, which we used to realize an application of our system. We are thankful to Chien Lee and Satoshi Matsuoka for their helpful discussions. We are also grateful to the referees and the editor for their comments and suggestions.

REFERENCES

1. ADOBE SYSTEMS INC. *PostScript Language Reference Manual.* Addison-Wesley, Reading, Mass., 1985.
2. ANSI. American National Standard for the functional specification of the programmer's hierarchical interactive graphics standard (PHIGS). ANSI/X3H3, American National Standards Institute, New York, 1983.
3. ARENS, Y., MILLER, L., SHAPIRO, S. C., AND SONDHEIMER, N. K. Automatic construction of user-interface displays. In *Proceedings of AAAI-88* (Minnesota, Aug. 1988). 808–813.
4. BENTLEY, J. L., AND KERNIGHAN, B. W. A system for algorithm animation: Tutorial and user manual. Tech. Rep. 132, AT&T Bell Labs., Jan. 1987.
5. BORNING, A. The programming language aspects of ThingLab, a constraint-oriented simulation laboratory. *ACM Trans. Program. Lang. Syst. 3,* 4 (Oct. 1981), 353–387.
6. BORNING, A. Graphically defining new building blocks in ThingLab. *Human-Comput. Interaction 2* (1986), 269–295.
7. BORNING, A., AND DUISBERG, R. Constraint-based tools for building user interfaces. *ACM Trans. Graph. 5,* 4 (Oct. 1986), 345–374.
8. BORNING, A., DUISBERG, R., AND FREEMAN-BENSON, B. Constraint hierarchies. In *Proceedings of OOPSLA '87* (Oct. 1987). 48–60.
9. BORNING, A., MAHER, M., MARTINDALE, A., AND WILSON, M. Constraint hierarchies and logic programming. Tech. Rep. 88-11-10, Dept. of Computer Science, Univ. of Washington, Seattle, Nov. 1988.
10. BROWN, M. H. *Algorithm Animation.* MIT Press, Cambridge, Mass., 1988.
11. BROWN, M. H., AND SEDGEWICK, R. A system for algorithm animation. *Comput. Graph. 18,* 3 (Jul. 1984), 177–186.
12. BUCHBERGER, B. Gröbner bases: An algorithmic method in polynomial ideal theory. Tech. Rep. CAMP-LINZ, 1983.
13. BURNS, L. M., ARCHIBALD, J. L., AND MALHOTRA, A. A graphical entity-relationship database browser. Res. Rep. RC 13238, IBM T. J. Watson Research Center, Yorktown Heights, N.Y., 1987.
14. CAPORAL, P. M., AND HAHN, G. J. Tools for automated statistical graphics. *IEEE Comput. Graph. Appl. 1,* 4 (Oct. 1981), 72–82.
15. CHANG, S. K. Visual languages: A tutorial and survey. *IEEE Softw. 4,* 1 (Jan. 1987), 29–39.
16. CHANG, S. K. *Principles of Pictorial Information Systems Design.* Prentice-Hall, Englewood Cliffs, N.J., 1989.
17. CHAPIN, N. New format for flowcharts. *Softw. Pract. Exper. 4,* 4 (Apr. 1974), 341–357.
18. CHEN, P. P. The entity relationship model—towards a unified view of data. *ACM Trans. Database Syst. 1,* 1 (Mar. 1976), 9–36.
19. COLMERAUER, A. Opening the Prolog III universe. *BYTE 12,* 8 (Aug. 1987), 177–182.
20. DERMAN, E., AND VAN WYK, C. J. A simple equation solver and its application to financial modelling. *Softw. Pract. Exper. 14,* 12 (Dec. 1984), 1169–1181.
21. DIONNE, M. S., AND MACKWORTH, A. K. ANTICS: A system for animating LISP programs. *Comput. Graph. Image Process. 7,* 1 (Feb. 1978), 105–119.
22. DUISBERG, R. A. Animation using temporal constraints: An overview of the Animus system. *Human-Comput. Interaction 3* (1987/1988), 275–307.
23. EADES, P., AND TAMASSIA, R. Algorithms for drawing graphs: An annotated bibliography. Tech. Rep. CS-89-09, Dept. of Computer Science, Brown Univ., Providence, R.I., Feb. 1989.
24. ENDERLE, G., KANSY, K., AND PFAFF, G. *Computer Graphics Programming GKS—The Graphics Standard.* Springer-Verlag, Berlin, 1984.

25. FERSTL, O. Flowcharting by stepwise refinement. *ACM SIGPLAN Not. 13,* 1 (Jan. 1978), 34–42.
26. FUTAMURA, Y., KAWAI, T., HORIKOSHI, H., AND TSUTSUMI, M. Development of computer programs by problem analysis diagram (PAD). In *Proceedings of 5th International Conference on Software Engineering* (1981). 325–332.
27. GOSLING, J. Algebraic constraints. Ph.D. dissertation, Carnegie-Mellon Univ., Pittsburgh, Pa., 1983.
28. HEARN, A. C. *REDUCE User's Manual, Version 3.3.* The Rand Corporation, Santa Monica, Calif., 1987.
29. JAFFAR, J., AND LASSEZ, J-L. From unification to constraints. In *Proceedings of 6th Logic Programming Conference* (Tokyo, Jun. 1987). 1–18.
30. JOHNSON, S. C. *YACC: Yet Another Compiler-Compiler.* UNIX Manual, vol. 2B, 1977.
31. KAMADA, T. On visualization of abstract objects and relations. Ph.D. dissertation, Dept. of Information Science, Univ. of Tokyo, Dec. 1988.
32. KAMADA, T., AND KAWAI, S. An enhanced treatment of hidden lines. *ACM Trans. Graph. 6,* 4 (Oct. 1987), 308–323.
33. KAMADA, T., AND KAWAI, S. A simple method for computing general position in displaying three-dimensional objects. *Comput. Vision, Graph. Image Process. 41,* 1 (Jan. 1988), 43–56.
34. KAMADA, T., AND KAWAI, S. Advanced graphics for visualization of shielding relations. *Comput. Vision, Graph. Image Process. 43,* 3 (Sept. 1988), 294–312.
35. KAMADA, T., AND KAWAI, S. An algorithm for drawing general undirected graphs. *Inf. Process. Lett. 31,* 1 (Apr. 1989), 7–15.
36. KERNIGHAN, B. W. PIC—A language for typesetting graphics. *Softw. Pract. Exper. 12,* 1 (Jan. 1982), 1–21.
37. KNUTH, D. E. Computer-drawn flowcharts. *Commun. ACM 6,* 9 (Sept. 1963), 555–563.
38. KNUTH, D. E. *TEX and METAFONT.* Digital Press, 1979.
39. KONOPASEK, M., AND JAYARAMAN, S. Constraint and declarative languages for engineering applications: The TK!solver contribution. *Proc. IEEE 73,* 12 (Dec. 1985), 1791–1806.
40. KOWALSKI, R. *Logic for Problem Solving.* Elsevier, Amsterdam, 1979.
41. LARSON, J. A. Visual languages for database users. In *Visual Languages.* S. K. Chang, Ed., Plenum Press, New York, 1986, 127–147.
42. LASSEZ, C. Constraint logic programming. *BYTE 12,* 8 (Aug. 1987), 171–176.
43. LELER, W. *Constraint Programming Languages, Their Specification and Generation.* Addison-Wesley, Reading, Mass., 1988.
44. MACKINLAY, J. Automating the design of graphical presentations of relational information. *ACM Trans. Graph. 5,* 2 (Apr. 1986), 110–141.
45. MACSYMA GROUP. *MACSYMA Reference Manual, Version 10.* Symbolics Inc., 1984.
46. MATA, J. M. Solving systems of linear equalities and inequalities efficiently. In *15th Southeastern Conference on Combinatorics, Graph Theory and Computing* (Mar. 1984). 251–260.
47. MATA, J. M. ALLENDE: A procedural language for the hierarchical specification of VLSI layouts. In *Proceedings of the ACM IEEE 22nd Design Automation Conference* (1985). 183–189.
48. MCCORMICK, B. H., DEFANTI, T. A., AND BROWN, M. D. Visualization in scientific computing. *Comput. Graph. 21,* 6 (Nov. 1987).
49. MESSINGER, E. B. Automatic layout of large directed graphs. Ph.D. dissertation, Tech. Rep. 88-07-08, Dept. of Computer Science, Univ. of Washington, Seattle, Jul. 1988.
50. MYERS, B. A. *Creating User Interfaces by Demonstration.* Academic Press, New York, 1988.
51. NASSI, I., AND SHNEIDERMAN, B. Flowchart techniques for structured programming. *ACM SIGPLAN Not. 8,* 8 (Aug. 1973), 12–26.
52. NELSON, G. Juno, a constraint-based graphics system. *Comput. Graph. 19,* 3 (July 1985), 235–243.
53. PINEDA, L. A. A compositional semantics for graphics. In *Proceedings of Eurographics '88.* Elsevier Science, Amsterdam, 1988.
54. PINEDA, L. A., AND CHATER, N. GRAFLOG: Programming with interactive graphics and Prolog. In *Proceedings of CG International '88.* Springer, New York, 1988, 469–478.
55. PINEDA, L. A., KLEIN, E., AND LEE, J. GRAFLOG: Understanding drawings through natural language. *Comput. Graph. Forum 7,* 2 (June 1988), 97–103.

56. RAEDER, G. A survey of current graphical programming techniques. *IEEE Computer 18,* 8 (Aug. 1985), 11–25.
57. RAO, C. R. *Linear Statistical Inference and Its Applications.* 2nd ed. Wiley, New York, 1973.
58. ROACH, J. A. The rectangle placement language. In *Proceedings of the ACM/IEEE 21st Design Automation Conference* (June 1984). 405–411.
59. SAKAI, K., AND AIBA, A. CAL: A theoretical background of constraint logic programming and its applications. Tech. Rep. 364, ICOT Research Center, Japan, Apr. 1988.
60. SHIMOMURA, T. A method for automatically generating business graphs. *IEEE Comput. Graph. Appl. 3,* 6 (Sept. 1983), 55–59.
61. SHU, N. C. *Visual Programming.* Van Nostrand Reinhold, New York, 1988.
62. SLOCUM, J. A survey of machine translation: Its history, current status, and future prospects. *Comput. Linguistics 11,* 1 (Jan./Mar. 1985), 1–17.
63. SUSSMAN, G. J., AND STEELE, G. L. CONSTRAINTS—A language for expressing almost-hierarchical descriptions. *Artif. Intell. 14,* 1 (Jan. 1980), 1–39.
64. SUTHERLAND, I. E. Sketchpad: A man-machine graphical communication system. In *Proceedings of Spring Joint Computer Conference* (1963). 329–346.
65. SZEKELY, P. A., AND MYERS, B. A. A user interface toolkit based on graphical objects and constraints. In *Proceedings of OOPSLA '88* (San Diego, Calif., Sept. 1988). 36–45.
66. TAMASSIA, R., BATINI, C., AND TALAMO, M. An algorithm for automatic layout of entity relationship diagrams. In *Entity-Relationship Approach to Software Engineering.* R. Yeh, Ed. Elsevier Science, Amsterdam, pp. 421–439.
67. TAMASSIA, R., BATTISTA, G., AND BATINI, C. Automatic graph drawing and readability of diagrams. *IEEE Trans. Syst. Man, Cybern, SMC-18,* 1 (Jan./Feb. 1988), 61–79.
68. THALMANN, D. An interactive data visualization system. *Softw.-Pract. Exper. 14,* 3 (Mar. 1984), 277–290.
69. UPSON, C., FAULHABER, T., KAMINS, D., LAIDLAW, D., SCHLEGEL, D., VROOM, J., GURWITZ, R., AND VAN DAM, A. The application visualization system: A computational environment for scientific visualization. *IEEE Comput. Graph. Appl. 9,* 4 (July 1989), 30–42.
70. VAN HENTENRYCK, P. *Constraint Satisfaction in Logic Programming.* MIT Press, Cambridge, Mass., 1989.
71. VAN WYK, C. J. A high-level language for specifying pictures. *ACM Trans. Graph. 1,* 2 (Apr. 1982), 163–182.
72. VANDER ZANDEN, B. T. Incremental constraint satisfaction and its application to graphical interfaces. Ph.D. dissertation, Dept. of Computer Science, Cornell Univ., Ithaca, N.Y., Jan. 1989.
73. WATANABE, H. Heuristic graph displayer for G-BASE. Tech. Rep. 5, Ricoh Software Research Center, Tokyo, Aug. 1987.
74. WINOGRAD, T. *Language as a Cognitive Process.* Addison-Wesley, Reading, Mass., 1983.
75. ZDYBEL, F., GREENFELD, N. R., YONKE, M. D., AND GIBBONS, J. An information presentation system. In *Proceedings of IJCAI '81* (Aug. 1981). 978–984.

Received March 1989; revised September 1989; accepted November 1989

Editor: M. Stone