Expressing Constraints for Data Display Specification: A Visual Approach

Isabel F. Cruz

Department of Computer Science
Brown University
Providence, Rhode Island 02912

CS-93-57
December 1993

Expressing Constraints for Data Display Specification: A Visual Approach*

Isabel F. Cruz†
Department of Computer Science
Brown University
Providence, RI 02912-1910
ifc@cs.brown.edu

Abstract

In this paper we introduce a constraint-based language that has a visual syntax, and allows for the declarative specification of the display of data. Other features of the proposed language include: (1) simplicity and genericity of the basic constructs; (2) ability to specify a variety of displays (graphs, bar charts, pie charts, etc.); (3) compatibility with the object-oriented framework of the database language DOODLE. We provide the syntax and the semantics of the language, and examples of applications that demonstrate the expressiveness of our language.

1 Introduction

Mappings between the data domain and the visual domain have been used for extracting information from the data, by reasoning in the visual domain [Gre83, CNI93]. For example, Venn diagrams are visual representations of abstract sets and of their inclusion relationships. Other diagrams are close to the concrete entities that they represent, such as transportation and communication networks. Also, bar charts, pie charts, and plot charts can facilitate the comprehension of large amounts of quantitative data [Tuf83].

Pictures (as displayed on a computer screen) are sets of graphical objects. Syntactically, a picture is a well-formed sentence in a visual language. In this paper we present a new constraint-based language, the *U-term language* to specify pictures *by example*: U-terms are pictures of the U-term language that exemplify how visual representations of data look like. We have designed the U-term language to provide:

- A declarative and visual specification of pictures with simple and generic constructs.
- The ability to specify a variety of pictures such as graphs, bar charts, and pie charts, using Cartesian or polar coordinates.
- Easy integration in an object-oriented framework.

Previous work on data display specification includes constraint-based languages that have a textual syntax, such as TRIP [Kam89], which uses Prolog terms and functors to specify pictures, and IDEAL [Wyk82], where textual programs instantiate abstract data types. Visual approaches include GVL [GC90] for the display of data structures (e.g., lists) that are created during the execution of a program, and ThingLab [Bor81]. ThingLab has an object-oriented approach, and the constraints are integrated with the visual classes that specify the pictures. Visual classes are described visually by means of prototypical

*A preliminary version of this paper was presented at the *First Workshop on Principles and Practice of Constraint Programming*, Newport, Rhode Island, April 1993.

†Partial support was given by ARPA order 8225, ONR grant N00014-91-J-4052.

symbols. Related research has focused on grammars for visual languages, with either a textual syntax (e.g., [GR90]) or a visual syntax (e.g.,[Lak87]).

This paper is organized as follows. In Section 2, we present an overview of the database language DOODLE that is based on the U-term language. In Section 3 we introduce the graphical principles and the objects and constraints that are the basis for the U-term language. In Section 4 we present the abstract and concrete syntax of the U-term language, followed by the semantics of the language in Section 5. Examples of applications are provided in Section 6. Finally, in Section 7 we discuss topics for future research.

2 Background

The U-term language is used to specify the visual representation of a database and to query the database using that visual representation. The database language is called DOODLE (*Draw an Object-Oriented Database LanguagE*). DOODLE as a visual query language is presented in [Cru92]. In this section we give an overview of DOODLE.

A DOODLE visual program is a set of visual rules, which can be read in any order. Visual rules are vertically divided by a double bar. We call the entities to the left and to the right of this bar *D-terms* (for DOODLE terms). The DOODLE program of Figure 1(i) relates to a software engineering application: the graph visualization of the components of a program. In Figure 1(i) there are two kinds of D-terms:

F-terms. These are terms from *F-logic* [KLW90]. F-terms are depicted as strings of characters.

U-terms. These are *user-defined* terms. A U-term is a picture, which is a sentence of the U-term language.

In this program all the terms to the right of the double bar are F-terms, hence the box `f-language`. On the left-hand side, the U-terms are used to define the visual language `softGraph` (for software graph), as indicated by `softGraph`. The program specifies that for the database facts that make the F-terms true, graphical objects similar to the U-terms on the left-hand side will be drawn on the screen. The U-terms are formed of two kinds of symbols: (1) *prototypical symbols*, which specify "by example" the visual attributes (e.g., shape) of the graphical objects to be displayed on the screen, and the spatial relationships between these objects. (2) *key symbols* (see Figure 1(ii)) determine how the prototypical symbols are interpreted.

The first rule states that any object M in class *module* is to be displayed by a box. The second rule states that any object P in class *procedure* is to be displayed by a diamond. In a similar way, objects C of class *calls* are to be displayed by (simple) arrows, while objects T of class *contains* are to be displayed by double arrows.

The U-term in the third rule specifies the following display: *"Draw a solid arrow that starts on the graphical object that displays database object* X *and ends on the graphical object that displays database object* Y*."* Therefore, refboxes allow for U-terms to refer to U-terms that were defined elsewhere. For example, in the objects of class *calls*, the refboxes make reference to the U-term that specifies the display of objects of class *procedure*. This reference is made possible by the use of a defbox in the rule that defines procedure. The defbox indicates the U-term (or part of the U-term) that can be referred to by another rule. In this example the defbox that is defined for objects of class *procedure* includes the prototypical symbol diamond and a labeled refbox. The roles of defbox and refbox are analogous to the concepts of procedure definition and of procedure call. The defbox specifies the display, and the refbox "calls" it.

In the third and fourth visual rules, the refboxes that contain X (or Y) can either refer to the defbox in the rule that specifies the display of procedures or to the defbox that specifies the display of modules (note that there is no specification for the display of objects of class *block*, and that *procedure* and *module* are subclasses of *block*).

Labeled refboxes are a generalization of refboxes. While (simple) refboxes make a call to a defbox in the current visual language (`softGraph` in the example), the labeled refboxes make a call to the visual language that labels the refbox (e.g., `str`)) which has been defined elsewhere. The last rule of the visual program

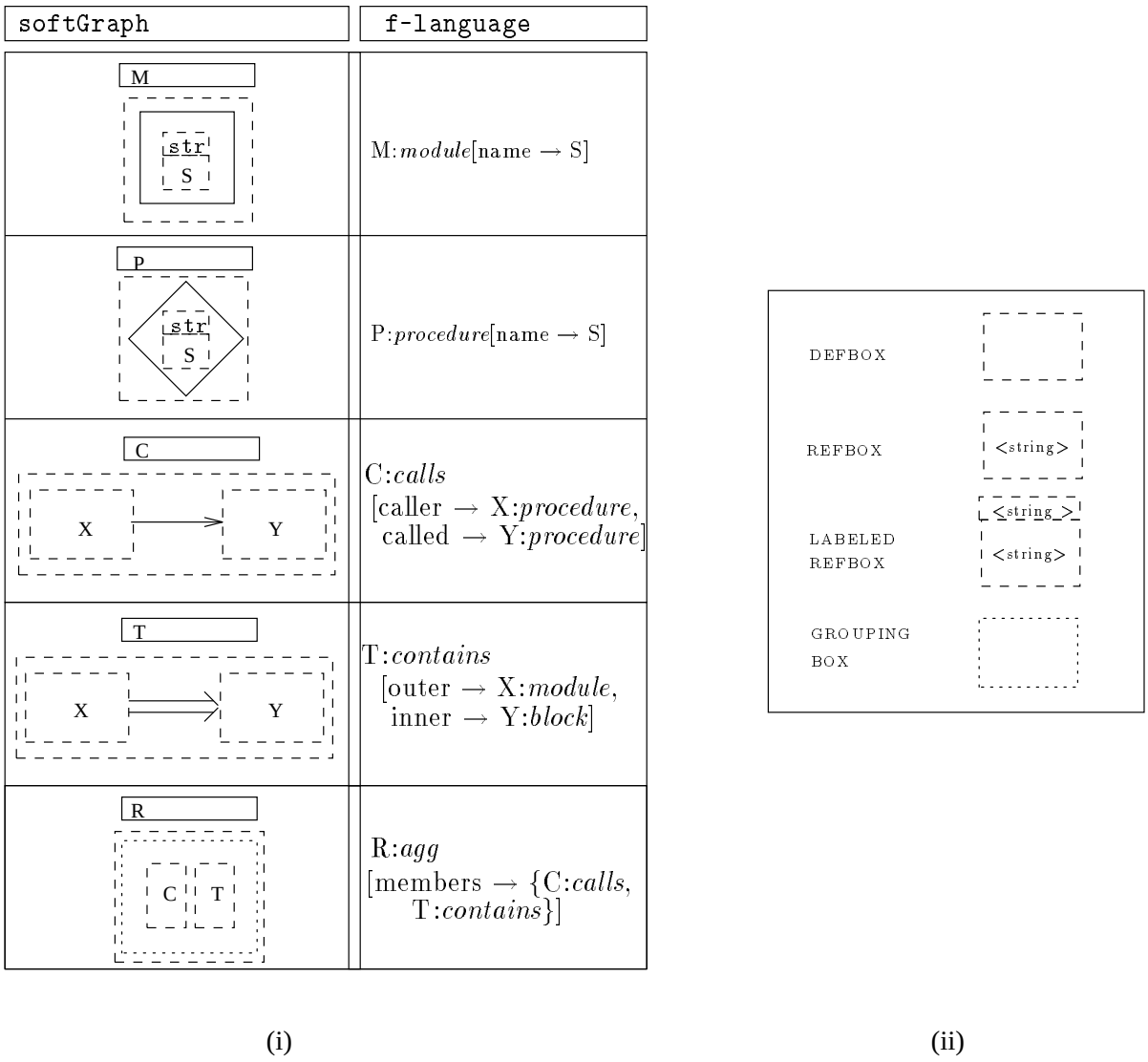

Figure 1: (i) Visual program that defines `softGraph`. (ii) Some key symbols in DOODLE.

states that the visual rendition of a set of objects R is the set of the visual representations of the members of the set (in this case C and T). We use a grouping box to denote the set of visual representations.

The semantics of DOODLE programs is given by the semantics of the equivalent F-logic program. This translation is achieved by mapping each visual rule to one F-logic rule and a U-term to a set of F-logic objects. The program of Figure 1(i) defines a mapping from database objects to graphical objects on the screen. We call such mapping a *visualization*. A set of F-logic objects that describes a picture (or set of pictures) defines a picture in the visual language `softGraph` if and only if it is a minimal model of the equivalent F-logic program. For example, if we assume that the rules of the DOODLE program of Figure 1(i) are the only ones that define the visual language `softGraph`, then (for any database) a graph where some of the nodes are circles is not a picture in `softGraph`. A DOODLE program describes only one picture, if the picture is completely specified (that is the exact placement of the graphical objects, dimensions of the objects, distances between objects, etc., are given). In general, though, a DOODLE program describes a set of pictures. Each two pictures in the set differ in one or more of the picture attributes that were not completely specified by the DOODLE program.

3 Objects and Constraints

We describe graphical objects and the spatial relationships between them using an object-oriented model. This model is the (textual) basis to the U-term language.

3.1 Visual Objects

Visual objects specify graphical objects. They correspond directly to the kinds of graphical objects that the layout program can display and are instances of *visual classes* such as *box*, *circle*, *arrow*, and *text*.

3.2 Landmarks and Anchor Points

Landmarks are used to give dimensions to the objects, and to place objects relatively to other objects. Landmarks can belong to the following classes: *landmark*, *cartesianLandmark*, *polarLandmark*, and *lineLandmark*. *Anchor point* objects are user-defined landmarks. From now on we use *point* to denote a landmark or an anchor point.

Both landmarks and anchorpoints can be described in two formats: *cartesian* or *polar*. For the cartesian format, there are two values that can be specified: the horizontal coordinate (x) and the vertical coordinate (y). For the polar format, there are two values that can be specified: the length (r) and the angular coordinate (θ). The absolute origin for both formats is the same point on the screen. Any point, and consequently any landmark or any anchor point can be positioned on the plane (or referred to) according to any of the formats, if the following constraints are enforced:

$r = \sqrt{x^2 + y^2}$
$\theta = arctan(y/x)$
$x = r\, sin\theta$
$y = r\, cos\theta$

The fact that any point can be dealt with in any of the formats makes it possible to specify the positions of polar objects in the cartesian plane (e.g., a set of pie charts, where the X-coordinate represents the year to which the pie chart refers), or to specify the position of cartesian objects in a polar picture (e.g., text that labels a sector in a pie chart).

The following discusses the different classes of landmarks and the visual objects where they can be used.

Landmarks. All visual objects have the following landmarks : `c` (for center) and `any`. `any` refers to any landmark belonging to the boundary of the object. For objects of class *text*, `c` refers to the the center of the bounding box, which is the minimal invisible box that contains the object, and `any` is any landmark that belongs to the bounding box.

Cartesian landmarks. Polygons and circles have the following cartesian landmarks: `mw` (for midwest), `mn` (for midnorth), `me` (for mideast), `ms` (for midsouth). We show their position, as located on a box and on a circle in Figure 2(i) and (ii).

Polar landmarks. Sectors (and therefore circles) can have the following polar landmarks: `first` (for first angle), `second` (for second angle), `inradius` (for inner radius), and `outradius` (for outer radius), as shown in Figure 2(iii).

Line landmarks. Lines (e.g., arrows, double-headed arrows) can have the following line landmarks `h` (for head), `t` (for tail), denoting the endpoints of a line.

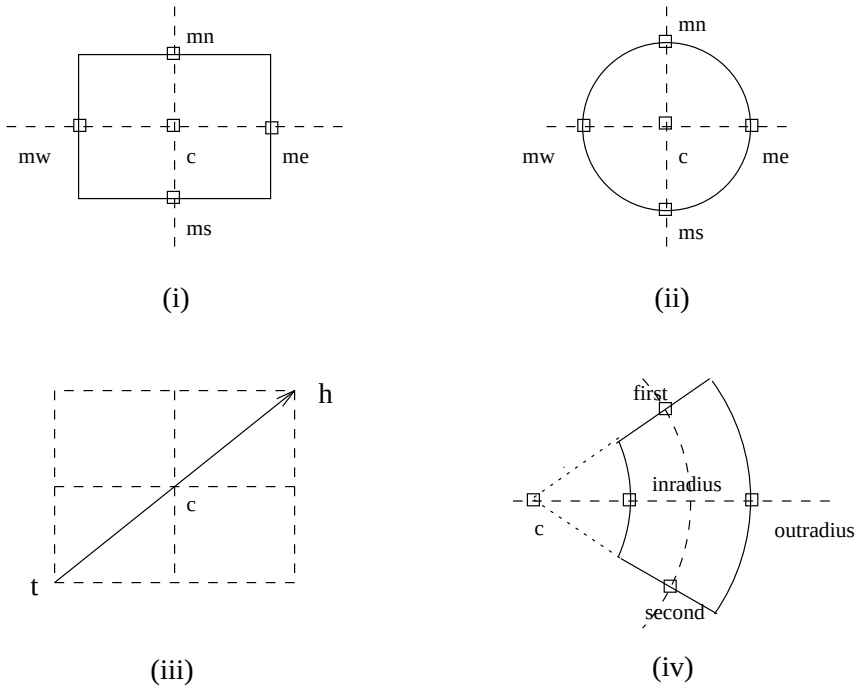

Figure 2: Cartesian landmarks: (i) on a box; (ii) on a circle; (iii) on a line. Polar landmarks: (iv) on a sector.

3.3 Visual Constraint Objects

Visual constraint objects express spatial relationships between one or more visual objects, and are instances of *visual constraint classes*. The constraints are expressed in terms of the objects' landmarks or anchor points. A visual constraint between points of the same object can be used to specify one of the dimensions of the object. A visual constraint between two points, one in each visual object, expresses a spatial relationship between two objects. We define two classes: *lengthConstraint* and *overlapConstraint* and describe their types with signature F-logic terms [KLW90].

Length constraint. The F-logic signature of the class *lengthConstraint* is as follows:

$$
\begin{aligned}
&\mathit{lengthConstraint}[\text{firstObj} \Rightarrow \mathit{visualObject};\\
&\qquad\qquad \text{secondObj} \Rightarrow \mathit{visualObject};\\
&\qquad\qquad \text{firstLand} \Rightarrow \{\mathit{landmark},\ \mathit{anchorpoint}\};\\
&\qquad\qquad \text{secondLand} \Rightarrow \{\mathit{landmark},\ \mathit{anchorpoint}\};\\
&\qquad\qquad \text{distance} \Rightarrow \mathit{realNumberExp};\\
&\qquad\qquad \text{kind} \Rightarrow \mathit{kindType}]
\end{aligned}
$$

The class *lengthConstraint* has attributes firstObject, secondObject, firstLand, secondLand, distance, and kind. Braces indicate class union. Objects of class *realNumberExp* are either constants, variables, or different kinds of expressions (e.g., $max(X, Y)$, $Height_1 + Height_2$, ≥ 0) of type *real*. Values of attribute *kind* include vertical, horizontal, absolute (Euclidean distance), radial and angular.

The class *lengthConstraint* is: (1) general, since it considers a variety of distances and kinds of distances, and (2) generic, because constraints apply to any objects as long as the landmarks are defined for those objects.

Overlap constraint. Given two visual objects, an overlap constraint specifies which object is to be drawn on top of the other object. The class *overlapConstraint* has the following signature:

overlapConstraint[firstObj ⇒ *visualObject*;
secondObj ⇒ *visualObject*;
top ⇒ *visualObject*]

Top indicates which of the two objects is to be displayed on top. The default value for the attribute top takes the first object to be the one on top.

3.4 Examples

Position between two Boxes

Figure 3 shows eight distances that can be defined between the landmarks of two boxes. These distances are a subset of all the possible distances that could be defined by these landmarks.. Not all the above

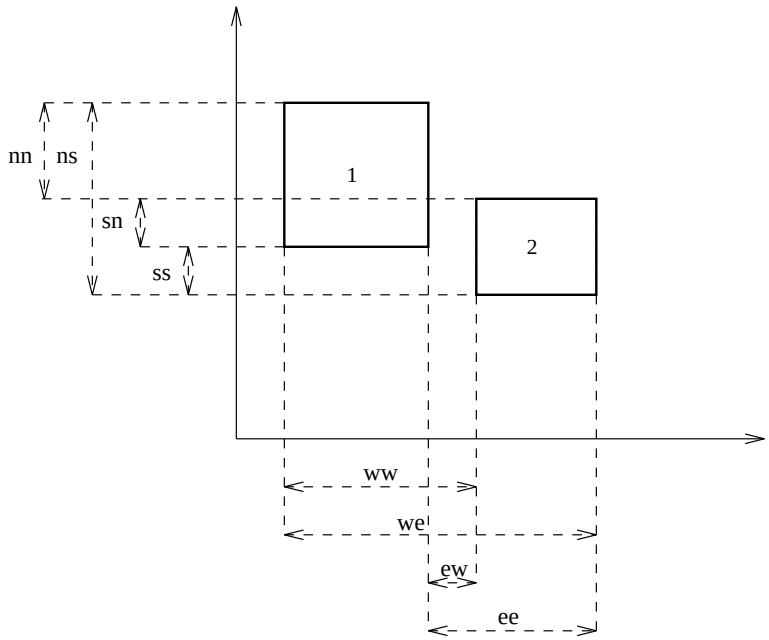

Figure 3: Constraining the position of two objects.

distances are needed to specify completely the spatial constraints between two objects The examples of Figure 4 illustrate this point. In these examples we consider the first graphical object to be the one with smallest identity, and we do not specify the absolute dimensions of each object. For example, three objects of class *lengthConstraint* are used to specify (i) and (iv) and four objects of class *lengthConstraint* are used to specify (ii) and (iii).

Position between two Sectors

Figure 5 shows four angular distances and four radial distances that constrain the position of two sectors.

Position of Anchor Points

The following example shows how the horizontal position of the anchor point labeled "headpoint" is specified using an object lc_1 of class *lengthConstraint*. In the example the vertical position of the anchor point depends on a landmark in the same object (the object with identity 1).

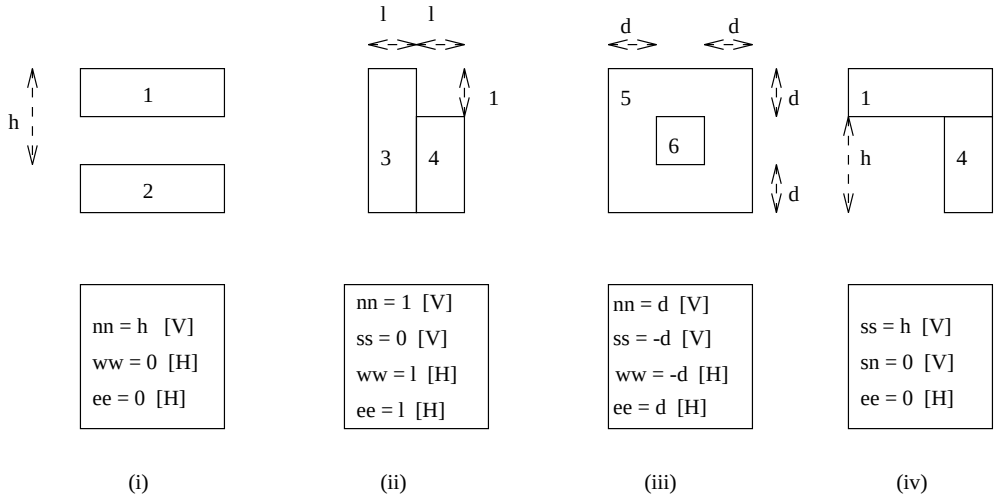

Figure 4: Spatial relationships between pairs of objects ("V" denotes vertical and "H" denotes horizontal).

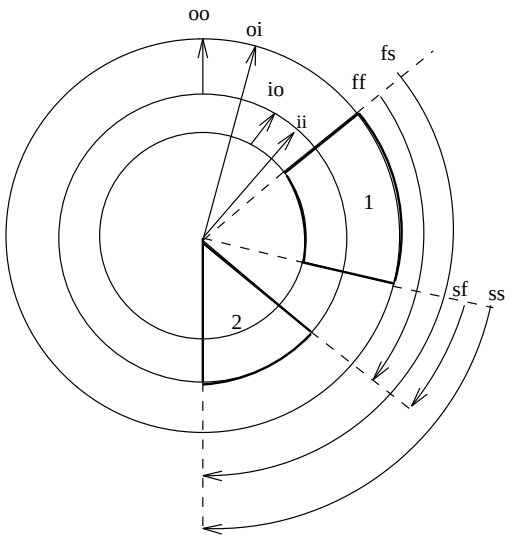

Figure 5: Spatial relationship between pairs of sectors.

lc_1 : *lengthConstraint*[firstObj → 1;
secondObj → 1;
firstLand → me;
secondLand → "headpoint";
distance → 0;
kind → horizontal]

The head of the arrow can be positioned in the following way (see Figure 6(i)):

lc_2 : *lengthConstraint*[firstObj → 1;
secondObj → 2;
firstLand → "headpoint";
secondLand → h;
distance → 0;
kind → absolute]

Positions on the Screen

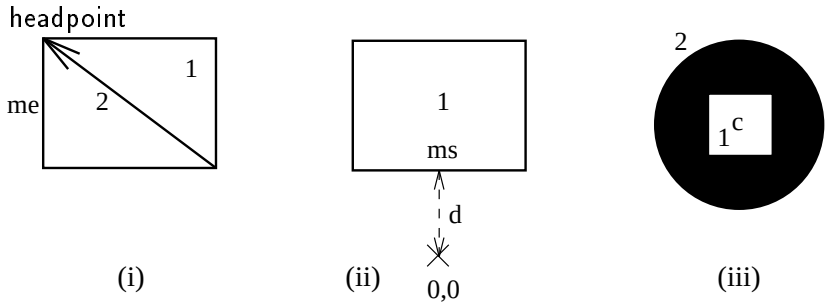

Figure 6: (i) Position of anchor point; (ii) Position on the screen; (iii) Overlapping of objects.

Vertical and horizontal positions on the screen can be defined by considering the origin to be a visual object (see Figure 6(ii)).

lc_3 : *lengthConstraint*[firstObj → origin;
secondObj → 1;
firstLand → c;
secondLand → ms;
distance → d;
kind → vertical]

Overlapping of Objects

The following object describes the overlap of two objects with identities 1 and 2, where object 1 is to be placed on top of object 2.

o_1 : *lengthConstraint*[firstObj → 1;
secondObj → 2;
top ↠ 1]

The relative position of the two objects is defined by a length constraint, as follows.

lc_4 : *lengthConstraint*[firstObj → 1;
secondObj → 2;
firstLand → c;
secondLand → c;
distance → 0;
kind → absolute]

This is illustrated by Figure 6(iii).

4 Syntax of the U-term Language

4.1 Abstract Syntax

In the U-term language there are four kinds of symbols: prototypical symbols, key symbols, macro symbols and generic symbols.

Prototypical Symbols

A prototypical symbol consists of:

symbol name. The symbol name uniquely identifies the symbol.

symbol class. Symbol classes include shapes (like box), lines (like straight line), and text.

attribute pairs (attribute name, attribute value). Classes have attributes, whose values specify further the objects. For example the boundary of a box can be solid or dashed. boundary is an attribute name and its attribute values include solid and dashed.

set of landmark pairs (name of landmark, landmark type). Each symbol has a set of landmarks. Landmarks can be of two types: cartesian and polar. Some symbol classes can have landmarks of the two types. Such is the case of circle. There is a special landmark called any, which refers to any landmark on the boundary of the symbol. Its format can be cartesian or polar.

set of anchor point pairs (name of anchor point, anchor point format). This set may be empty. The names of the anchor points are strings chosen by the user. The format of the anchor points can be cartesian or polar.

In Figure 7 we give some examples of prototypical symbols using the syntax of F-logic. From this point on we do not write the attribute landmarks and its value, and we only write the attribute anchorpoints when the set is non-empty.

```
b : box[boundary → solid, density → opaque, color → black, texture → plain,
        landmarks →→ {[name → c, format → cartesian], ..., [name → mw, format → cartesian]},
        anchorpoints →→ {[name → "headpoint", format → cartesian]}]
s : sector[boundary → dashed, density → transparent, color → black, texture → plain,
          landmarks →→ {[name → c, format → polar], ..., [name → outradius, format → polar]},
          anchorpoints →→ {}]
t : text[value → "Draw", font → roman, size → 12pt]
```

Figure 7: Examples of abstract U-terms for prototypical symbols.

Key Symbols

Table 1 summarizes the syntax of the key symbols in the U-term language. In addition, the key symbols defbox, refbox, labeled refbox, and grouping box have a set-valued attribute of pairs of anchorpoint pairs. In the table, $<$ *any symbol but defbox* $>$ comprises any prototypical symbol, but also any macro symbol or generic symbol that we describe below.

```
d : defbox[contains → b]
l : labeledRefbox[name → "X", labelname → "barChart"]
t : lengthconstraint[firstobject → l, secondobject → l,
                     firstlandmark → ms, secondlandmark → mn,
                     distance → α Y, kind → vertical]
```

Figure 8: Examples of abstract U-terms for key symbols.

Symbol Class	Attributes Names	Attribute Value Types
defbox	contains	$\{<$ *any symbol but defbox* $>\}$
refbox	name	$<$ *string* $>$
labeled refbox	name	$<$ *string* $>$
	label name	$<$ *string*$>$
grouping box	contains	$\{<$ *any symbol but defbox*$>$ $\}$
origin	landmark	origin
topright	landmark	topright
length constraint	first object	$<$ *prototypical symbol* $>$ | $<$ *refbox* $>$
	second object	$<$ *prototypical symbol* $>$ | $<$ *refbox* $>$
	first landmark	$<$ *landmark* $>$ | $<$ *anchorpoint* $>$
	second landmark	$<$ *landmark* $>$ | $<$ *anchorpoint* $>$
	distance	$+$ ($<$ *expression* $>$) | $-$ ($<$ *expression* $>$) | abs ($<$ *expression* $>$)
	kind	horizontal | vertical | absolute | radial | angular
overlap constraint	first object	$<$ *prototypical symbol* $>$ | $<$ *refbox* $>$
	second object	$<$ *prototypical symbol* $>$ | $<$ *refbox* $>$
	top	$\{<$ *prototypical symbol* $>$ | $<$ *refbox* $>\}$

Table 1: Key symbols.

Figure 8 gives examples of abstract U-terms for key symbols. The following observations complement the summarized information in Table 1.

- The objects that are values for the attribute contains have to be physically contained in the boxes that form a defbox or a grouping box. This means that if the object overlaps but is not within the boundaries then it is not part of any of those constructs. A defbox cannot be contained in another defbox or in a grouping box.

- origin denotes the point of (0,0) coordinates in the active display area (this is the area where the graphical objects that are specified by the U-term will be displayed).

- topright denotes the point that has greatest X-coordinate (right) and greatest Y-coordinate (top) in the active display area. origin, topright and length constraints can be used to place objects anywhere in the active display area.

- distance can be either positive, negative or the absolute value of an expression (the distance is restricted to be absolute when the kind of the length constraint is absolute).

- An expression can be either a simple expression or a proportionality expression. Examples of simple expressions were given in Section 3.3. The proportionality expression α, is used, for example, to specify that a visual attribute (e.g., height of a bar in a bar chart) is proportional to the value of a database attribute (e.g., salary). The syntax of the proportionality expression is as follows:

$$<\mathit{proportionalityExpression}>:: \alpha\ <\mathit{simpleExpression}>\ [\mathsf{min} :< \mathit{num} >][\mathsf{max} :< \mathit{num} >]\ [\mathsf{sum} :< \mathit{num} >]$$

  The value for max (min) is the greatest (smallest) value in the active domain of the specified attribute. The value of sum is the sum of all the values of an attribute. For example, sum can be used to display pie charts, where it is necessary to know the sum of all the values being represented, so that the sum of the angular widths of each sector (the angular width of a sector being proportional to its value) can be made equal to 2π.

Macro Symbols

There are spatial relationships between objects that result from combining two or more length constraint objects. They are not therefore indispensable, but their existence can simplify the user's task of assembling a picture. Next, we present in some detail the macro symbols: nest and grid alignment. Other macro symbols include: Cartesian position (to specify the placement of a graphical object on the plane), zero distance (to specify that the absolute distance between two points is zero), and macroarrow (to specify that an arrow goes from a point in an object to another point in another object).

Nest

The nest relationship is a macro for four length constraint objects, as outlined in Figure 9. Figure 10

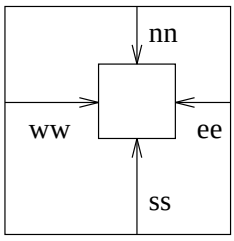

Figure 9: nest relationship.

shows the four length constraint objects that an instance of nest is equivalent to.

$$
\begin{aligned}
&C : \mathsf{nest}[\mathsf{firstobject} \rightarrow O_1,\ \mathsf{secondObject} \rightarrow O_2,\\
&\qquad \mathsf{nn} \rightarrow \geq 0,\ \mathsf{ss} \rightarrow \leq 0,\ \mathsf{ee} \rightarrow \geq 0,\ \mathsf{ww} \rightarrow \leq 0]\\
&\equiv\\
&\{Z_1 : \mathsf{lengthConstraint}[\mathsf{firstObject} \rightarrow O_1,\ \mathsf{secondObject} \rightarrow O_2,\\
&\qquad\qquad \mathsf{firstLandmark} \rightarrow \mathsf{mn},\ \mathsf{secondlandmark} \rightarrow \mathsf{mn},\\
&\qquad\qquad \mathsf{distance} \rightarrow \geq 0,\ \mathsf{kind} \rightarrow \mathsf{vertical}],\\
&\ Z_2 : \mathsf{lengthConstraint}[\mathsf{firstObject} \rightarrow O_2,\ \mathsf{secondObject} \rightarrow O_1,\\
&\qquad\qquad \mathsf{firstLandmark} \rightarrow \mathsf{ms},\ \mathsf{secondLandmark} \rightarrow \mathsf{ms},\\
&\qquad\qquad \mathsf{distance} \rightarrow \geq 0,\ \mathsf{kind} \rightarrow \mathsf{vertical}],\\
&\ Z_3 : \mathsf{lengthConstraint}[\mathsf{firstObject} \rightarrow O_1,\ \mathsf{secondObject} \rightarrow O_2,\\
&\qquad\qquad \mathsf{firstLandmark} \rightarrow \mathsf{me},\ \mathsf{secondLandmark} \rightarrow \mathsf{me},\\
&\qquad\qquad \mathsf{distance} \rightarrow \geq 0,\ \mathsf{kind} \rightarrow \mathsf{horizontal}],\\
&\ Z_4 : \mathsf{lengthConstraint}[\mathsf{firstObject} \rightarrow O_2,\ \mathsf{secondobject} \rightarrow O_1,\\
&\qquad\qquad \mathsf{firstLandmark} \rightarrow \mathsf{mw},\ \mathsf{secondlandmark} \rightarrow \mathsf{mw},\\
&\qquad\qquad \mathsf{distance} \rightarrow \geq 0,\ \mathsf{kind} \rightarrow \mathsf{horizontal}]\}
\end{aligned}
$$

Figure 10: nest abstract U-term and set of equivalent length constraint abstract U-terms.

Grid alignment

The principle behind grid alignment is the following: all landmarks and anchor points that are on the same horizontal (vertical) line specify points that have the same horizontal (vertical) coordinates in the picture. Figure 11 shows a specification of a bar chart and of two nodes of a linked list, where the grid alignment is enforced. The syntax of this macro symbol is GRID ON (otherwise it is GRID OFF). In Figure 11 we have emphasized the landmarks for which the vertical or horizontal positions will be enforced.

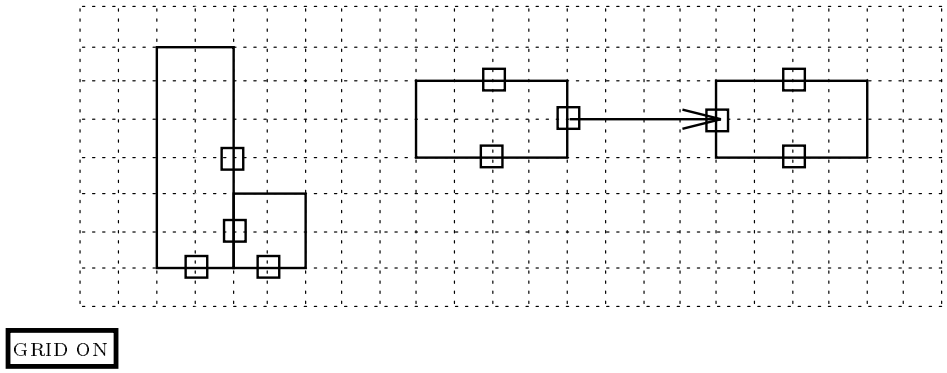

Figure 11: The grid alignment option.

Generic Symbols

Generic symbols are predefined symbols to the layout program. For example, a set of orthogonal axes (as in Figure 12(i)). The parameters for these axes, are the names of two domains, the maximum and minimum values in each of the domains, the space along each axis between each consecutive pair of "labeled ticks" The grid generic symbol is similar to axes, but there the "ticks" are replaced by sets of horizontal and vertical lines (see Figure 12(ii)).

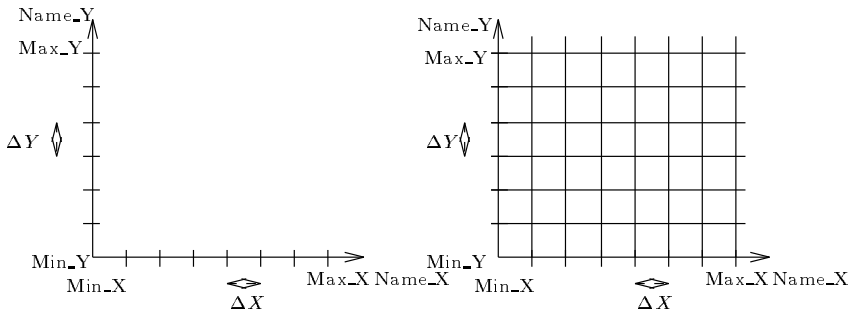

Figure 12: (i) X-Y axes; (ii) X-Y grid.

4.2 Concrete Syntax

Figure 13 exemplifies (part of) the concrete syntax of the U-term language. The U-term in the figure specifies entity-relationship (E-R) diagrams, such as the one represented in Figure 14. In the example, the diamond is specified to be centered at the point with coordinates (20,20). Attached to the diamond there is a box, and a set of (zero or more) circles. There is a defbox that contains the diamond, the set of circles, the box and two straight lines. The defbox has the following anchor points: next and previous. There is a larger box that is also connected to the diamond. This box contains a labeled refbox (with two anchorpoints), a smaller box, and a line that connects the labeled refbox and the smaller box. This picture does not contain syntactic errors, and is therefore a well-formed U-term.

The syntactic description of the picture is given in Figure 16. It takes into account the exact position of the objects in the U-term of Figure 13. The objects are described by their shape and by their extent. For example, the extent of objects such as boxes, and lines, is the minimum box that contains the object. The extent is described by the coordinates of two points in that box (as in [GR90]). The extent of circles is defined by the coordinates of the center, and the radius. Examples of the extents of two objects are given in Figure 15. The abstract syntax for the U-term of Figure 13, is given in Figure 17. There may be more

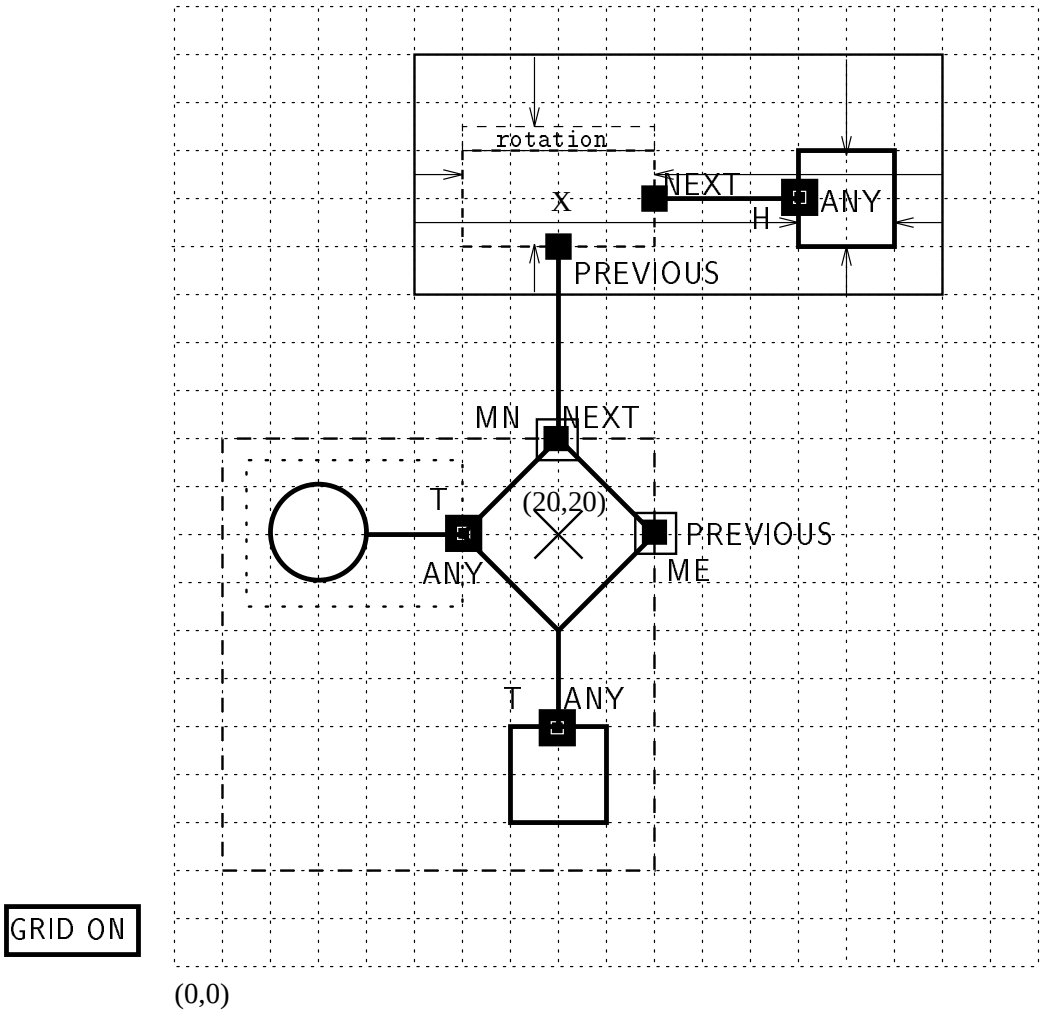

Figure 13: U-term that specifies an E-R diagram.

than one well-formed U-term that is mapped to the same abstract U-term. We can therefore partition the U-terms into equivalence classes. We say that two U-terms are *specification equivalent* if they are described by the same (up to macro symbol equivalence) abstract syntax. Note that U-terms may specify the same display without this condition being true.

Figures 13 and 14 also demonstrate another point: that it is easier to draw a visual specification than to write all the constraints in textual form.

5 Semantics

The semantics of a U-term is the set of pictures that the U-term specifies. Each U-term is associated with a visual language, and with a database object or a set of database objects in a class. A database object in the F-logic model is an instance of a class, and has methods. Each method maps one object to one object or to a set of objects (zero or more). For example, in the following term

$$\begin{aligned} &\mathit{relationship}[\text{entity\_1} \rightarrow \text{a} : \mathit{erClass}[\text{label} \rightarrow \text{"assistant"}]; \\ &\qquad \text{entity\_2} \rightarrow \text{X} : \mathit{erClass}[\text{entity\_2} \rightarrow \text{c} : \mathit{er-class}[\text{label} \rightarrow \text{"course"}]; \\ &\qquad \text{attributes} \twoheadrightarrow \{\text{h} : \mathit{erClass}[\text{name} \rightarrow \text{"hours"}]\}] \end{aligned}$$

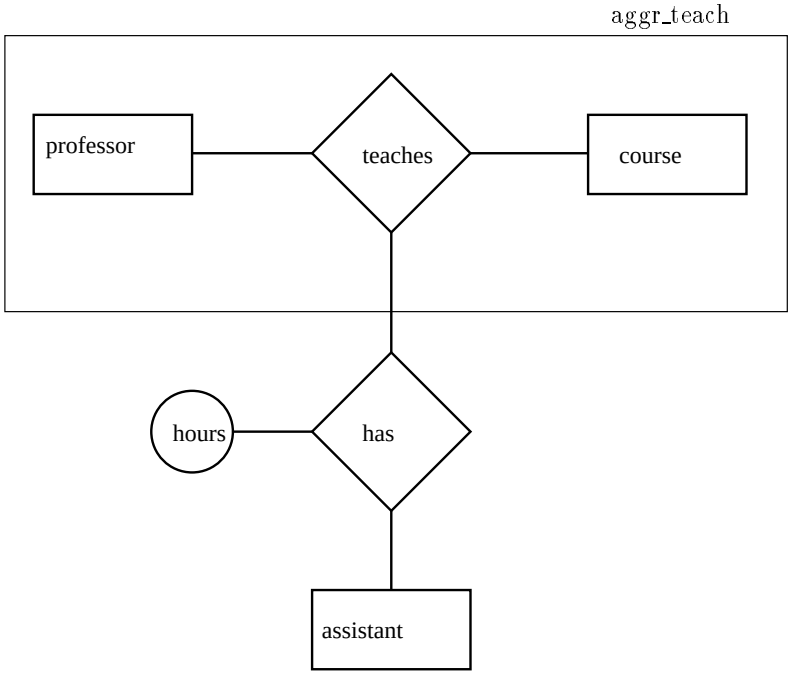

Figure 14: E-R diagram with aggregation [KS86]. The entities professor and course, and the relationship teaches form an aggregate that is treated as an entity. This E-R diagram describes information about professors that teach courses and employ teaching assistants for each of the courses for a number of hours.

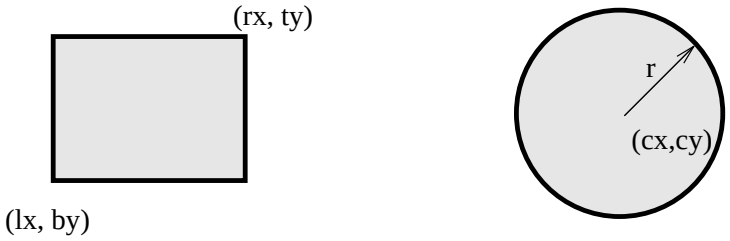

Figure 15: The extent of two graphical objects.

the method *entity_1* maps one object in the class *relationship* to one object with identity a, and *entity_2* maps one object in the class *relationship* to one object with identity X. X is a variable which ranges over objects of the class *erClass* that comprises all the objects that are relevant to an entity-relationship diagram. X can belong to *erClass* or to any of its subclasses, and its display can be defined differently depending on which class it belongs to (for example, *relationship* is a subclass of *erClass*). The display of an object of class *relationship* can be specified with the U-term of Figure 13 (if we do not attempt to display the labels).

For example, the picture of Figure 14 can be obtained if the display of the class to which X belongs is replaced by the objects that are inside the defbox of the U-term of Figure 13, rotated of 90^o (clockwise). In this case such a display is specified by the user (who has defined the visual language `rotation`). Even when the class of X is fixed, the U-term specifies a set of pictures. Each pair of pictures in such a set can differ in one of more picture attributes (e.g., the size of the diamond) that has not been fixed in the U-term.

```
box[lx → 7, by → 3, rx → 9, ty → 5, boundary → solid, width → 3]
box[lx → 13, by → 15, rx → 15, ty → 17, boundary → solid, width → 3]
box[lx → 5, by → 14, rx → 16, ty → 19, boundary → solid, width → 2]
groupingbox[lx → 1.5, by → 7.5, rx → 6, ty → 10.5]
labeledrefbox[lx → 5, by → 15, rx → 10, ty → 17.5, name → "X", labelname → "rotation"]
circle[cx → 3, cy → 9, r → 1, boundary → solid]
defbox[lx → 1, by → 2, rx → 10, ty → 11]
diamond[lx → 6, by → 7, rx → 10, ty → 11, boundary → solid, width → 2]
cartesianposition[cx → 8, cy → 9, label → "20,20"]
anchorpoint[cx → 8, cy → 11, name → "next"]
anchorpoint[cx → 10, cy → 9, name → "previous"]
anchorpoint[cx → 10, cy → 16, name → "next"]
anchorpoint[cx → 8, cy → 15, name → "previous"]
straightline[lx → 8, by → 5, rx → 8, ty → 7, boundary → solid, width → 3]
straightline[lx → 4, by → 9, rx → 6, ty → 9, boundary → solid, width → 3]
straightline[lx → 10, by → 16, rx → 13, ty → 16, boundary → solid, width → 3]
nestarrows → {arrow[lx → 7.5, by → 19, rx → 7.5, ty → 17.5],
              arrow[lx → 5, by → 16.5, rx → 6, ty → 16.5],
              arrow[lx → 7.5, by → 14, rx → 7.5, ty → 14],
              arrow[lx → 16, by → 16.5, rx → 10, ty → 16.5]}
nestarrows → {arrow[lx → 14, by → 19, rx → 14, ty → 17]
              arrow[lx → 5, by → 15.5, rx → 13, ty → 15.5],
              arrow[lx → 14, by → 14, rx → 14, ty → 15],
              arrow[lx → 16, by → 15.5, rx → 15, ty → 15.5]}
gridon
```

Figure 16: Description of the U-term of Figure 14.

6 Examples

The examples in this section are applications of graph drawing [DETT93] to the U-term language. The U-term of Figure 18 is used to specify the display of a binary tree in the following way (see figure 19):

- Downward, that is, a child is placed below its parent.
- The X-coordinate of each vertex is proportional to the inorder rank of that vertex.
- The Y-coordinate of each vertex is proportional to the distance of that vertex to the root. This layout ensures that no vertices overlap or edges cross.

The following points explain some of the features of Figure 18:

- The picture specifies the following prototypical symbols: two arrows, an invisible refbox (in light grey), and a circle (in darker grey).
- The invisible box expresses the concept of bounding box (that is the tightest box) around the whole tree. This bounding box contains most of the landmarks and anchorpoints that are needed to define these binary trees.
- If the objects of the class that is associated with the U-term have the following form:

  $tree$[leftSubtree → l : $tree$; rightSubtree → r : $tree$]

  then each refbox refers to the defbox that (by default) contains the whole U-term.

```
a : box[boundary → solid, width → 3]
b : box[boundary → solid, width → 3]
c : box[boundary → solid, width → 2]
d : labeledrefbox[name → "X"; labelname → "rotation";
                  anchorpoints →→ {[name → "next", format → cartesian],
                                   name → "previous", format → cartesian]}]
e : circle[boundary → solid]
f : defbox[contains → {a, e, g},
           anchorpoints →→ {[name → "next", format → cartesian],
                            name → "previous", format → cartesian]}]
g : diamond[boundary → solid]
h : groupingbox[contains → {e, i}]
i : straightline[boundary → solid]
j : straightline[boundary → solid]
k : straightline[boundary → solid]
l : straightline[boundary → solid]
m : cartesianposition[object → g, landmark → c, x_position → 20, y_position → 20]
n : lengthconstraint[firstObject → e, secondObject → i, firstLandmark → any, secondLandmark → t,
                     distance → 0, kind → absolute]
o : lengthconstraint[firstObject → i, secondObject → g, firstLandmark → h, secondLandmark → mw,
                     distance → 0, kind → absolute]
p : lengthconstraint[firstObject → a, secondObject → j, firstLandmark → any, secondLandmark → t,
                     distance → 0, kind → absolute]
q : lengthconstraint[firstObject → j, secondObject → g, firstLandmark → h, secondLandmark → ms,
                     distance → 0, kind → absolute]
r : lengthconstraint[firstObject → g, secondObject → f, firstLandmark → mn, secondLandmark → "next",
                     distance → 0, kind → absolute]
s : lengthconstraint[firstObject → g, secondObject → f, firstLandmark → me, secondLandmark → "previous",
                     distance → 0, kind → absolute]
t : lengthconstraint[firstObject → f, secondObject → l, firstLandmark → "next", secondLandmark → t,
                     distance → 0, kind → absolute]
u : lengthconstraint[firstObject → l, secondObject → d, landmark → h, landmark → "previous",
                     distance → 0, kind → absolute]
v : lengthconstraint[firstObject → d, secondObject → k, firstLandmark → "next",
                     secondLandmark → t, distance → 0, kind → horizontal]
w : lengthconstraint[firstObject → k, secondObject → b, firstLandmark → h,
                     secondLandmark → "previous", distance → 0, kind → vertical]
u : contains[firstObject → c, secondObject → d,
             nn →≥ 0, ss →≤ 0, ww →≥ 0, ee →≤ 0]
v : contains[firstObject → c, secondObject → b,
             nn →≥ 0, ss →≤ 0, ww →≥ 0, ee →≤ 0]
```

Figure 17: Abstract U-term for Figure 14.

- An overlap constraint is also used (denoted by the dark grey color of the circle), which specifies that the circle is the top object. In this way, arrows are specified to come out of the center of a node, but are hidden in the part that coincides with the node.
- For simplicity, we have omitted the representation of the grid in the picture.
- Length constraints (represented as dotted lines between anchor points and landmarks), specify that each parent node is horizontally placed in between its subtrees. Other configurations are possible by changing the length of the constraint that positions the root (e.g., centering the root between its

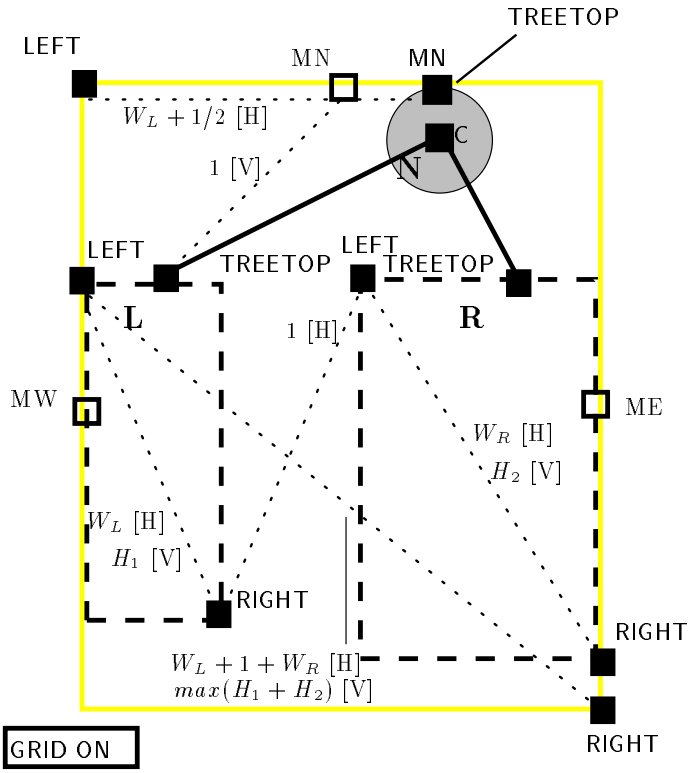

Figure 18: U-term that specifies the display of a binary tree in a downward fashion.

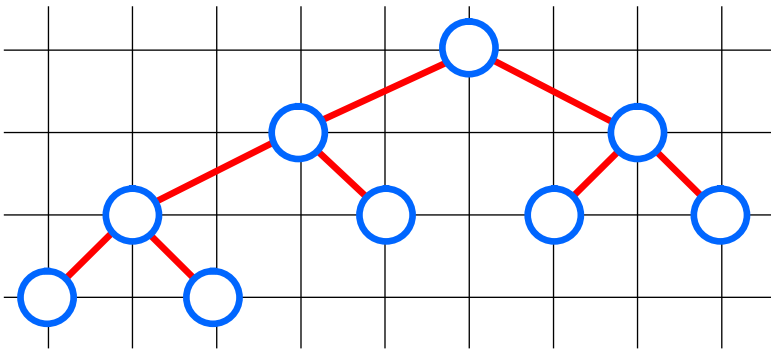

Figure 19: Layout of a binary tree specified by the U-term of Figure 18.

children). Other constraints have zero length and are specified by superposition of landmarks and anchorpoints.

- The width of the tree's bounding box is given by the sum of the widths of the bounding boxes of the subtrees plus one.
- The height of the bounding box of the tree is equal to one plus the maximum height of the bounding boxes of the two subtrees.

Another way of visualizing a binary tree is specified by the U-term in Figure 20. A node is represented by a box that contains the boxes associated with the children of that node (see Figure 21).

7 Conclusions

We have presented a new constraint-based visual language, the U-term language, to specify the display of data. This language is declarative and visual. The U-term language is geared to the display of database objects, and to fit in an object-oriented framework. The fact that the language has a visual syntax distinguishes it from other approaches such as IDEAL [Wyk82]. IDEAL, ThingLab, and the U-term language are the only proposals that deal with polar as well as Cartesian coordinates. In ways, our work is closer to (and also drew from) ThingLab [Bor81], in that the user can define visual classes by example, and the language is constraint-based and object-oriented. However, ThingLab is a visual programming language while the U-term language is a language to specify pictures that represent data. Computational power can be given to the U-term language by embedding it in DOODLE [Cru92, Cru93]. Constraint satisfaction is an important part of ThingLab, which we have not yet tackled. This could be the subject for future research. Other topics of future research include:

Language design. We would like to test the robustness of the key symbols with different sets of primitive symbols (e.g., for 3D display).

Temporal constraints. Another intriguing subject is the use of the U-term language to express temporal constraints. These could use 3D display, where the third dimension is used for time. Simple temporal reasonings can already be expressed by DOODLE, using a 2D display of temporal charts [Cru92].

Constraint query languages. Constraint query languages such as [KKR90] have a textual syntax. The U-term language could provide a visual syntax for such languages.

Graph drawing. Further work in graph drawing should address more complex problems, such as the identification of the kinds of graph layouts that can be specified by the U-term language, and characterization of the graph properties (e.g., planarity) that can be expressed (see also [CTV93]).

Design of the interface. The level of the detail of the U-terms suggest a sophisticated interface, to assist the user in the visual specification of constraints. Different levels of detail as in Figures 1(i) and 13 should also be supported.

Expressive power of DOODLE as a picture generator. The example of Figure 1(i) is relatively simple: for instance we have no recursion in the rules. An interesting topic for future research includes the precise comparison of the expressive power of DOODLE with the expressive power of visual grammars, such as the multiset grammars in [GR90].

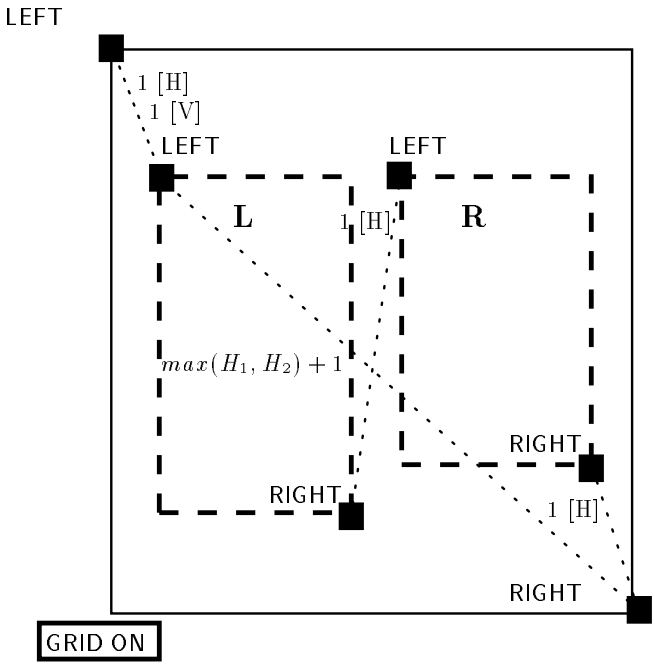

Figure 20: U-term that specifies the display of a binary tree using box containment.

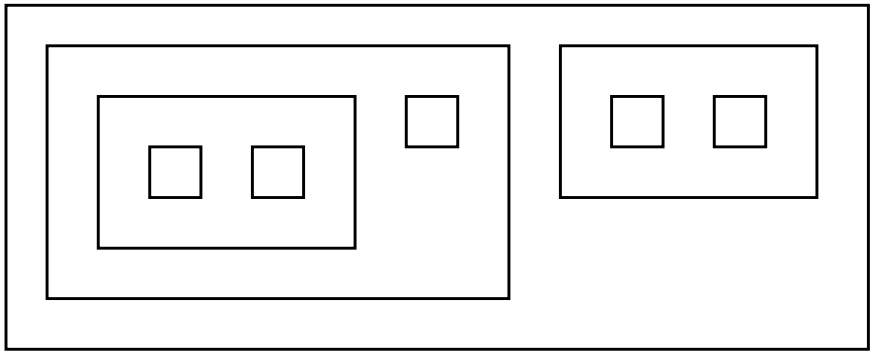

Figure 21: Containment layout of the binary tree of Figure 19 specified by the U-term of Figure 20.

Acknowledgements

Thanks to Theo Norvell for fruitful discussions that led to the definition of the visual constraint objects and to the refinement of the abstract syntax. Many thanks to Roberto Tamassia for suggesting the applications to graph drawing, and for contributing to the examples.

References

[Bor81] Alan Borning. The Programming Language Aspects of ThingLab, a Constraint-Oriented Simulation Laboratory. *ACM Transactions on Programming Languages and Systems*, 3(4):353–387, October 1981.

[CNI93] B. Chandrasekaran, N. Hari Narayanan, and Yumi Iwasaki. Reasoning with Diagrammatic Representations: A Report on the Spring Symposium. *AI Magazine*, 14(2), Summer 1993.

[Cru92] Isabel F. Cruz. DOODLE: A Visual Language for Object-Oriented Databases. In *ACM-SIGMOD Intl. Conf. on Management of Data*, pages 71–80, 1992.

[Cru93] Isabel F. Cruz. *Querying Object-Oriented Databases with User-Defined Visualizations.* PhD thesis, Department of Computer Science, University of Toronto, 1993.

[CTV93] Isabel F. Cruz, Roberto Tamassia, and Pascal Van Hentenryck. A Visual Approach to Graph Drawing. In *Graph Drawing'93*, Sèvres, France, September 1993. `The abstracts of the papers are available via anonymous ftp from wilma.cs.brown.edu (128.148.33.66), in file gd93-v2.tex.Z in the directory /pub/papers/compgeo.`

[DETT93] G. Di Battista, P. Eades, R. Tamassia, and I.G. Tollis. Algorithms for Drawing Graphs: an Annotated Bibliography. Technical report, Department of Computer Science, Brown University, March 1993. To appear in *Computational Geometry: Theory and Applications.*

[GC90] T. C. Nicholas Graham and J. R. Cordy. GVL: A Graphical, Functional Language for the Specification of Output in Programming Languages. In *Proc. IEEE Intl. Conference on Computer Languages*, pages 11–22, 1990.

[GR90] Eric J. Golin and Steve P. Reiss. The Specification of Visual Language Syntax. *Journal of Visual Languages and Computing*, (1):141–157, 1990.

[Gre83] James G. Greeno. Conceptual Entities. In Derdre Gentner and Albert L. Stevens, editors, *Mental Models*, pages 227–252. Lawrence Erlbaum Associates, Publishers, Hillsdale, N.J., 1983.

[Kam89] Tomihisa Kamada. *Visualizing Abstract Objects and Relations – A Constraint-Based Approach.* World Scientific, Singapore, 1989.

[KKR90] Paris C. Kanellakis, Gabriel M. Kuper, and Peter Z. Revesz. Contraint Query Languages. In *ACM Symposium on Principles of Database Systems*, pages 299–313, 1990.

[KLW90] Michael Kifer, Georg Lausen, and James Wu. Logic Foundations of Object-Oriented and Frame-Based Languages. Technical Report 90/14 (2-nd revision), Department of Computer Science, SUNY Stony Brook, 1990. To appear in *JACM.*

[KS86] Henry F. Korth and Abraham Silberschatz. *Database System Concepts.* McGraw-Hill Book Company, New York, NY, 1986.

[Lak87] Fred Lakin. Visual Grammars for Visual Languages. In *Proc. AAAI*, 1987.

[Tuf83] Edward R. Tufte. *The Visual Display of Quantitative Information.* Graphics Press., Cheshire, Conn., 1983.

[Wyk82] Christopher J. Van Wyk. A High-Level Language for Specifying Pictures. *ACM Transactions on Graphics*, 1(2):163–182, April 1982.