# DOODLE: A Visual Language for Object-Oriented Databases *

Isabel F. Cruz
Department of Computer Science, Univ. of Toronto,
Toronto, Ont. M5S 1A4, Canada.
isabel@db.toronto.edu

## Abstract

In this paper we introduce DOODLE, a new visual and declarative language for object-oriented databases. The main principle behind the language is that it is possible to *display and query the database with arbitrary pictures*. We allow the user to tailor the display of the data to suit the application at hand or her preferences. We want the user-defined visualizations to be stored in the database, and the language to express all kinds of visual manipulations. For extendibility reasons, the language is object-oriented. The semantics of the language is given by a well-known deductive query language for object-oriented databases. We hope that the formal basis of our language will contribute to the theoretical study of database visualizations and visual query languages, a subject that we believe is of great interest, but largely left unexplored.

## 1 Introduction

Many benefits seem to arise from the transition from relational database systems to object-oriented database systems, there are, however, issues that need to be addressed. Declarative querying of object-oriented databases is one such issue that has been attracting a great deal of attention, as a number of database researchers believe that the success of object-oriented databases is largely dependent on the existence of declarative query languages for these databases.

In this paper, our approach to this problem is visual. We introduce a visual language, whose main principle is *display and query the database with arbitrary pictures*; we call it DOODLE (*Draw an Object-Oriented Database LanguagE*). Our language extends the concept of object-oriented database querying, while using as its foundation the technology of deductive query languages for object-oriented databases.

*Financial support was given by an INVOTAN grant, by a Government of Canada Award, and by the Institute for Robotics and Intelligent Systems.



In taking a visual approach, one has first to visualize the database. This is in itself an interesting problem: the object-oriented model is not as simple as the relational data model, as it is intended to capture the semantics of the application in a more complete way; data has a complex structure, and an associated behaviour.

We want the user to be free to specify in a visual way how the data is to be visualized, that is, how *database objects* are mapped to *visual objects*. In the sequel we refer to this mapping, or more specifically to the set of conventions for obtaining pictures from data, as *visualization*. In a broader sense, a picture obtained in this way will also be referred to as a visualization.

Our system differs from other systems in that there are no pre-defined ways to display the data. This means that, for instance, *graph* is not a visualization that is previously known to the system, as in other systems [CMW88, KKD89, CM90, GPV90]. Furthermore, the user can transform the display from one visualization to another.

The following example gives an overview of some of the goals of our language, as far as display specifications are concerned. Consider a binary relation where both attributes are defined on the domain of the integers. Figure 1 suggests four possible visualizations of this relation. One of the visualizations is of the relation as a table; we made it the kernel of the picture, because it is a common representation in the relational-database world. But other visualizations are also common and

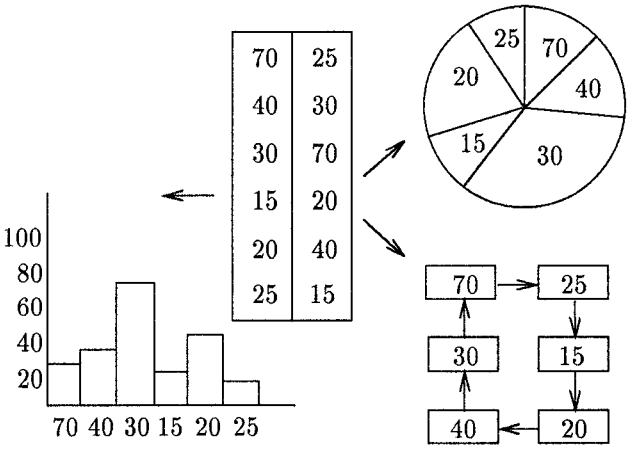


Figure 1: Visualizations of a binary relation

perhaps more useful in certain applications. We have in the same picture a bar chart, a pie chart, and a graph. Each could provide another view of the data that is more convenient for the application at hand.

To specify a visualization of the data, or a visual transformation (e.g., from a graph to a bar chart), the user writes a program in DOODLE. In this program the user can draw the pattern that describes the shapes of the visual objects. In addition, we would like the user to extract information from the database, by simple manipulation of the visual objects. In this regard, we take the approach of G$^+$ [CMW87, CMW88]: the visual query language should ideally be close to the visualization of the database. This allows the user to perceive the queries as patterns to be matched against the database. It seems natural therefore that DOODLE allows for both the specification of the display and for querying in a similar way. It is in fact our goal to make DOODLE a language with which the user can perform any kind of data manipulation (e.g., updates) by visual manipulation.

For reusability and extendibility, the visualizations are part of the database, and the language is object-oriented, so that the user can easily define new visualizations based on previously defined ones.

Another important consideration is that the language has a formal basis. This will establish the grounds for the formal study of database visualizations and visual query languages, which seems to be lacking in spite of a number of approaches to visual database querying (see [BCCL91] for a recent survey). This study will include the characterization of the expressive power of visual languages building on a rich theory for relational and object-oriented query languages.

The paper is organized as follows. In Section 2 we introduce the object-oriented model that we will be using. In Section 3 we describe the DOODLE visual language: we give an overview of the language, its syntax and its semantics, and illustrate some features of the language with examples. We conclude the paper with a comparison with related work (Section 4), conclusions, and our perspective of what remains to be done (Section 5).

# 2 The Object-Oriented Model

In this section we cover some of the object-oriented concepts that will be needed throughout the paper. We present a subset of F-logic [KL89, KLW90], since we will be using F-logic for our extensional database (facts) and to give semantics to our visual language.

### F-terms and F-rules

An *id-term* is defined inductively as follows: (1) a variable is an id-term (denoted by a capital letter); (2) a constant is an id-term; (3) if $f$ is an $n$-ary function and $t_1, \ldots, t_n$ are id-terms, then $f(t_1, \ldots t_n)$ is an id-term; $f$ is called an *object constructor*.

We consider three kinds of *F-terms* (terms in F-logic):

- *is-a terms*, of the form $o : c$ (read $o$ *is-a* $c$), where $o$ and $c$ are id-terms.
- *data F-terms*, denoted by $c[l_1 \rightarrow c_1; \ldots; l_n \rightarrow c_n]$, where $c$ is an id-term. The *labels* $l_i, 1 \leq i \leq n$ are either functional or set-valued. If $l_i$ is functional $c_i$ is a data F-term; otherwise $c_i$ is a set $\{s_1, \ldots, s_k\}$, where each $s_j, 1 \leq j \leq k$, is a data F-term. A label is of the form $a@a_1, \ldots, a_m$ where $a, a_i, 1 \leq i \leq m$, are id-terms, and $m \geq 0$ (when $m = 0$ we write $a$). If there are no labels, we write simply $c$.
- *signature F-terms* are syntactically similar to data F-terms, except for the arrows ($\rightarrow$), which are now replaced by double arrows ($\Rightarrow$), and for each $c_i$, $1 \leq i \leq n$, which is now an id-term.

It is often convenient to combine is-a terms and data F-terms, in the following way: $o : c[l_1 \rightarrow c_1; \ldots; l_n \rightarrow c_n]$ (note that each $c_i$ could be also a term of this form). From now on, we will be referring to these terms as *generalized data F-terms.*

An *F-rule* has the form $A \leftarrow B_1, \ldots, B_n$, where the literals $A, B_1, \ldots, B_n$ are F-terms or generalized data F-terms. An *F-logic program* is a set of F-rules. An *F-fact* is an F-rule with no body, which contains no variables. An *F-query* is an F-rule with no head.

### Objects and Denotations

An *object* is a database implementation of an entity (single or collective) in the world. For example we can have an object corresponding to a person or to the set of all persons. In F-logic, objects are referred to via their *denotations* or *identifiers* (*oids*), which are ground id-terms. The following are oids: john, father(john).

### Classes and the *is-a* hierarchy

A *class* in F-logic is at the same time a set of objects which share a semantic similarity, and an object. For example *person*, and *visualObject* (the set of objects that can be displayed), are classes. The most general class is the class *object*, which is the top of the *is-a* hierarchy. We can also write john:*person*. Because john is a leaf in the *is-a* hierarchy, we say that john is an instance of *person*.

### Attributes, Methods, and Signatures

An *attribute* is a function that maps the set of objects into a domain. Syntactically attribute names are 0-arity labels. In the following F-terms

john[firstName $\rightarrow$ "John", children $\rightarrow$ {sue, ann}]
box[height $\rightarrow$ 5]

firstName, children and height are attribute names. Note that children is a set-valued attribute.

A *method* is an attribute that expects arguments. Syntactically, a method name is a label with arity 1 or greater. In the following, salary is a method with argument 1991 and height is also a method with argument window_1.

john[salary@1991 $\rightarrow$ 40K]
box[height@window_1 $\rightarrow$ 3]

Signatures use signature F-terms and express typing constraints. For example,

$$person[\text{firstName} \Rightarrow string,\ \text{children} \Rightarrow \{person\}]$$
$$visualObject[\text{height@}window \Rightarrow integer]$$

The same method name can take different number of arguments, and be functional or set-valued as long as the respective signatures allow for that.

#### Inheritance

In F-logic signatures are inherited monotonically, that is, if $c : c'$ then $c$ inherits the signatures of $c'$. On the other hand, F-logic attributes and methods are inherited nonmonotonically, that is, attributes and methods defined for subclasses override those inherited from superclasses.

# 3 The Visual Language

## 3.1 Overview

As briefly described in Section 1, we want the visual language to be able to specify, query, and transform visualizations of data. We opted for the following guidelines in the design of the language:

- The language is visual.
- The language is declarative. This is an important feature in query languages for relational databases, and recently much effort has been put in this direction in the development of query languages for object-oriented databases (see survey in [Cru90]).
- The language can express important object-oriented features, such as the ones mentioned in Section 2.
- The semantics is given by F-logic. In this way we get a formal and sound basis for our visual language. Also, as described in [KLW90] and briefly outlined in Section 2, F-logic accounts for most object-oriented features. Other interesting features of F-logic are its second-order syntax and capability to express meta-queries.
- The system is extensible and very few of its features are hard-wired. We want the user to specify visualizations with very few primitives, and to easily build new visualizations based on previously defined ones.

We define a *visualization* to be the set of conventions for obtaining pictures from data. The visualizations we define are hard-wired as little as possible. The visualization of a relation as a graph is not built-in as in other systems [CMW88, KKD89, CM90, GPV90], but can easily be programmed. One of the novel aspects of the visual language we propose is that the user can specify her own visualizations.

## 3.2 The Syntax

### 3.2.1 Visual Programs, Visual Terms, and Visual Rules

A *visual program* is a set of visual rules. Each *visual rule* is composed of a left-hand side and a (possibly empty) right-hand side. The order of the rules is immaterial. A *visual query* is similar to a visual rule but consists only of the right-hand side.

Figure 2 depicts a visual rule. The double vertical

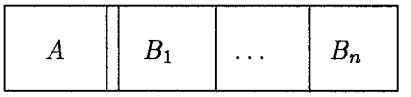


Figure 2: Visual rule

bar separates the left-hand side (the head of the rule) from the right-hand side (the body of the rule). The literals $A, B_1, \ldots, B_n$ are D-terms (for DOODLE terms). A *D-term* is one of the following:

**F-term** These are terms from F-logic (see Section 2). They are depicted as strings of characters.

**V-term** F-terms, such as f : *family*[father → john, mother → mary, children : {sue, ann}] can be expressed in a visual syntax as in Figure 3. We call these terms *V-terms* (for *visual* F-terms). V-terms are inspired by the depiction of TEDM objects [ZM88]; there are however some differences, one being the existence of set-valued attributes. In V-terms, the visual counterpart for the braces in the set notation of F-terms is provided by a dotted rectangle.

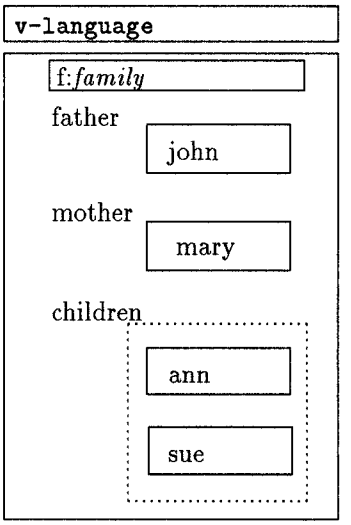


Figure 3: V-term

**U-term** These are the *user-defined* terms. A U-term is syntactically a set of icons. This set of icons is put together by the user to describe a visual pattern.

### 3.2.2 Visualization Languages

There are two pre-defined visualization languages : the language of F-terms and the language of V-terms. The former we call **f-language** and the latter we call **v-language**. There is also any number of visualization languages that are user-defined and make use of U-terms. To make clear which language is being used for the term we provide its name inside a box (e.g, `f-language` , `v-language` , `myGraph` ), near the corresponding term.

The following F-rule expresses the NEST operation of [RKS88] on a supplier-part database:

$$\begin{aligned} \mathrm{id(X)} : \mathit{nestedType}[\mathrm{supplier} \rightarrow \mathrm{X}; \mathrm{parts} \rightarrow \{\mathrm{Y}\}] \\ \longleftarrow \mathrm{W} : \mathit{sp}[\mathrm{supplier} \rightarrow \mathrm{X} : \mathit{string}; \\ \mathrm{part} \rightarrow \mathrm{Y} : \mathit{part}]. \end{aligned}$$

This rule "groups" all the parts Y that are supplied by X. The object constructor id(X) indicates that the oid of the object of class *nestedType* is determined by X, achieving the effect of existential quantification [CW89, KW89]. The visual program for the NEST F-rule is given in Figure 4. The visual program consists of a single visual rule whose terms are V-terms.

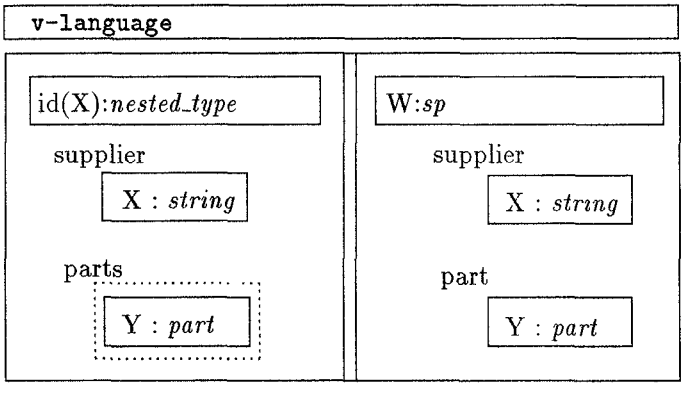


Figure 4: Visual program for NEST

### 3.2.3 User-defined Visualizations

We illustrate a user-defined visualization with an example from software engineering. Given a set of instances of class *block* and set of instances of class *contains*[outer ⇒ *block*, inner ⇒ *block*] we want to display them as a graph. A set of instances of class *contains* defines a binary relation R.

In our example we consider only two subclasses of *block*: *module* and *procedure*. Figure 5 shows a visual program for specifying the visualization **softGraph**. We would like to obtain a graph whose nodes are visual representations of either modules or procedures. The modules should be displayed as diamonds and the procedures as squares. Instances of *contains* should be displayed as arrows that will connect visual representations of modules and procedures.

We use U-terms to specify how the data objects are to be visualized. To this end, U-terms contain patterns (sets of icons) that are drawn by the user. In this example the pattern for objects X:*module* is a diamond and the pattern for objects of class Y:*procedure* is a square. The pattern for objects of class E:*contains* is an arrow which is drawn from a dashed rectangle with label X to a dashed rectangle with label Y. Labelled dashed rectangles are called *refboxes* (for *reference boxes*).

The refboxes in the third visual rule of Figure 5 stand for the visual representations of objects X and Y. The visualizations for the subclasses *module:block* and *procedure:block* are defined in the first and second rules of Figure 5. Unlabelled dashed rectangles are called *defboxes* (for *definition boxes*). Defboxes indicate which parts of the visual display of *modules* and *procedures* are referred to by refboxes. In these cases, the entire display is surrounded by a dashed rectangle. The concepts of defbox and refbox are analogous to the concepts of procedure definition and procedure call.

The last rule of the visual program states that the visual rendition of a set of objects R is the set of the visual renditions of the members of the set. We use a dotted rectangle to denote a set. There is a refbox, which is labelled by E, in the last rule, and a corresponding defbox in the third rule.

Next, we note some syntactic conventions of visual programs. For each term we identify the visualization language that we are using. As all the left-hand sides of the rules use the same language (**softGraph**) we can write it only once on top of the program, and likewise for the language that is used on the right-hand side of

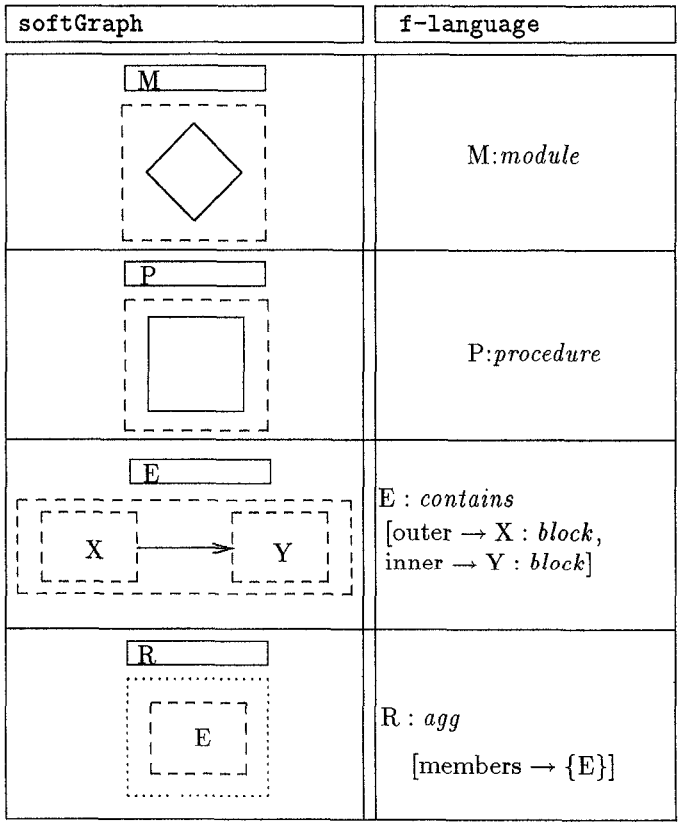


Figure 5: Visual program that defines **softGraph**

the rules (**f-language**).

The name of the object whose visualization is defined by the U-term is put inside a box together with the class to which it belongs (for example M:*module* , or simply M , when no ambiguity arises).

The icons that are used to draw the patterns of Figure 5 are simple. Some of them have pre-defined meanings in the language (such as the dashed rectangles and the dotted rectangles). The others can be chosen by the user out of an alphabet of icons, or can be assembled using simpler icons;[1] these icons describe the shape of the visual objects that will be drawn by a layout manager. It is conceivable that in real applications the user would have (or would choose) to supply more information about the visual objects (e.g., the dimensions of the visual objects with diamond shape). The information to supply depends on the particular icons and visual primitives that are available.

Part of the extendibility of the system (which we listed as one of our goals in Section 3.1) comes from the fact that our visual language is object-oriented. For instance, a dashed rectangle in the figure refers to the visualization of an object regardless of its class. The specific display depends on the class the object belongs to. If the object is of class *module* or *procedure* then its display is specified. If for example the visualization of objects of class *module* had not been defined then their display could be inherited from an ancestral class of *module*. The dashed rectangles also provide for genericity. For instance, the visual definition of *contains* allows for any visual objects at the end points of the instance of *edge*.

There are other aspects regarding the extendibility of the language that we leave for Section 3.4.

## 3.3 Semantics

We give semantics to visual programs by showing how they can be translated to F-logic programs. A detailed account of the F-logic semantics is given in [KLW90]. In what follows, we will give an informal description of the meaning of the F-logic programs that we obtain for a DOODLE visual program.

The translation from a visual program to an F-logic program is performed in the following way. Each visual rule is translated into an F-rule. An F-term remains the same in the translation, and a V-term gets translated to its underlying F-term. If a V-term or an F-term is on the head (resp. body) of the rule, they will be part of the head (resp. body) of the F-rule to which the visual rule gets translated.

Figure 6 shows the F-logic program that is obtained from translating the visual program of Figure 5 into an F-logic program. For each object of classes *module*, *procedure*, *contains*, the *display* method is a function from the objects of each of these classes to objects of a visual class (respectively *diamond*, *square*, *arrow*). Note that for the last rule where a set of visual objects was specified visually (by means of a dotted rectangle), there is an instance of the class *visAgg* (for *visual aggregate*) which groups the individual visual objects.

The visualization language that is being defined is **softGraph**, which is the argument to the display method. The display method passes the specification of the display to a layout manager. The class of the values of the display method is either *visualObject*, which is the most general class in the hierarchy of visual objects, or a descendant class of *visualObject*. The classes *diamond*, *square* and *arrow* are descendant classes of *visualObject* and should be recognized by the layout manager.

Note that *visualObject:object* and *object* is the most general class in our model. There can be more than one language in a rule (or in the corresponding visual rule), although we only allow one language for each term. The method *display*@**softGraph** is overloaded, since it is defined (differently) for each class.

In view of the above example, we now give a more detailed description of how the translation is achieved. We distinguish two cases, depending on whether there is a defbox in the visual rule.

If the U-term contains no defbox, then the translation from a U-term to F-logic, yields an F-term of the following form:

$$X[\text{display@}\mathtt{v} \rightarrow \alpha]$$

where X is the object whose visualization is being defined, **v** is the visualization name, and $\alpha$ is a visual object or set of visual objects.

When the U-term has a defbox the F-term will have the following form:

$$X[\text{display@}\mathtt{v} \rightarrow \alpha; \text{defbox@}\mathtt{v} \rightarrow \beta]$$

where $\alpha$ is the visual object or set of visual objects that are specified by the U-term and $\beta$ is the visual object or set of visual objects that are inside the defbox.

If the U-term contains refboxes then there will be an F-term for each of the refboxes. These F-terms have the following form:

$$Y[\text{defbox@}\mathtt{v} \rightarrow \gamma]$$

where Y is the label of the refbox and $\gamma$ is a variable that will be used to refer to the defbox of Y. If the U-term that contains the refboxes is in the body of the visual rule, then the F-terms for the refboxes will be in the body of the F-rule. If the U-term that contains the refboxes is in the head of the visual rule, then the F-terms for the refboxes will again be in the body of the F-rule. The informal idea behind this translation is that the visualization for refboxes has been defined in another visual rule, so that it is added to the right-hand side of the rule, that is to the list of "known visualizations".

## 3.4 More Examples

### 3.4.1 Visualizing Cyclic Data

Cyclic data is one of the problems that one faces when displaying data. The picture in Figure 7 is just a simple example of cyclic data. Suppose that data is stored in objects with type *node*[successor $\Rightarrow$ *node*]. In Figure 8 we show the visual program with which instances of the class are to be displayed as a graph. The corresponding F-logic program is given in Figure 9.

[1] Editors in the style of *xfig* or *MacDraw* can be designed for the purpose of designing new icons.

M[display@**softGraph** → vis(M) : *diamond*; defbox@**softGraph** → vis(M)] ⟵ M : *module*.
P[display@**softGraph** → vis(P) : *square*; defbox@**softGraph** → vis(P)] ⟵ P : *procedure*.
E[display@**softGraph** → vis(E) : *arrow*[from → U, to → V]; defbox@**softGraph** → vis(E)]
⟵ E : *contains*[outer → X, inner → Y],
X[defbox@**softGraph** → U : *visualObject*],
Y[defbox@**softGraph** → V : *visualObject*].
R[display@**softGraph** → vis(R) : *visAgg*[visMembers → {X}]]
⟵ R : *agg*[members → {E}], E : *contains*[defbox@**softGraph** → X : *visualObject*].

Figure 6: F-logic program for **softGraph**

N[display@**nodeGraph** → {c(N) :*circle*, e(N) : *arrow*[from → X, to → Y]}; defbox@**nodeGraph** → c(N)]
⟵ N : *node*[succ → S : *node*],
N[defbox@**nodeGraph** → X : *visualObject*],
S[defbox@**nodeGraph** → Y : *visualObject*].

Figure 9: F-logic program for **nodeGraph**

### 3.4.2 Other Visualizations

So far we have concentrated only on graph visualizations. Another representation for data is given by *higraphs* [Har88]. We define a less complex formalism that we call simple higraph. A *simple higraph* is a tuple

$$H = (B, \sigma)$$

where $B$ is a finite set of elements, called *blobs*, and $\sigma$ is a *subblob function* defined as

$$\sigma : B \rightarrow 2^B$$

which is assumed to define a tree.

The following set of objects of class *inclusion*, represents a simple higraph:

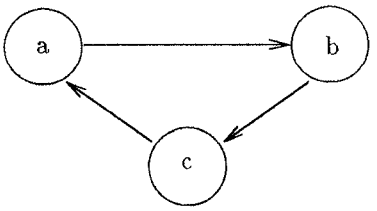


Figure 7: Cyclic data

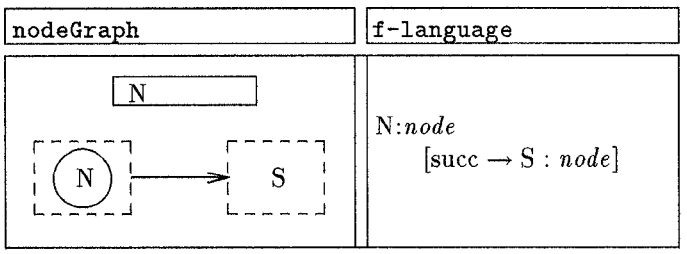


Figure 8: Visual program for **nodeGraph**

$o_1$ : *inclusion*[first → a, second → {b, c}].
$o_2$ : *inclusion*[first → b, second → {c}].

As in [Har88], we would like to display these objects as in Figure 10. Figure 11 shows the visual program that

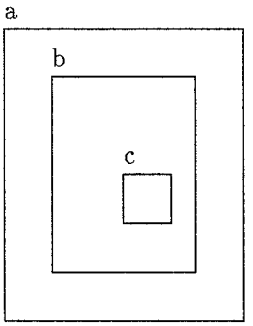


Figure 10: Simple higraph

achieves this. The U-term on the left-hand side of the second rule specifies that X contains a *visual aggregate* (visually represented as a dotted rectangle) of the visual objects that depict Y. Visually the U-term stands for geometric containment between the visual representations of X and of all the Ys. Therefore, there may be other blobs or dotted rectangles between blob X and the dotted rectangle around Y. Having both this consideration in mind and that each database object is visualized only once, we see that we obtain the picture of Figure 10.

Figure 12 shows the F-rules for the visual program of Figure 11.

### 3.4.3 Visual Transformations

In the previous sections we used visual programs to express how F-terms can be visualized. With our visual

H[display@**simpleHG** → vis(H) : *blob*; defbox@**simpleHG** → vis(H)] ⟵ H : *object*.
I[display@**simpleHG** → vis(I) : *nest*[outside → U, inside → agg(I) : *visAgg*[visMembers → {V}]]]
⟵ I : *inclusion*[first → X, second → {Y}],
X[defbox@**simpleHG** → U : *visualObject*],
Y[defbox@**simpleHG** → V : *visualObject*].

Figure 12: F-logic program for **simpleHG**

language it is also possible to transform visualizations into other visualizations. We illustrate this capability with the visual program of Figure 13, which specifies how to transform from the **softGraph** visualization to the **simpleHG** visualization. In the **simpleHG** visualization we want all objects of class *blob* (or of its subclasses *module* and *procedure*) to be displayed as blobs, independently of their representation in the **softGraph** visualization.

Figure 14 shows the F-logic program that corresponds to the visual program of Figure 13.

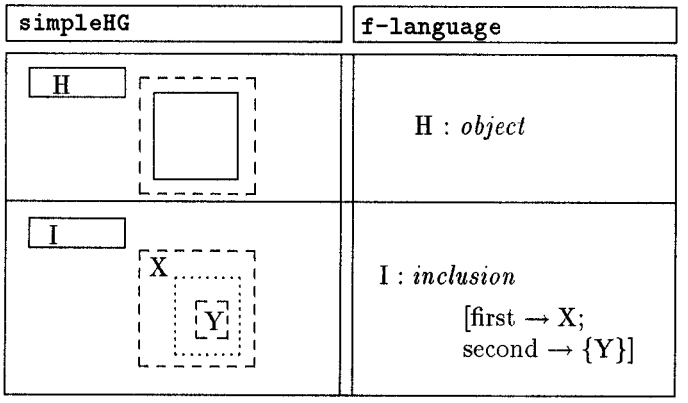


Figure 11: Visual program for **simpleHG**

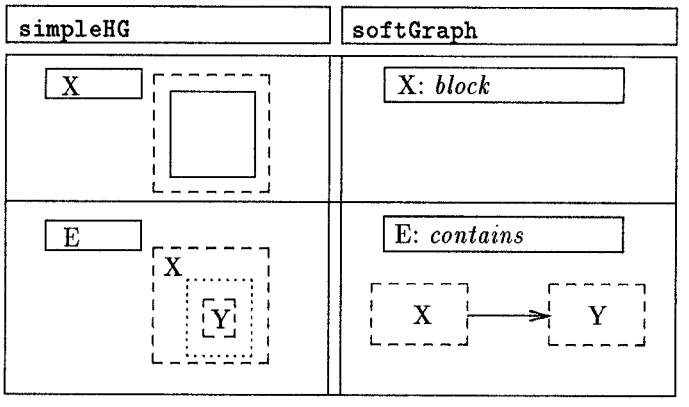


Figure 13: Visual program for the **softGraph** to **simpleHG** transformation

### 3.4.4 Using the Hierarchy of Visualizations

Visualizations can be organized in a hierarchy. For example, we can define a visualization that is like **softGraph**, except for the display of the objects of type *contains*, which are to be displayed as double arrows. If we call this visualization **newGraph**, we specify **newGraph:softGraph** and write a visual rule similar to the visual rule of Figure 5, except that we use a double arrow on its left-hand side. This example illustrates overriding. But we can also define a new visual rule to display objects of class *calledBy*, e.g., as simple higraphs.

### 3.4.5 Queries and Meta-Queries

The transitive closure query has become a "typical example" of a recursive query. Consider that we have a graph generated by the visual program of Figure 5. We would like to produce another graph where we will connect nodes X and Y by a dotted edge, if there is some path from X to Y (a path is a sequence of one or more solid edges). The visual query is shown in Figure 15. We note that we need a way to specify the number of dotted edges that will be created between every pair of nodes X and Y in the graph that depicts the transitive closure. Should its number depend on the number of paths that exist between X and Y or should there be a single edge independent of the number of such paths? We choose the latter possibility, and create a new oid, tc(X,Y), which has the intended meaning.

The F-logic program in Figure 16 gives the semantics for the visual program of Figure 15. We consider that the class *tcContains* has been previously defined in the class lattice.

With F-logic it is possible to express meta-queries. As outlined in Section 1, we want to be able to display the data using different visualizations. It is therefore possible that there will be more than one visual object associated with an instance of a class, and that the user would like to know all possible visualizations for the objects of a class *c*. In F-logic this can be expressed as the following rule

displaysOf(*c*)[displays → {W}]
⟵ X : *c*[display@W → Y : *visualObject*].

The following rule, on the other hand, groups all classes for which a visualization **v** has been defined.

classesWhere(**v**)[classes → {*C*}]
⟵ X : *C*[display@**v** → Y : *visualObject*].

X[display@**simpleHG** → vis(X) : *blob*; defbox@**simpleHG** → vis(X)] ⟵ X : *block*.
E[display@**simpleHG**] → vis(E) : *nest*[outside → U : *visualObject*,
inside → agg(E) : *visAgg*[visMembers → {V : *visualObject*}]]
⟵ E : *contains*[display@**softGraph** →
*arrow*[from → X : *visualObject*, to → Y : *visualObject*]],
R : *object*[defbox@**softGraph** → X : *visualObject*],
S : *object*[defbox@**softGraph** → Y : *visualObject*],
R : *object*[defbox@**simpleHG** → U : *visualObject*],
S : *object*[defbox@**simpleHG** → V : *visualObject*].

Figure 14: F-logic program for the **softGraph** to **simpleHG** transformation

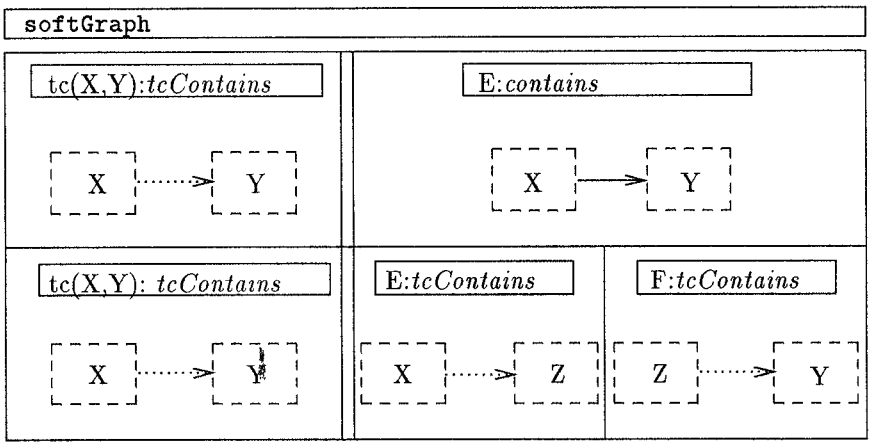


Figure 15: Visual transitive closure program

How to express the above meta-queries in a visual way (without using V-terms) is an open issue. We would like for example to obtain a visual program similar to the visual program of Figure 5, by asking the query: "For which classes have we defined the representation **softGraph**?".

# 4 Related Work

The departure point for our work comes from the G$^+$/Graphlog project [CMW88, CM90, CCM91]. The visualization for this system is graph-oriented, and the queries are themselves graphs, which are seen as patterns to be matched against the database (nodes and edges in the query graph map to nodes and paths in the database, respectively). However, the visualization is pre-defined and the model is relational.

Graph-oriented query languages have been proposed for object-oriented models [KKD89, GPV90]. In [Cru90] we show how G$^+$ can be used for modelling and querying object-oriented databases. Again, these models have a pre-defined graph representation as a basis.

PROTEUS [AEM86] is an interface system that is built on top of an object-oriented database language, which is deductive and declarative. In this system there is no pre-defined visualization: Instead, the user can define and use different visualizations. Visualizations are stored in the system as objects (which represent rules), and the system is extendible. One of the differences between PROTEUS and our visual language is that we specify visualizations visually, while in PROTEUS the display is specified textually, but more importantly with our language the user can design the query language herself.

The following approaches are not intended for database querying. Their main concern is the display of data. In that, they have common points with DOODLE. They are however not designed to display object-oriented data, nor do they make use of object-oriented technology.

GVL [GC90] is a language for the specification of conceptual views (e.g., linked lists) of programming language structures (e.g., sets of records). As in our language the user specifies the views in a visual way.

The approach in [Kam89] is concerned with the display and layout of data. The user specifies the layout by giving a set of Prolog terms. Prolog functors represent pre-defined predicates between graphical objects.

Mackinlay [Mac86] studies the automatic display of relational information. Data can be represented as plot charts, bar charts, etc., using different primitive graphical languages (although some of them, for example for the display of pie charts, are not completely developed).

tc(X, Y) : *tcContains*[display@**softGraph** → vis(E) : *dottedArrow*[from → U, to → V] ]
←—E : *contains*[defbox@**softGraph** → Z : *arrow*[from → U, to → V],
X : *object*[defbox@**softGraph** → U : *visualObject*],
Y : *object*[defbox@**softGraph** → V : *visualObject*].
tc(X, Y) : *tcContains*[display@**softGraph** → vis(E, F) : *dottedArrow*[from → U, to → V] ]
←—E : *tcContains*[display@**softGraph** → R : *dottedArrow*[from → U, to → S],
F : *tcContains*[display@**softGraph** → T : *dottedArrow*[from → S, to → V],
X : *object*[defbox@**softGraph** → U : *visualObject*],
Z : *object*[defbox@**softGraph** → S : *visualObject*],
Y : *object*[defbox@**softGraph** → V : *visualObject*].

Figure 16: F-logic program for transitive closure

# 5 Conclusions and Future Work

In this paper we introduced DOODLE, a new language for object-oriented databases. DOODLE is firstly a visual query language for object-oriented databases (analogous to the textual letter-logic query languages). As such, it provides a pre-defined visualization suitable for all classes of object-oriented data.

With DOODLE, the user can define her own visualizations, on a class by class basis, thus creating tailored displays of data. These user-defined visualizations, like the pre-defined one, can be used in queries, and when defining other visualizations. New visualizations can be easily defined using inheritance of visualizations, and the user can visually specify the transformation of visual data from one visualization to another visualization. The semantics of the language is given by F-logic. F-logic incorporates object-oriented mechanisms such as overloading, and inheritance, thus providing our language with flexibility and extendibility.

We give next a brief description of some of the problems we are investigating. This research will appear as part of the author's PhD dissertation.

We would like to express visually more complex transformations, and to test DOODLE with a variety of visual formalisms, such as pie charts, plot charts, temporal charts, etc.[Tuf90]. We present an example which illustrates some of the ongoing work. In Figure 17, the user specifies an iterative transformation from a graph visualization representing flights between Rome and Toronto (Figure 17 (ii)) to a temporal chart visualization (Figure 17 (iii)). The first rule of the program in Figure 17 (i) says that a node of the graph should be placed on the axis that represents the place of departure or of arrival. The second rule says that nodes on an axis are placed according to a scale. The third rule specifies that the origins of the two axes are to be placed at the same Y-coordinate.

In this paper we have not addressed layout issues, which arise in practise. One way to restrict the display is by using constraints or even a hierarchy of constraints [BMMW88], but how these constraints can be visually specified (or incorporated in F-logic) is a subject for future research.

We are also investigating queries on visual objects and on the encoding of their display. These queries retrieve information that is not in the database, but is introduced by the display process.

There are features of the language that we would like to quantify in terms of their expressiveness. For example, we would like to compare the expressiveness of a language that can only specify graph visualizations with a language that can only specify temporal charts. Notice that Figure 17 (iii) conveys more information at a glance than Figure 17 (ii) if we were to choose a flight from Rome to Toronto based on the elapsed time between departure and arrival.

A starting point to the study of expressiveness of visualizations has been taken in [Mac86]. With DOODLE, visualizations, visual queries, and transformations can be seen as extensions to object-oriented database querying. In addition, the language has a formal basis, because its semantics is given by F-logic. We therefore believe that with DOODLE we can undertake the theoretical study of database visualizations and visual query languages. This study can make use of the rich theory of expressive power of query languages. This is just one of the many interesting issues that we believe need to be tackled so that visual manipulations of the data can become a useful reality.

### Acknowledgements

I would like to thank Alberto Mendelzon for discussions and for raising important questions. Thanks to Michael Kifer, Theo Norvell, and the referees for valuable suggestions and a most careful reading of a previous draft of the paper. Michael also clarified some aspects of F-logic. Theo was a source of inspiration and ideas when mine seemed to falter.

# References

[AEM86] T. Lougenia Anderson, Earl F. Ecklund,Jr., and David Maier. PROTEUS: Objectifying the DBMS User Interface. In *Intl. Workshop on Object-Oriented Database Systems*, 1986.

[BCCL91] C. Batini, T. Catarci, M. F. Costabile, and S. Levialdi. Visual Query systems. Technical Report RAP.04.91, Università degli Studi di Roma, La Sapienza, Dipartimento di Informatica e Sistemistica, March 1991.

[BMMW88] Alan Borning, Michael Maher, Amy Martindale, and Molly Wilson. Constraint Hierarchies

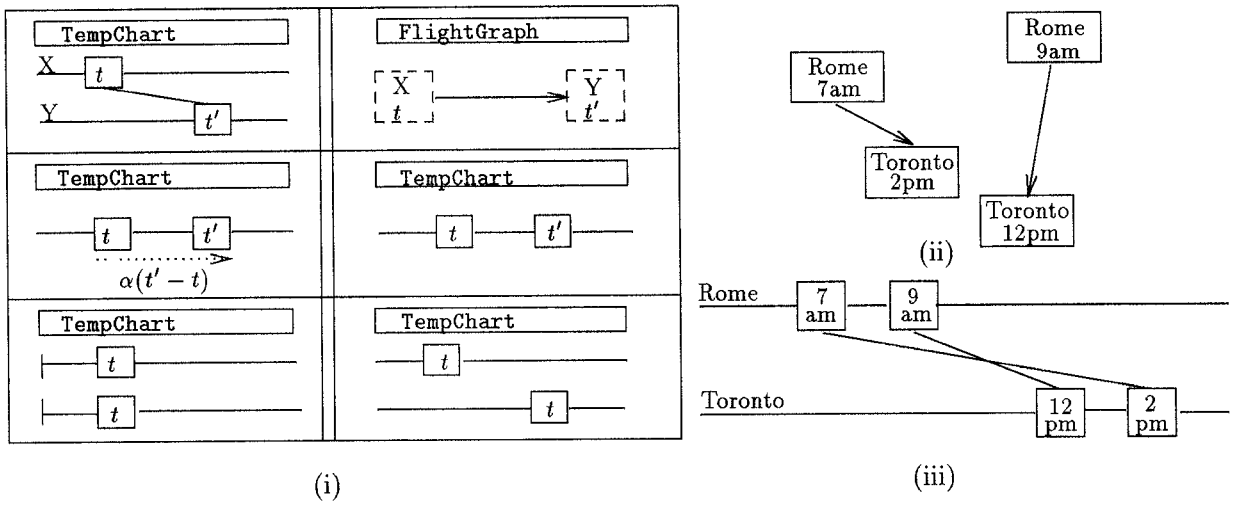


Figure 17: Graph to temporal chart transformation


and Logic Programming. Technical Report 88-11-10, Computer Science Department, University of Washington, November 1988.

[CCM91] Mariano P. Consens, Isabel F. Cruz, and Alberto O. Mendelzon. Visualizing Queries and Querying Visualizations. *SIGMOD RECORD, Special Issue on Advanced User Interfaces*, 21(1), March 1991.

[CM90] Mariano P. Consens and Alberto O. Mendelzon. Graphlog: a Visual Formalism for Real Life Recursion. In *ACM Symposium on Principles of Database Systems*, pages 404–416, 1990.

[CMW87] Isabel F. Cruz, Alberto O. Mendelzon, and Peter T. Wood. A Graphical Query Language Supporting Recursion. In *ACM-SIGMOD Intl. Conf. on Management of Data*, pages 323–330, 1987.

[CMW88] Isabel F. Cruz, Alberto O. Mendelzon, and Peter T. Wood. $G^+$: Recursive Queries Without Recursion. In *Expert Database Conference*, pages 355–368, 1988. Also in Larry Kershberg, editor, *Expert Database Systems*, The Benjamin/Cummings Publishing Company, Inc., Redwood City, California, pages 645-666, 1989.

[Cru90] Isabel F. Cruz. Declarative Query Languages for Object-Oriented Query Languages. In F. H. Lochovsky, editor, *Office and Data Base Systems Research '89*, pages 92–130. Technical Report CSRI-238, June 1990.

[CW89] Weidong Chen and David Scott Warren. C-Logic of Complex Objects. In *ACM Symposium on Principles of Database Systems*, pages 369–378, 1989.

[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.

[GPV90] Marc Gyssens, Jan Paredaens, and Dirk Van Gucht. A Graph-Oriented Object Database Model. In *ACM Symposium on Principles of Database Systems*, pages 417–424, 1990.

[Har88] David Harel. On Visual Formalisms. *Communications of the ACM*, 31(5):514–530, May 1988.

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

[KKD89] Kyung-Chang Kim, Won Kim, and Alfred Dale. Cyclic Query Processing in Object-Oriented Databases. In *IEEE Intl. Conference on Data Engineering*, pages 564–571, 1989.

[KL89] Michael Kifer and Georg Lausen. F-Logic: A Higher-Order Language for Reasoning About Objects, Inheritance, and Scheme. In *ACM-SIGMOD Intl. Conf. on Management of Data*, pages 134–146, 1989.

[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.

[KW89] Michael Kifer and James Wu. A Logic for Object-Oriented Logic Programming (Maier's O-Logic Revisited). In *ACM Symposium on Principles of Database Systems*, pages 379–393, 1989.

[Mac86] Jock D. Mackinlay. Automatic Design of Graphical Presentations. Technical Report STAN-NCS-86-1138, Department of Computer Science, Stanford University, 1986.

[RKS88] Mark A. Roth, Henry F. Korth, and Abraham Silberschatz. Extended Algebra and Calculus for Nested Relational Databases. *ACM Transactions on Database Systems*, 13(4):389–417, December 1988.

[Tuf90] Edward R. Tufte. *Envisioning Information.* Graphics Press., Cheshire, Conn., 1990.

[ZM88] Jianhua Zhu and David Maier. Abstract Objects in an Object-Oriented Data Model. In *Intl. Conference on Expert Database Systems*, pages 3–16, 1988.