Research Report LSI-98-31-R, Software Department (LSI), Technical University of Catalonia (UPC), June 1998. A shorter version of this paper has been presented at DEXA'98 (Conf. on Databases and Expert Systems Applications).

Defining and Translating Visual Schemas for Deductive Databases

JORDI PUIGSEGUR [1,2] JOAN A. PASTOR [3] JAUME AGUSTÍ [1]

{jpf,agusti}@iiia.csic.es, pastor@lsi.upc.es

1 Artificial Intelligence Research Institute – CSIC
Campus UAB, 08193 Bellaterra, Catalonia, European Union

2 Visual Inference Laboratory – Indiana University
Lindley Hall 215, Bloomington IN 47405, USA

3 Software Department – Technical University of Catalonia
Campus Nord UPC, 08034 Barcelona, Catalonia, European Union

June 8th 1998

Abstract

We present a visual language expressive enough to capture an important subset of First Order Predicate Logic as well as its straightforward translation to the logic-based paradigm of deductive databases. We use the diagrams of our language to represent all the components of a deductive database schema: base predicates, derived predicates with their deductive rules, and integrity constraints. Our diagrams are grounded on two powerful visual metaphors: Venn/Euler diagrams and graphs, familiar to most designers; they are formal and independent of the application domain; they emphasize basic forms of logic description, the diagrammatic syntax being closer to the semantics; and they have a simple translation to Horn clauses. Thus, we aim at a situation where the generality of deductive databases would be fostered by the expected greater usability of visual schema languages in the hands of a wider group of practitioners.

1 Introduction

Deductive databases, also known as logic databases, result from decades of research in the areas of logic, databases, logic programming and artificial intelligence (see [19]). Fortunately, they seem to be evolving from a research concept into a practical tool. Indeed, the large amount of theoretical research devoted to this field has not only penetrated current relational DBMSs, but is also inspiring several of their future extensions. Furthermore, this research is now materializing in some deductive DBMS prototypes and commercial products (see [29]). Deductive databases extend relational databases by allowing for the representation and management of more general forms of 'application semantics'. Besides data explicitly stored in base predicates, comprehensive views permit the handling of additional forms of derived data, and more powerful integrity constraints can be used to care for overall data consistency. This generality of deductive databases is a consequence of their underlying theory: First Order Predicate Logic.

However, it is precisely for this same reason that deductive databases share with other areas, such as logic programming, an important problem: the inherent difficulty that many practitioners encounter when working directly with logical expressions. In fact, we believe that this may have

been one of the main reasons for their slow diffusion. Certainly, few people in industry enjoy working directly with logic and, hence, we need more natural forms of communication which can be translated (sometimes automatically) into logic. In this way, both generality and usability may be obtained. This paper aims to be a first step towards this goal within the context of deductive databases, where we explore the visual definition of their schemas. In doing so, we draw from our prior results —here adapted and extended— regarding the definition of a visual logic programming language [23, 1, 24, 22].

The increasing power of computers and specially the improvement of their graphical capabilities has fostered in the last years the study of new forms of expression within formal languages: *visual languages*. These languages offer a completely new way to interact with computers that may now also be applied to existing paradigms like logic programming and deductive databases. In particular, we believe that it is possible to construct diagrammatic languages that benefit from the generality of these paradigms. We aim at languages that are independent from any particular application domain and which emphasize visually some semantic and pragmatic features of logic instead of its mathematical syntax patterned after Boolean algebra. Hopefully, their exploration will help us agree in the future on some standard ways to use logic, that is, the pragmatics of logic.

So far, by the hand of information systems design methodologies (sometimes with their associated CASE tool), some few visual formalisms have made their way through wide industrial software practise; entity-relationship and data-flow diagrams (in many versions) being among the most popular ones [30]. On the other hand, there are plenty of research proposals offering various forms of visual languages for representing software systems and their application domain, from conceptual modeling [6] to business requirements engineering [14] and enterprise modeling [7]. Given the nature of the above tasks, their proposed visual languages are mostly special-purpose and mainly centered in visually modeling the static dimension of the modeled system; dynamic aspects are either left out or operationally modeled. Other more declarative approaches with visual systems representation, such as [15, 31], usually limit themselves to textual definitions for some behavioral aspects of the system such as derived information and general consistency conditions. In all the above cases, translating visual representations to logic is not always obvious and is often partial. This fact complicates the use of logic-based paradigms and tools, such as logic programming and deductive databases, for purposes like automated system validation, verification, implementation and execution.

Without pretending to offer a visual language suited to all the above high-level tasks, in this paper we propose a visual language expressive enough to capture an important subset of First Order Predicate Logic as well as its straightforward translation to the logic-based paradigm of deductive databases. We use the diagrams of our language to represent all the components of a deductive database schema: base predicates, derived predicates with their deductive rules, and integrity constraints. These diagrams possess the following characteristics:

- They are grounded on two powerful visual metaphors: Venn/Euler diagrams and graphs, familiar to most designers.
- They are formal, that is, unambiguously defined and interpreted, and independent of the application domain.
- They emphasize basic forms of logic description, like recursion, and have a simple translation to Horn clauses.

Among other graphical metaphors that are being proposed to represent logic, a whole community is growing from the seminal work of [26] to form the new research area of Conceptual Graphs.

From the visual language point of view, the main difference between our approach and theirs consists in our option to represent relations (sets of tuples) by set operations (membership, inclusion, union, intersection) instead of their explicit representation of the corresponding logical connectives (implication, disjunction and conjunction). We believe our option to be more designer friendly —- our syntax by means of boxes *resembles* the corresponding semantics of relations (sets of tuples)— while still very easy to translate to textual logic. However, despite the appealing features of visual languages in general, including ours, more empirical research is needed in order to compare their usability, usefulness and adequacy, both between the various existing proposals and between them and textual representations. We do not address this issue here.

The paper is organized as follows. After this introduction, Section 2 defines the kind of deductive database schemas that we address with the visual schema language presented in Section 3. Section 4 is devoted to the translation of visual into textual schemas. Section 5 describes our implemented environment for editing and translating visual schemas. While Section 6 discuses other research work that may be related to our approach, in Section 7 we present both our conclusions as well as our ideas for further work.

2 Deductive Databases Schemas Considered

We define here the kind of deductive database schemas addressed in this paper. For this purpose, we review the basic concepts of deductive databases [12, 17]. Since we concentrate on the visual representation of the database schema, irrespective of any particular database contents, we find convenient to clearly separate the definitions of database contents and database schema.

We use first-order logic as the main convention. Thus, we consider a first-order language with a set of constants, a set of variables, a set of predicate names and no function symbols. In formal definitions we will use names beginning with a capital letter for predicate symbols, and names beginning with lower-case letters for variables.

A *term* is a variable symbol or a constant symbol. We assume that the possible values for terms range over finite domains. If P is an m-ary predicate symbol and $t_1, ..., t_m$ are terms, then $P(t_1, ..., t_m)$ is an *atom*. The atom is *ground* if every $t_i (i = 1, ..., m)$ is a constant. A *literal* is defined as either an atom or a negated atom, in other words, a positive or a negative atom. A *fact* is a formula of the form $P(t_1, ..., t_m) \leftarrow$, where $P(t1, ..., tm)$ is a ground atom. In the propositional case of $m = 0$, a fact is regarded as an untermed atom, ground by definition. Facts are considered to be explicitly stored in the database, and they are said to form its *extensional* part, or database *extension*.

2.1 Base predicates

Every fact is modeled in the database schema through some *base predicate* (scheme). Since base predicates schemes describe the extensional database, they take the form of:

$$B(t_1, ..., t_m) \qquad \qquad with\ m \geq 0$$

where every term $t_1, ..., t_m$ is interpreted as a distinct variable, modeling the possible extensional term instances. A base predicate syntactically coincides with its modeled fact in the propositional case of $m = 0$. A base predicate scheme appears only in its own definition, and possibly as a condition atom in the body of deductive and integrity rules (see below). A base predicate is present in the extensional database, as a fact of the corresponding scheme.

Before providing further definitions, let us introduce the base predicate schemes corresponding to the database schema example that we will be using throughout the paper. They are shown in Fig. 1, together with their intended meaning.

Our example is (part of) a database for the *Human Resources Unit* of a hierarchically-structured organization. Besides enforcing such an organizational structure, the database is to keep track of the employees and managers assigned to the different units, and of arranged interviews between job applicants and units. Furthermore, for legal reasons, it also keeps track of residential status and the existence of criminal records for the people administered by them. Rather than to represent a real application domain, we regard this database schema as a representative example of the situations that we are able to represent with our visual schema language.

Notice that the syntax of base predicates tells us nothing of its intended meaning: a set of tuples. The standard syntax of symbolic logic was set by mathematicians in linear form, patterned after Boolean algebra. This syntax favors the interpretation of predicates as truth functions and nothing suggest their more usual interpretation as sets of tuples. Even the terms do not distinguish syntactically the different role of variables and constants. To deal with these and other similar issues by visual means is the main goal of this paper.

Base pred.	Base predicate meaning
$Unit(u)$	'u' is a unit in the organization
$Reports(u, u1)$	Unit 'u' directly reports to unit 'u1'
$Works(p, u)$	'p' works for unit 'u', i.e. 'p' is employee of 'u'
$Mng(p)$	'p' is a manager
$App(p)$	'p' is a job applicant
$Intw(p, u)$	'p' has a job interview with unit 'u'
$Cit(p)$	'p' is a citizen
$Ra(p)$	'p' is a registered alien
$Cr(p)$	'p' has some criminal record

Figure 1: Base predicates: textual definition

2.2 Derived predicates and deductive rules

A *derived predicate* (or *view*) is a scheme representing information which is not stored in the database but can be derived using deductive rules. It takes the form of:

$$D(t_1, ..., t_m) \qquad with\ m \geq 0$$

where terms $t_1, ..., t_m$ are interpreted as distinct variables. A derived predicate appears as the head of deductive rules, and possibly also in the bodies of deductive and integrity rules. Formally, a *deductive rule* is a formula of the form:

$$D(t_1, ..., t_m) \leftarrow L_1 \wedge ... \wedge L_n \qquad with\ m \geq 0 \wedge n \geq 1$$

$D(t_1, ..., t_m)$ is the *conclusion* (i.e. the derived predicate being defined) and $L_1, ..., L_n$ are literals representing defining *conditions*, which can be base, derived or evaluable predicates, possibly negated. $D(t_1, ..., t_m)$ is also called the *head of the rule* while $L_1 \wedge ... \wedge L_n$ is the *body of the rule*. *Evaluable predicates* are system predicates, such as the comparison or arithmetic predicates, operating on terms from other conditions, that can be evaluated without accessing the database. Variables in the conclusion or in the conditions of a rule are assumed to be universally quantified over the whole formula. While the terms in the conclusion must be distinct variables, the terms

in the conditions may be variables or constants. Variables not appearing in the conclusion are *existential variables*, also called local variables. The *definition* of a derived predicate $D(t_1, ..., t_m)$ is the set of all deductive rules having $D(t_1, ..., t_m)$ as their conclusion.

Fig. 2 below shows the six derived predicates of our example with their corresponding deductive rules. Employees are those people working for some unit, and are defined with a single rule having an existential variable. Two derived predicates, each one with a rule, are used to know those units having some manager and those ones with some employee who is not a manager. Subordination among organizational units is modeled with a derived predicate recursively defined with two rules, respectively dealing with direct and indirect reporting. The right-of-residence status of a person is defined using two deductive rules without any existential variable. A single rule with an existential variable is used to define job candidates.

Again, nothing in the textual syntax enforces neither the meaning of the deductive rules nor their pragmatics. By making explicit diagrammatically the membership relations of tuples to predicates and variable sharing inside rules we will make the rules more intuitive and fit for its operational interpretation [22].

Derived pred. + Deductive rules	Derived predicate meaning
$Emp(p) \leftarrow Works(p, u)$	'p' is an employee because s/he works for some unit 'u'
$HasMng(u) \leftarrow Works(p, u) \wedge Mng(p)$	Unit 'u' has some manager because at least its employee 'p' is a manager
$HasEmp(u) \leftarrow Works(p, u) \wedge \neg Mng(p)$	Unit 'u' has some non-manager employee because at least its employee 'p' is not manager
$Subordinate(u, u1) \leftarrow Reports(u, u1)$ $Subordinate(u, u1) \leftarrow Reports(u, u2) \wedge Subordinate(u2, u1)$	Unit 'u' is subordinate to unit 'u1' because it either directly reports to 'u1' or it reports to some unit 'u2' which is subordinate to 'u1'
$Cand(p) \leftarrow Intw(p, u) \wedge Unit(u)$	'p' is considered a job candidate when s/he has an interview with some unit 'u'
$Rr(p) \leftarrow Ra(p) \wedge \neg Cr(p)$ $Rr(p) \leftarrow Cit(p)$	'p' has right-of-residence if s/he is either a registered alien with no criminal record or a citizen

Figure 2: Derived predicates and deductive rules: textual definition

2.3 Integrity Constraints and Integrity Rules

Integrity constraints are general conditions that the database is required to satisfy at all times. They are used to specify unwanted database states and forbidden database changes. Accordingly, integrity constraints are either *state* (or static), when they must be satisfied in any state of the database, or *dynamic*, when they involve the evolution between two or more database states. Dynamic integrity constraints compelling only one transition between two successive states are further called *transition* integrity constraints. In this paper we only represent visually state integrity constraints and, thus, we restrict our definitions and examples to this case.

State integrity constraints can be defined in terms of base and/or derived predicates. Formally, a (state) *integrity constraint* is a closed first-order formula that the database is required to satisfy. We deal with constraints that have the form of a denial:

$$\leftarrow L_1 \wedge ... \wedge L_n \qquad with\ n \geq 1$$

where the L_i are literals (i.e. positive or negative base, derived or evaluable predicates) and variables are assumed to be universally quantified over the whole formula. More general constraints can be transformed into this form by first applying the range form transformation [11], and then using the procedure described in [18].

We associate with all integrity constraints an inconsistency (derived) predicate Ic_n, with or without terms, and, thus, they take the same form as deductive rules. We call them integrity rules. In our example we use the seven state integrity constraints shown in Fig. 3. The set of employees is a subset of the set of legal residents (Ic_1), and is disjoint with the set of applicants (Ic_2), which is a superset of candidates (Ic_3). While employees must only work for units (Ic_4), units with employees must have some manager (Ic_5). Finally, the organizational structure must be hierarchical (Ic_6 and Ic_7).

Integrity rule	Integrity constraint meaning
$Ic_1(p) \leftarrow Emp(p) \wedge \neg Rr(p)$	Every employee must have right-of-residence, i.e. it is inconsistent to have an employee without right-of-residence
$Ic_2(p) \leftarrow Emp(p) \wedge App(p)$	No employee may also be applicant, i.e. it is inconsistent to have someone being both employee and applicant
$Ic_3(p) \leftarrow Cand(p) \wedge \neg App(p)$	Every candidate must be applicant, i.e. it is inconsistent to have a candidate who is not an applicant
$Ic_4(p,u) \leftarrow Works(p,u) \wedge \neg Unit(u)$	Employees must work for units, i.e. it is inconsistent to have someone working for some area which is not considered an organizational unit
$Ic_5(u) \leftarrow Hasemp(u) \wedge \neg Hasmng(u)$	Every unit with some employee must have some manager, i.e. it is inconsistent that a unit with assigned employees does not have some manager
$Ic_6(u) \leftarrow Reports(u,u1) \wedge$ $Reports(u,u2) \wedge u1 \neq u2$	No unit may report to two different units, i.e. it is inconsistent that a unit directly reports to more than one unit
$Ic_7(u) \leftarrow Subordinate(u,u)$	No unit may be subordinate to itself, i.e. it is inconsistent that a unit directly or indirectly subordinates to itself

Figure 3: Integrity Constraints: textual definition

2.4 Deductive database schemes considered

A deductive database schema consists of three finite sets: a set B of base predicates, a set D of derived predicates with their corresponding deductive rules, and a set I of integrity constraints with their respective integrity rules.

We assume that database predicates are either base or derived predicates, but not both. Every deductive database can be defined in this form [4, 9]. While facts are said to form the *extensional* part of the database, the deductive database schema is also referred to as its *intensional* part.

In this paper we require, as usual, that the database schema is *allowed* and *stratified* [17, 3]. Rather than due to limitations of our visual schema language, these requirements are usually imposed on deductive schemas in order to avoid computational problems when processing queries and updates.

In the rest of this paper, exceptions apart, we will use the example deductive database schema resulting from the union of the above Figs. 1, 2 and 3, our goal being to visually represent the intensional part of a deductive database, i.e. its schema. We will focus on the two components that usually have more complex definitions: *derived predicates* and *integrity constraints.* We try to obtain visual representations of these components that are more visually compelling than their textual equivalents, while still completely formal.

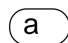
3 Our Visual Schema Language

The visual representation used is based on a set metaphor: we use Venn/Euler-like diagrams, representing sets as square boxes, and then using the graphical inclusion relation to represent set

membership and set inclusion. A predicate (either base, derived or an inconsistency predicate) can be seen extensionally as a set, i.e. a set of n-tuples where n is the arity of the predicate. We use this property to visually represent predicates as boxes and then define them by means of graphical tuples included in them.

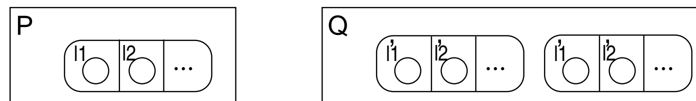
We consider a visual language semantically equivalent to the textual one defined in Section 2. Terms are either variables or constants. A variable is always represented as a circle (without any name). In a diagram —as we will see— there is no need to give names to variables, which distinguishes their role from that of constants.

Constants are represented as rounded boxes with a constant symbol inside, its name.

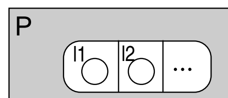


We follow the same convention as in Section 2: we use names beginning with lower case letters for constant symbols and names beginning with upper case letters for predicate symbols.

A visual *positive literal* is thus a square box with a predicate symbol P in one of its corners (inside the box) and one or several n-ary tuple included in the box, where n is the arity of the predicate symbol P. Variables and constants can be included in the box on their own when the arity of the predicate symbol is 1. A tuple is represented as a partitioned rounded box, where each partition contains a label and a term (i.e. a variable or a constant). An important difference with textual literals is that visual terms are labeled and therefore the order in which they appear in the tuple is irrelevant. This makes the syntax clearer and more flexible.



A visual *negative literal* is defined as a visual positive literal where the box is shadowed.



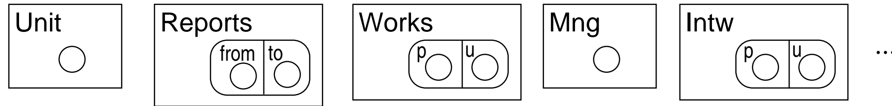
A negative literal represents the membership of the tuple to the complementary set of the predicate intension, which is equivalent to the negation of the membership.

Finally we need to define another type of literals, *visual evaluable literals*. They are used to indicate a relationship between two terms, like equality, inequality, comparison, etc. For example to represent variable equality (resp. inequality) we use a straight line (resp. a crossed straight line) joining both variables:

Other evaluable literals such as comparison are defined similarly.

3.1 Visual Base Predicates

Visual base predicates are represented using a visual literal with one tuple included, where all terms are variables. The following are examples of the visual predicate schemes corresponding to some of the base predicates introduced in Fig. 1:



For instance, the visual predicate schema corresponding to *Works* indicates that its arity is two and its two terms are labeled 'p' and 'u' (referring to 'person' and 'unit'). The rest of base predicates are visually represented in the same straightforward way.

3.2 Visual Derived Predicates and Visual Deductive Rules

Visual derived predicates are defined using *visual deductive rules* (*diagrams*). A visual deductive rule (like its textual counterpart) contains one and only one conclusion visual predicate, and one or more condition visual literals. The conclusion literal is always a box (no evaluable literals), which contains one and only one tuple and it is distinguished from the others by drawing it using thick lines. Furthermore, each diagram (visual deductive rule) is graphically enclosed in a square box to delimit its syntactical scope.

In Fig. 4 we find the derived predicates of our example deductive database schema. We would like to note some properties of the visual language. First, the fact that variables do not need to have names assigned and that variable equality is expressed using connecting lines simplifies the task of reading the rules. For instance, comparing the textual rules defining *Subordinate* with their visual counterparts: the textual ones are difficult to understand at first sight because it is necessary to follow the different variables, while in the visual ones these relations are more evident. Other features of deductive rules like existential variables are also clearly visualized using these diagrams. For instance: in the definition of $HasMng$ or $HasEmp$ in Fig. 4 one can see that the existential variables are the ones not attached to any variable included in the box corresponding to the conclusion.

3.3 Integrity Constraints

As we said in Section 2, in this paper we restrict ourselves to *state integrity constraints* expressed as denials. An integrity constraint in our visual language is a diagram (a visual deductive rule) defining an *inconsistency predicate* (Ic_n). Whenever a Ic_n predicate holds then the consistency of the deductive database has been violated. The box corresponding to the conclusion literal of a visual integrity constraint rule represents the set of elements that violate that integrity constraint. The database is consistent when all Ic_n boxes represent empty sets.

In Fig. 5 we find the visual integrity constraints equivalent to those defined textually in Fig. 3. Notice how negations are expressed using the notion of complementary set in Ic_1, Ic_3, Ic_4 and Ic_5. Notice also the Ic_6 definition: we refer to two instances of the *Reports* predicate and then use inequality to indicate that they must be different instances.

However, the main strength of our approach to visualize database schemas comes from the set metaphor we use, i.e. from representing predicates as sets of tuples. Difficult rules with various literals corresponding to the same predicate, like the one defining Ic_6, are clearly represented using sets. In other more complicated real-world schemas these advantages would be even more apparent.

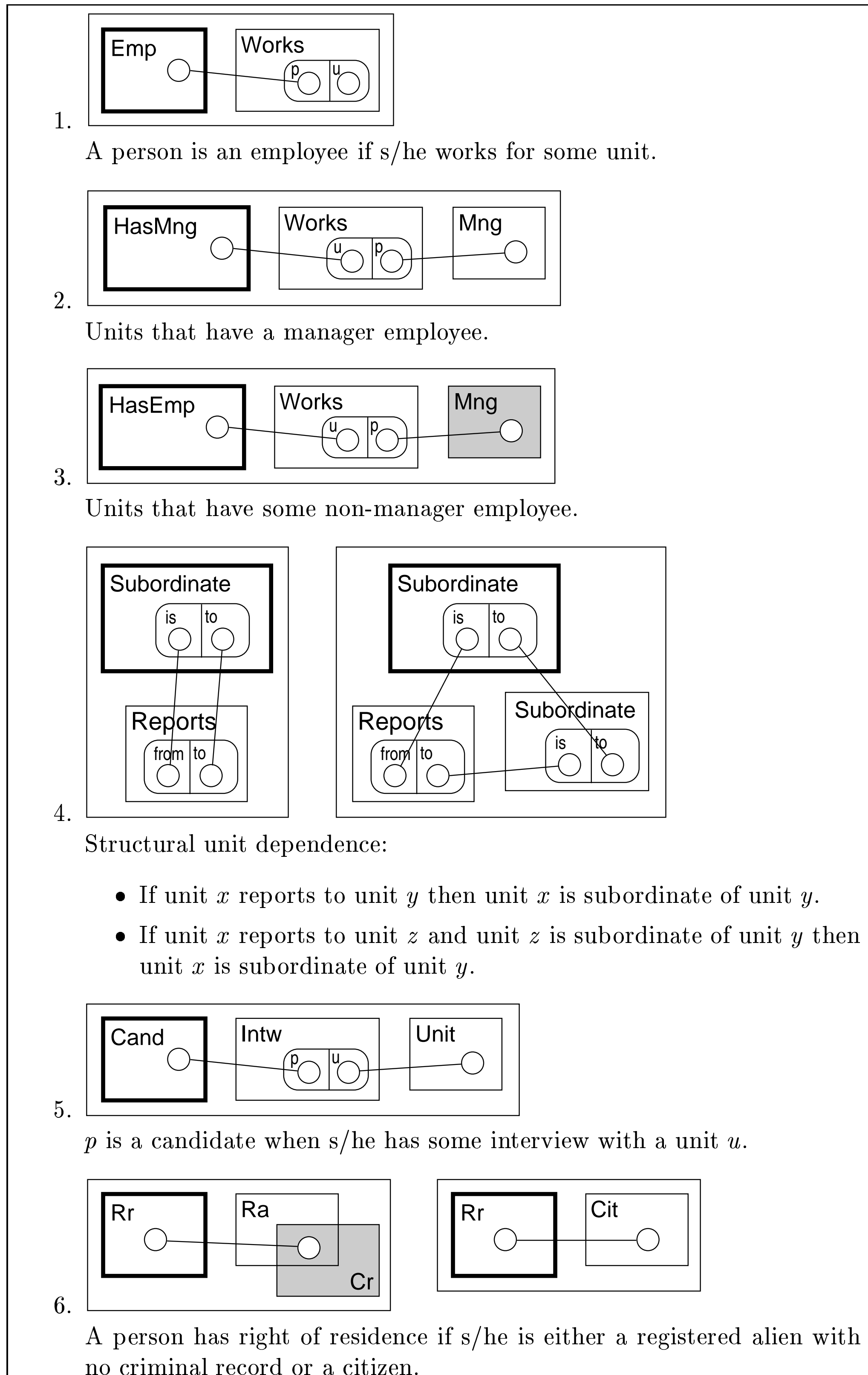


Figure 4: Derived Predicates: visual definition

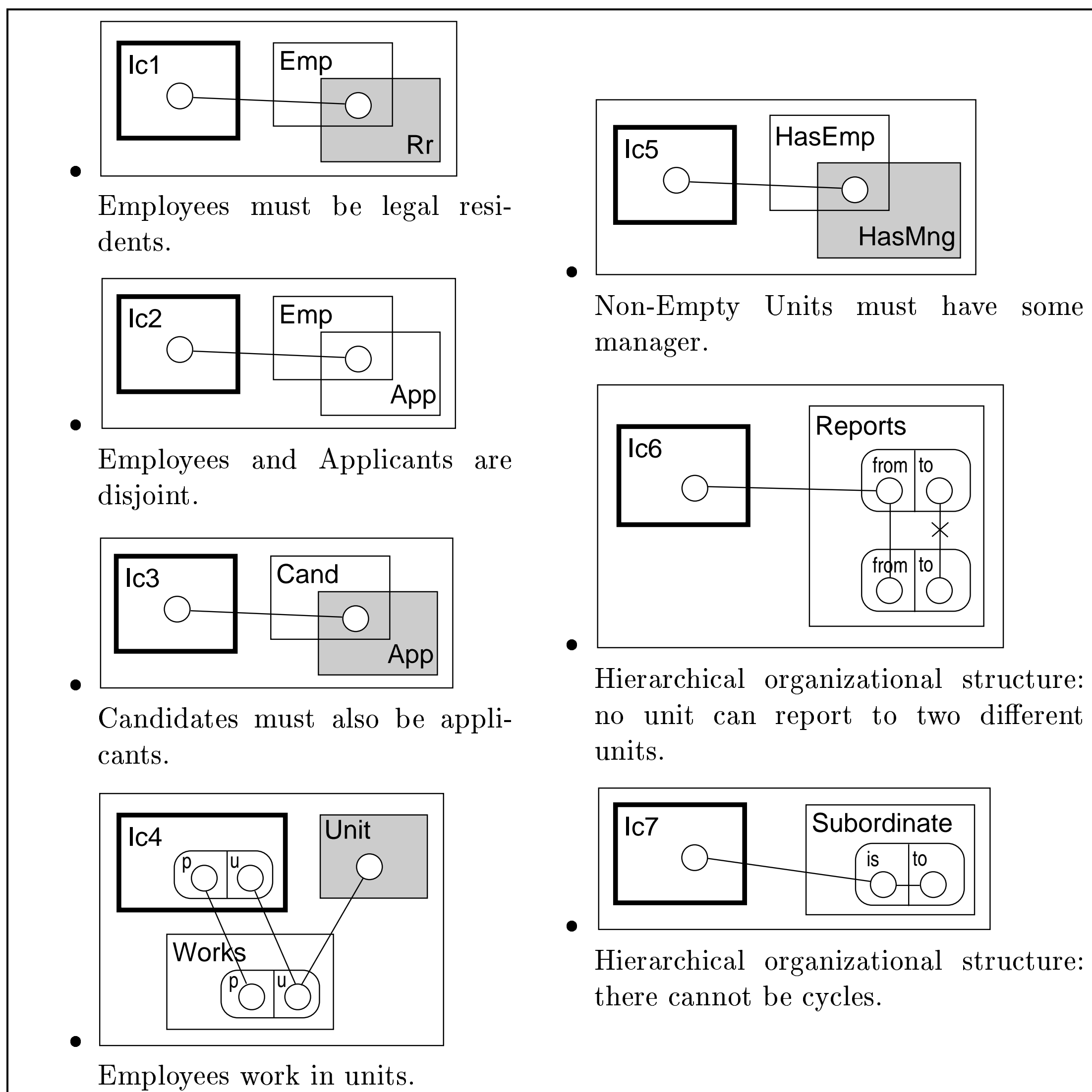
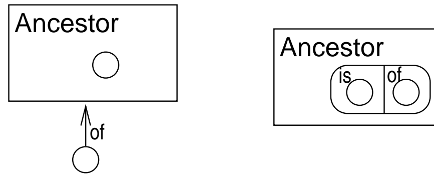


Figure 5: Integrity Constraints: visual definition

3.4 Other Features

There are other expressive potentialities of our diagrams —not used in the example used up to now— which can be useful in other situations. The most obvious of these visual constructs is graphical inclusion of boxes (sets) representing the logical implication. The unrestricted use of inclusion in our diagrams correspond to the unrestricted use of the corresponding connective (implication) and it would lead us outside the limits of deductive database schemas. Then to keep the simplicity of the computational interpretation of our diagrams we have to restrict the use of this visual construct (see [1]). In this section we show its utility by means of a different example. First we introduce a functional-like representation of predicates, alternative to the one defined up to now and then exemplify the use of inclusion.

The alternative visual representation of predicates is based on the observation that some predicates are better understood as nondeterministic functions (each input can have one or more outputs) than as relations without any preferred directionality (input / output). For instance, given a genealogical database, some predicates, like *Ancestor*, are easier to see as a function that gives us the ancestors (output) of any given person (input). In the inverse case the predicate even has a different name, that is, *Descendant*. Then it seems natural in these cases to distinguish the set of outputs (results) named by the predicate from the inputs. Compare the alternative representations of the predicate scheme *Ancestor* in the two following visual representations.

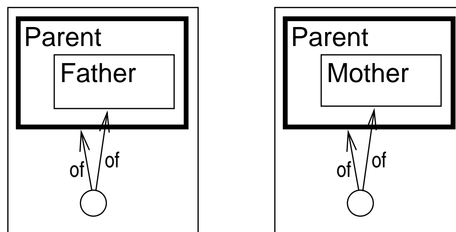


The left box is supposed to represent the set of ancestors of a person represented by a variable united to the box by an arrow. The right box represents the set of tuples constituted by descendant-ancestor pairs.

The functional representation favors the use of box (set) inclusion. For instance, the deductive rules representing that the parents of someone are his/her father and mother:

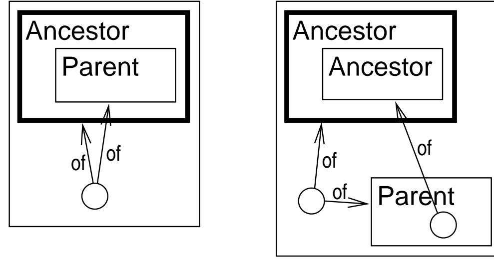
$$Parent(x,y) \leftarrow Father(x,y)$$
$$Parent(x,y) \leftarrow Mother(x,y)$$

can be represented by box inclusion as follows:



where the box inclusion corresponds to the implication: every element of the *Father* / *Mother* boxes (sets) must belong to the *Parent* box. As it was explained before, other boxes corresponding to condition visual literals can be added to the diagrams, but no box inclusion between these boxes is allowed in order to keep the visual language expressive power within computational tractable limits.

This functional representation also allows to make more evident the *recursivity* of some predicates and predicate composition in some definitions. For instance, the deductive rules for ancestor can be visualized as follows:

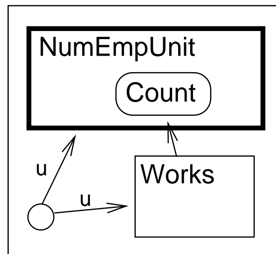


where the recursivity is shown by including ancestor inside itself. The composition of $Parent$ and $Ancestor$ (ancestors of parents are ancestors) is made evident by the graphical structure of the diagram itself.

Finally, in databases it is also usual to have *aggregation functions* which allow to perform operations over selections of given relations. In this visual language we represent them as a function whose argument is a box. Graphically they are like constants (i.e. round boxes with the name of the function inside) and its argument is indicated using an arrow going from the visual literal to the aggregate function rounded box. Let us see an example, from the Human Resources Unit example: suppose we want to calculate for each unit the number of employees of that unit, defining a new derived predicate $NumEmpUnit$, with rule:

$$NumEmpUnit(x, y) \leftarrow Unit(x) \ \wedge \ Count(\{w|Works(x, w)\}, y)$$

We use the $Count$ aggregation function, obtaining the following diagram:



4 Translating Visual into Textual Schemas

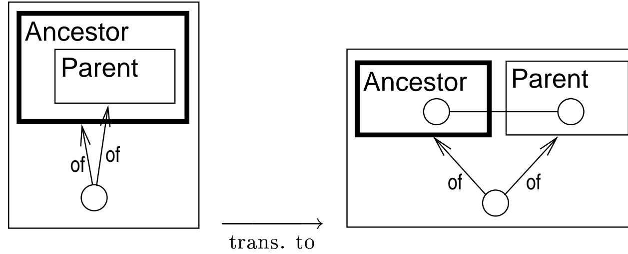
4.1 Translation Algorithm

First of all, for each predicate with arity greater than one we need to supply an extra piece of information: the correspondence between term labels (visual literals) and term ordering (textual literals), so that the visual-to-textual translation can be performed correctly. In our example we only have three predicates with more than one term and this correspondence is defined as follows:

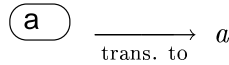
$Reports(\text{from} : x, \text{to} : y) \ \longmapsto \ Reports(x, y)$
$Works(\text{p} : x, \text{u} : y) \ \longmapsto \ Works(x, y)$
$Subordinate(\text{is} : x, \text{of} : y) \ \longmapsto \ Subordinate(x, y)$

Since we are using a syntax-directed diagram editor (see Section 5) we can assume that the input to the translation algorithm is always a syntactically correct visual database schema. Each visual deductive rule corresponds to one textual deductive rule, and is translated performing the following steps:

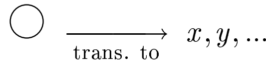
1. *Box inclusion elimination*: if there is a box inclusion the diagram is transformed into an equivalent one using only variables and no box inclusion:



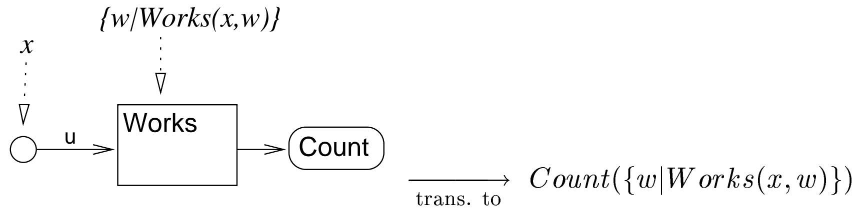
2. Each *constant* is translated into the symbol contained inside the round box.



3. Each *variable* is translated into a fresh variable name (i.e. a new variable name).

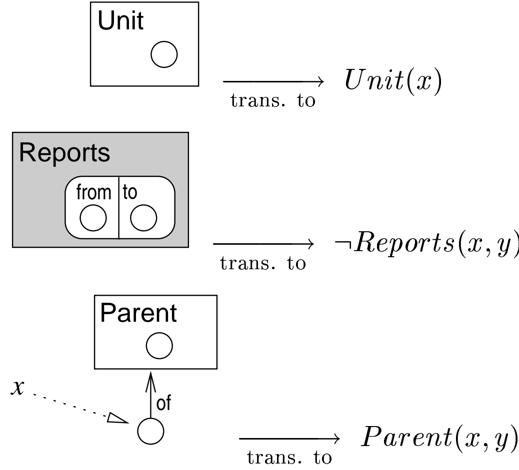


4. *Aggregation functions* are translated into their textual equivalent expression.

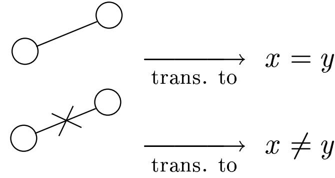


5. *Visual Literals*: Each *tuple* included in a *box* is translated into a textual literal such that:

   - Its predicate symbol is the symbol indicated inside the box corresponding to the visual literal being translated.
   - The arity of the predicate symbol is the arity of the tuple included in the box plus the number of incoming arrows. When a variable or a constant is included in the box corresponding to the visual literal then the predicate symbol has arity one plus the number of incoming arrows. (i.e. for translation purposes constants and variables are seen as singleton tuples).
   - The literal will be a negated atom iff the box is shadowed.
   - The terms of the predicate are the textual translations of the visual terms of the tuple and those of the incoming arrows, as obtained in steps num. 2 and num. 3.

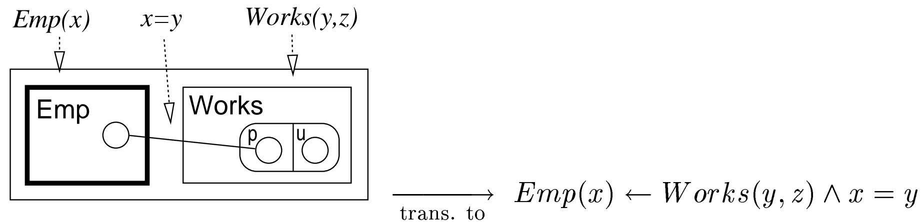


6. *Visual evaluable literals*: each visual evaluable literal is translated into an equivalent textual literal. For example, variable equality/inequality evaluable literals are translated into textual evaluable literals of the form '$x = y$' or '$x \neq y$' where 'x' and 'y' are the textual translations of the two variables as obtained in step num. 2.



7. *Visual deductive rules* (diagrams): each visual deductive rule is translated into an equivalent textual deductive rule such that:

   - The conclusion of the rule is the translation corresponding to the conclusion box (drawn with thick lines) as obtained in step num. 5.
   - The conditions of the rule are the translations of all condition boxes (as obtained in step num. 5) plus the translations of visual evaluable literals (as obtained in step num. 6).



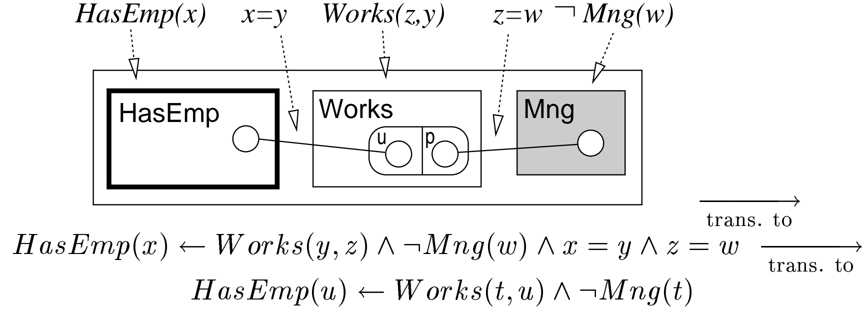
   Notice that visual integrity constraints rules are treated as normal visual deductive rules defining an inconsistency predicate Ic_n.

8. (optional) *Textual variable equality literals* can be eliminated by substituting along the new deductive rule the two variable names appearing in a literal '$x = y$' by a unique new name.

$$Emp(x) \leftarrow Works(y, z) \wedge x = y \xrightarrow[\text{trans. to}]{} Emp(w) \leftarrow Works(w, z)$$

4.2 Example Visual-to-Textual Translations

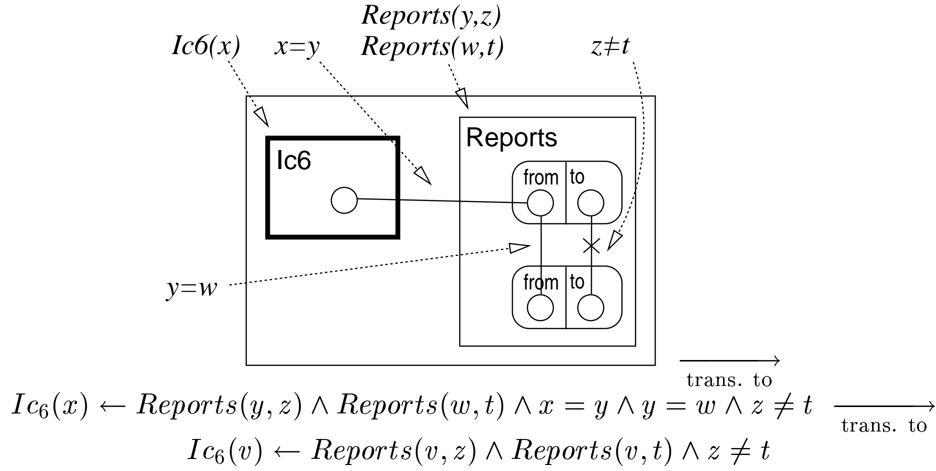
First we translate a deductive rule: the one defining the *HasEmp* predicate as the units that have some non-manager employee.



$$HasEmp(x) \leftarrow Works(y,z) \wedge \neg Mng(w) \wedge x = y \wedge z = w \;\xrightarrow[\text{trans. to}]{}$$

$$HasEmp(u) \leftarrow Works(t,u) \wedge \neg Mng(t)$$

And now we translate an integrity constraint rule: a unit cannot report to two different units.

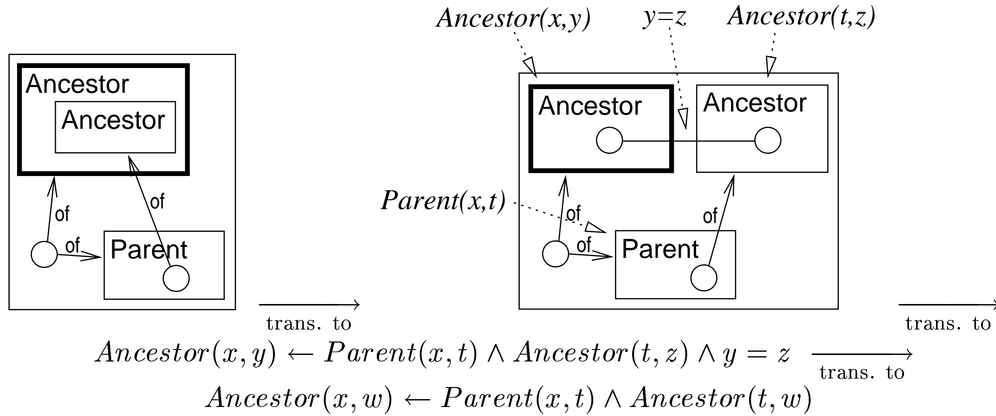


$$Ic_6(x) \leftarrow Reports(y,z) \wedge Reports(w,t) \wedge x = y \wedge y = w \wedge z \neq t \;\xrightarrow[\text{trans. to}]{}$$

$$Ic_6(v) \leftarrow Reports(v,z) \wedge Reports(v,t) \wedge z \neq t$$

Notice that in this example, the *Reports* box has two tuples included in it and therefore two different textual literals are produced as translation.

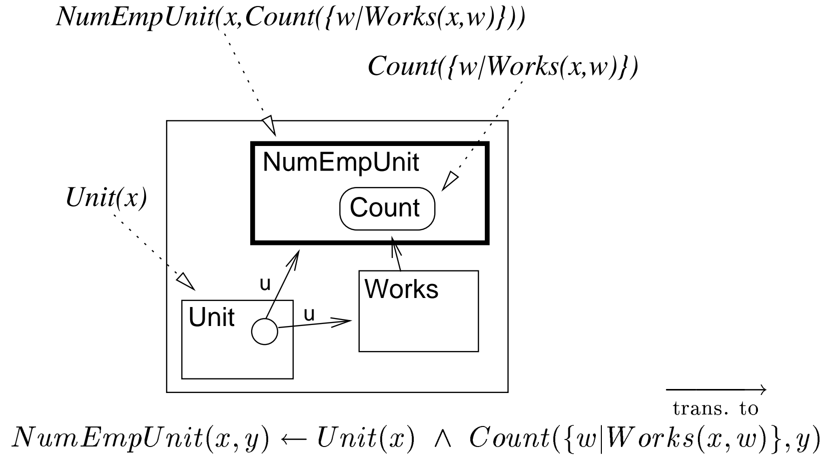
Now we translate the recursive case of the ancestor definition as defined in Section 3.4. Since there is a box inclusion step 1 above is performed to eliminate this box inclusion.



$$Ancestor(x,y) \leftarrow Parent(x,t) \wedge Ancestor(t,z) \wedge y = z \;\xrightarrow[\text{trans. to}]{}$$

$$Ancestor(x,w) \leftarrow Parent(x,t) \wedge Ancestor(t,w)$$

Finally we translate the example where an aggregation function is used:



$$NumEmpUnit(x, y) \leftarrow Unit(x) \ \wedge \ Count(\{w|Works(x, w)\}, y)$$

5 Our Visual Environment Prototype

We outline here the *environment prototype* in which the concepts explained have been experimented, and which could be added as a front-end to an existing deductive database system. In Figure 6 we sketch the structure of this system. The idea is simple: the visual module is inserted in between the database designer interface and the existing deductive database system. Thus the database designer works with the visual tool, writing visual schemas, while the input to the deductive database system is conveniently translated to a conventional textual language as the one defined in Section 2.

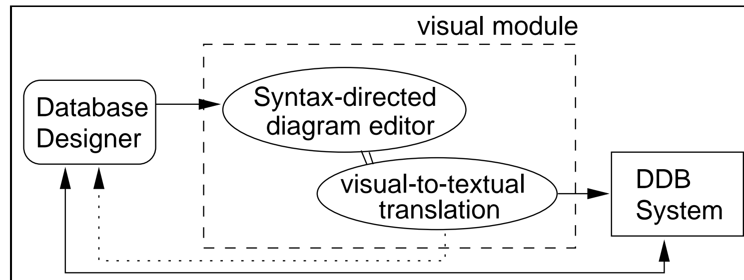


Figure 6: Schema of the Prototype

The *visual module* contains two parts: a *syntax-directed editor* and a *visual-to-textual translator*. The syntax-directed editor is a diagram editor that produces a syntactic graph of the diagram that is fed into the translator. The translator is just an implementation of the algorithm explained in section 4 which takes as input that syntactic graph and produces a textual equivalent suitable to the deductive database system. Let us now focus on the syntax-directed editor.

5.1 Syntax-Directed Diagram Editing

The syntax-directed editor provides the user with high-level operations to draw diagrams that conform to the syntactic rules of our given language. But why do we propose a syntax-directed editor? While editing and parsing in textual languages is a well studied field, visual languages present new difficulties. On the one hand, general purpose graphical editors are more difficult,

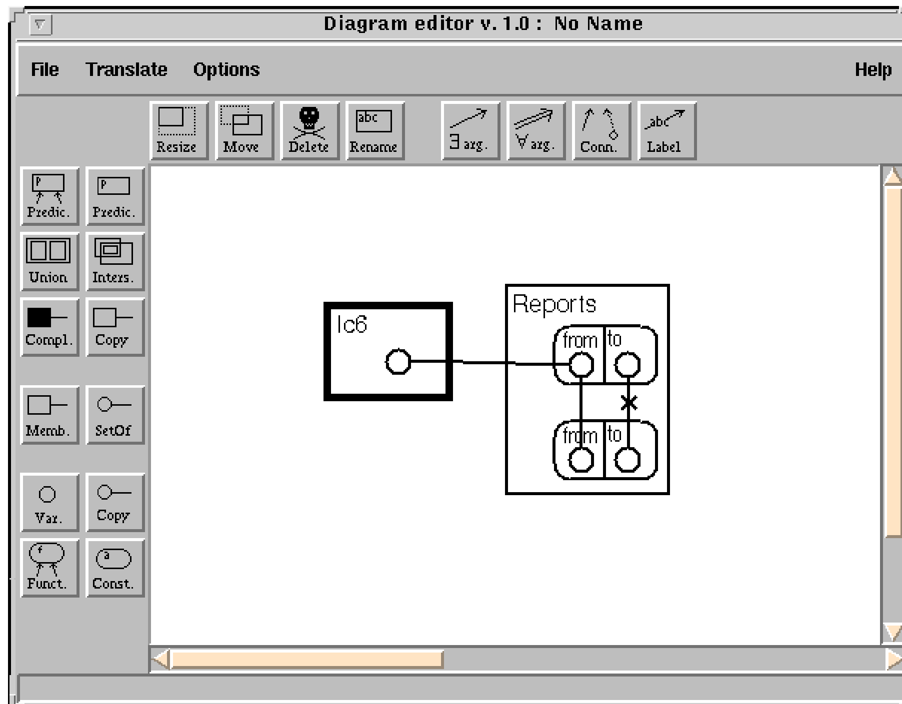


Figure 7: Snapshot of the Diagram Editor

and sometimes tedious to use. On the other hand, parsing techniques for visual languages are still much underdeveloped. Furthermore it seems that the intrinsic complexity of visual languages is also higher than that of textual languages, which may complicate parsing. Our goal using syntax-directed editing techniques is to make the environment simpler, eliminating the need for a parser, and facilitating the edition process. The editor that we have implemented produces directly a syntactic graph of the diagram in a form that it can be directly translated to predicate-logic based language.

In Fig. 7 we give a snapshot of the syntax-directed editor screen we are using. The editor is been implemented using Tcl/Tk and Sicstus Prolog, and we are currently implementing a new improved version using Java. It was originally implemented having in mind a visual logic programming language as described in [1], and was designed to be easily customizable to edit other similar languages, like the one described in this paper. We do not want to enter into much detail about the syntax-directed editor. The main goal of this Section is to show that we have skipped one of the main problems of implementing a visual language, the construction of a parser, by using a syntax-directed editor. Since our language has a limited number of constructs this editor is not difficult to construct, and apart from obviating the need of a parser it helps the user to follow the syntactic rules of the language during diagram editing.

6 Related Work

Previous work of our group has focused on the area of visual declarative programming. We designed a visual logic programming [23, 1] and more recently we have devised a visual operational semantics

for this language [24, 22], the visual inference conducting to query solutions. The starting point of the work of our research group in the visual languages area was diagrammatic reasoning as presented by Jon Barwise's group (see [5, 2]). However our goals differ from those of the diagrammatic reasoning community. Our use of Venn/Euler diagrams is centered on their computational aspects. We want to focus on simple diagrams with a clear computational interpretation, avoiding as many logical symbols as possible. There exist other visual declarative programming languages like CUBE [20, 21], VEX [10] and SPARCL [28], but none of them uses sets, Venn/Euler diagrams and graphical inclusion as its foundations. In the Database field, most visual formalisms are applied to what is known as visual query languages and we do not know of any other formal visual language devoted to the task of deductive database schemas representation. Visual Query languages are languages designed to be similar to the contents of the database. An interesting example is NQS [16], a graphical system for binary models.

The graphical schemes representing conceptual models in [6] do not attempt a formal and systematic visual representation of deductive rules. However they are an inspiration for our future work. The *existential graphs* of Charles S. Peirce (see [25, 13]), a full First-Order-Predicate-Logic diagrammatic reasoning system, are of great interest and a source of inspiration of our research; together with John Sowa's *conceptual graphs* (see [26, 27]) modeled after Peirce's diagrammatic approaches to predicate logic. As we already pointed out in the Introduction, our approach differs from that of conceptual graphs mainly on the type of visual representations used.

7 Conclusions and Further Work

The visual language we have presented here is a complementary alternative (although not empirically tested) to conventional textual languages. While being *completely formal*, we believe that it allows to visually define deductive database schemas in a *simple, concise and elegant way*. The graphical metaphor of representing predicates as sets of tuples —the key-point of this work— is, we claim, easy to understand. Venn/Euler diagrams are one of the most used visual notations in mathematics, specially at elementary school when learning the basic facts about set theory. Moreover, as we have pointed out before, using a visual syntax allows us not to give names to variables, using instead a graph-like mechanism to express variable relationships, simplifying the comprehension of the rule.

Using this visual language users are able to introduce much more secondary notation in their descriptions, than using a textual language. *Secondary notation* is the set of syntactic properties and items of a language that, while in its formal description are not taken into account they are very useful to facilitate its understanding. For instance, indentation in C++ or box distribution in our visual language are elements of secondary notation. It is known that small simple rules are commonly preferred over long and complicated ones. We believe that using our visual notation one tends to write more compact and smaller deductive rules: a diagram with too many boxes is not visually appealing while in a textual rule it is not so clear when to stop adding literals.

Recent studies stress the fact that, well used, visual languages might give better empirical results than textual ones. For instance in [8] an empirical study shows that the performance of a visual query language based on the well known Entity-Relationship model is better than SQL. However, our implementation of the language is still on its early stages and we have performed only limited tests with subjects outside the development group. A *full empirical study* of the language is planned once a full Java version of the environment is available.

Up to now we have concentrated our efforts on studying the basic problems of deductive database schema visual definition. However, we have not addressed the question of how to visually define

transition integrity constraints, nor *updates*. Both are goals to be pursued in the future. Our work in visual logic programming languages, and specially that of [24], gives us the bases to think that it is possible to visually define *queries* and to visualize their *answers* using our visual language. We have found that our visual syntax allows us to represent query answers showing different alternative solutions in a single diagram together with their trace (i.e. how the solutions have been obtained).

We also want to study how different *properties of deductive database schemas* (like being *allowed* or *stratified*) can be represented or even defined in a visual way. A drawback of deductive databases presented by means of a set of deductive rules is its flat character, the lack of explicit connections between the same predicates in different rules. This has been addressed by representation of dependency graphs in some environments (i.e. LPA MacProlog). We are considering similar ways of reflecting the global structure of the intensional component of the database. It is also interesting to study how to combine our formalism with other existing ones, like for instance the Entity-Relationship model for base predicates. An ongoing research project deals with the design of a heterogeneous environment that allows to combine our visual language with other visual formalisms or textual languages. Another important work, complementary to the one explained in Section 4, is to study how to visualize existing textual schemas. Actually, the translation algorithm has already been devised. The main challenge is to draw the diagram, i.e. to provide the layout of the diagram. Much work has been done designing graph layout algorithms, but it is still an interesting open problem in the general case.

Acknowledgments

Part of the work has been done while the first author was visiting the Visual Inference Laboratory of the Indiana University supported by a doctoral grant of the *Direcció General de Recerca* (*Generalitat de Catalunya*). The first and third authors have been partially supported by the MODELOGOS project TIC97-0579-C02-01, and the second author has been partially supported by the PRONTIC program project TIC97-1157.

References

[1] Jaume Agustí, Jordi Puigsegur, and Dave Robertson. A Visual Syntax for Logic and Logic Programming. *Journal of Visual Languages and Computing*, 1997. To appear.

[2] Gerard Alwein and Jon Barwise, editors. *Logical Reasoning with Diagrams*. Oxford University Press, New York, 1996.

[3] K.R. Apt, H.A. Blair, and A. Walker. Towards a theory of declarative knowledge. In J. Minker, editor, *Foundations of deductive databases and logic programming*, pages 89–148. Morgan Kaufmann Publ., 1988.

[4] F. Bancilhon and R. Ramakrishnan. An amateur's introduction to recursive query processing strategies. In *Proc. of ACM Int. Conf. on Management of Data*, pages 16–52, Washington D.C., 1986.

[5] Jon Barwise and John Etchemendy. *Hyperprooff*. CSLI Publications, Stanford, 1993.

[6] M. Borman, J.A. Bubenko, P. Johannensson, and B. Wangler. *Conceptual Modelling*. Prentice Hall, 1997.

[7] J. A. Bubenko. Extending the scope of information modelling. In *Proc. Fourth Int. Workshop on the Deductive Approach to Information Systems and Databases*, pages 73–98, Lloret, Catalonia, 1993.

[8] T. Catarci and G. Santucci. Diagrammatic vs Textual Query Languages: A Comparative Experiment. In Stefano Spaccapietra and Ramesh Jain, editors, *Proc. of the 3rd IFIP 2.6 Working Conference on Visual Database Systems*, 1995.

[9] U.S. Chakravarthy, J. Grant, and J. Minker. Foundations of semantic query optimization for deductive databases. In J. Minker, editor, *Foundations of deductive databases and logic programming*, pages 243–273. Morgan Kaufmann Publ., 1988.

[10] Wayne Citrin, Richard Hall, and Benjamin Zorn. Programming with Visual Expressions. In *Proceedings of the 11th IEEE Symposium on Visual Languages*, Darmstadt, Germany, September 1995. IEEE Computer Society Press.

[11] Hendrik Decker. The range form of databases or: How to avoid floundering. In *Proc. of 5th. GAI*, Innsbruk, 1989.

[12] H. Gallaire, J. Minker, and J.M. Nicolas. Logic and databases: A deductive approach. *ACM Computing Surveys*, 16(2):153–185, 1984.

[13] Eric Hammer. *Logic and Visual Information.* Studies in Logic, Language and Computation. CSLI and FoLLI, Stanford, CA, 1995.

[14] IBM Business Systems Development Method Technical Description. Technical Report Tech. Reps. GE19-5387, SC19-53(09,10,12,13), International Business Machines Corporation, 1992.

[15] M. Jarke, R. Gallersdorfer, M.A. Jeusfeld, M. Staudt, and S. Eherer. Conceptbase – a deductive object base for meta data management. *Journal of Intelligent Information Systems, special issue on Advances in Deductive Object-Oriented Databases*, 4(2):167–192, 1995.

[16] H. J. Klein and D. Krämer. NQS – A Graphical Query System for Data Models with Binary Relationship Types. In Stefano Spaccapietra and Ramesh Jain, editors, *Proc. of the 3rd IFIP 2.6 Working Conference on Visual Database*, 1995.

[17] J.W. Lloyd. *Foundations on Logic Programming, 2nd. ed.* Springer, 1987.

[18] J.W. Lloyd and R.W. Topor. Making prolog more expressive. *Journal of Logic Programming*, (3):225–240, 1984.

[19] J. Minker. Logic and databases: A 20 year retrospective. In *Proc. of Int. Workshop on Logic in Databases.* San Miniato, Pisa, 1996.

[20] Mark Alexander Najork. *Programming in Three Dimensions.* PhD thesis, University of Illinois at Urbana-Champaign, Urbana, Illinois, 1994.

[21] Mark Alexander Najork. Programming in Three Dimensions. *Journal of Visual Languages and Computing*, 7:219–242, 1996.

[22] J. Puigsegur and J. Agustí. Visual Logic Programming by means of Diagram Transformations. In *Proc. of APPIA-GULP-PRODE Joint Conference in Declarative Programming*, La Coruña, Spain, July 1998.

[23] Jordi Puigsegur, Jaume Agustí, and Dave Robertson. A Visual Logic Programming Language. In *Proc. of the 12th IEEE Symposium on Visual Languages*, Boulder, Colorado, September 1996.

[24] Jordi Puigsegur, W. Marco Schorlemmer, and Jaume Agustí. From Queries to Answers in Visual Logic Programming. In *Proc. of the 13th IEEE Symposium on Visual Languages*, Capri, Italy, September 1997.

[25] Don D. Roberts. *The Existential Graphs of Charles S. Peirce.* Mouton and co., The Hague, 1973.

[26] John F. Sowa. *Conceptual Structures. Information Processing in Mind and Machine.* Addison Wesley, 1984.

[27] John F. Sowa. Relating Diagrams to Logic. In Guy W. Mineau, Bernard Moulin, and John F. Sowa, editors, *Conceptual Graphs for Knowledge Representation, Proc. of the First Int. Conf. on Conceptual Structures, ICCS'93, Quebec City, Canada*, Lecture Notes in Artificial Intelligence (699). Springer Verlag, Berlin, 1993.

[28] Lindsey Spratt and Allen Ambler. A Visual Logic Programming languages based on Sets and Partitioning constraints. In *Proceedings of the 9th IEEE Symposium on Visual Languages*, Bergen, Norway, September 1993. IEEE Computer Society Press.

[29] Special issue on prototypes of deductive database systems. *Journal of Very Large Databases*, 3(2), 1994.

[30] R.J. Wieringa. *Requirements Engineering: Frameworks for Understanding.* Willey Publ., 1995.

[31] J.J.V.R. Wintraecken. *The NIAM Information Analysis Method: Theory and Practice.* Kluwer, Deventer, The Netherlands, 1990.

Papers [1, 22, 23, 24] are available at the following URL:
`http://www.iiia.csic.es/~jpf/publications.html`