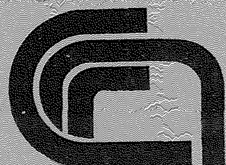

CONSIGLIO NAZIONALE
DELLE RICERCHE

IST. EL. INF.
BIBLIOTECA
Posiz. ARCHIVIO BG-69

P. Asirelli*, D. Di Grande*,
P.Inverardi*, F.Nicodemi**

# Graphics by a Logic Data Base Management System

N. R/6/16 BG-69

Luglio 1990

PROGETTO FINALIZZATO
SISTEMI INFORMATICI E CALCOLO PARALLELO

SOTTOPROGETTO 6

METODI E STRUMENTI PER LA PROGETTAZIONE DI SISTEMI

PROGETTO FINALIZZATO
SISTEMI INFORMATICI E CALCOLO PARALLELO

SOTTOPROGETTO 6
Metodi e Strumenti per la Progettazione di Sistemi

Coordinatore Bruno Fadini

P. Asirelli*, D. Di Grande*,
P.Inverardi*, F.Nicodemi**

# Graphics by a Logic Data Base Management System

N. R/6/16 RG-69 Luglio 1990 Dic.

Relazione di Ricerca

*I.E.I - C.N.R.
Via S. Maria,46-56100 Pisa

**Olivetti D.O.R.
Ing.C. Olivetti &C. S.p.A.
Lungarno Galilei-56100 Pisa

## Sommario

In questo lavoro descriviamo un modello dei dati, ed il corrispondente linguaggio di definizione, progettato e sviluppato per la manipolazione di oggetti grafici in un contesto di base di dati logiche.

La caratteristica principale del modello e' costituita da un insieme di concetti nuovi che cercano di eliminare i difetti presenti nei modelli dei sistemi grafici piu' diffusi. Il modello e' stato integrato con un Sistema di Gestione di Basi di Dati logiche, EDBLOG, al fine di ottenere GRAPHEDBLOG, che e' quindi un sistema per la gestione di basi di dati logiche in cui si possono manipolare, in modo uniforme (dichiarativo), informazioni sia grafiche che non grafiche.

**parole chiavi:** Grafica, modello dei dati, programmazione logica, sistema di gestione di basi di dati logiche, prototipazione rapida.

## Abstract

We describe a data model, and the corresponding data definition language, which have been designed and developed for the management of graphical objects in a logic data base environment.

The principle feature of the model is a set of new concepts that try to eliminate the defects of the most commonly used graphical system models. The model has been integrated in an existing Logic Data Base Management System, EDBLOG, to obtain GRAPHEDBLOG, an LDBMS in which graphical and non graphical information is handled in a uniform (declarative) way.

**key-words:** Graphics, data model, logic programming, logic database management system, rapid prototyping

# 1 Introduction

In recent years, low cost workstations characterized by high speed computing power and large memory capacity, have become easily accessible to all kind of users. Thus, the need for graphics in various application areas has increased.

Generally, graphical systems are characterized by three main components [6]:

- the *data model*;
- the *graphical component*;
- the *interface* that connects the data model with the graphical routines.

So far, considerable effort has been spent on the standardization of the graphical component in order to guarantee a uniform interface to graphical functionalities, and to render them neither host dependent nor application dependent.

The standardization of the graphical component permits applications to become hardware independent but existing data models still lack in the following aspects:

- descriptions of graphical objects are extremely unnatural and affected by the algorithmic programming style used. Thus, efforts have been devoted to making graphics more powerful, on one side by introducing object-oriented characteristics [11] [19] and, on the other, by using logic languages to define graphical objects and their operations [12] [ 8] [9].
- the data model is often application dependent. Thus, the use of a data base management system to represent and manipulate data in graphical applications is investigated [17] [1] [2].

In the following, we present an approach to graphics which is logic database oriented. We introduce a (data) model, and the corresponding linguistic constructs to manage graphical objects in a logic data base management environment.

The main feature of the model is a set of new concepts that try to eliminate the defects previously outlined. The model was then defined in terms of an existing Logic Data Base Management System, EDBLOG, thus obtaining GRAPHEDBLOG, an LDBMS in which graphical and non graphical information is handled in a uniform way. Furthermore, because of the declarative style, it is possible to validate applications, e.g. system specifications against user requirements, etc., thus making GRAPHEDBLOG a system to *define* and to *prototype* applications.

In section 2, a brief description of the kernel LDBMS is introduced in order to presents the framework within which the data model is defined. In section 3, the model is presented. Section 4 gives the linguistic constructs related to graphical objects. Section 5 presents all the definitions in EDBLOG that implement the model. An application example is sketched in Section 6 while in the Appendix the whole set of rules and integrity constraints are presented by a sequence of interactions with the system.

## 2. The kernel Logic Database Management System

In the following, we describe the logic database management system EDBLOG [2] which is the kernel LDBMS. We assume some familiarity with Logic Programming terminology, and for further details we refer to [10].

The EDBLOG system consists of four parts:

1) a logic program in which:

- *the set of facts*, "unit" definite clauses, are considered to be the Extensional component of the DB (EDB);
- *the set of deductive rules*, definite rules, are considered to be the Intensional component of the DB (IDB);

2) a set of integrity constraint formulas of two kinds:

- *a set of Integrity Constraints (IC)*, which are formulas of the form:

$$A_k \rightarrow B_1, \ldots, B_s \quad \text{with } s \geq 0.$$

- *a set of Control formulas* which are either IC formulas or else

$$A_1, \ldots, A_m \rightarrow B_1, \ldots, B_n \quad \text{with } m,\ n \geq 0$$

where the logical connective $\rightarrow$ has the usual semantics and "," stands for the logical connective $\wedge$.

3) Given a database (a logic program), a set of clauses that define compound updating operations (*transactions*) can be defined with the following syntax:

i) $trans_i \leftarrow$ **prec** | $t_1, \ldots, t_n$ | **post**

The procedural interpretation is that, to execute the operation *trans*, the precondition (**prec**) must be verified in the current DB, the clause containing this precondition must then be committed, the body executed and the corresponding postcondition verified in the modified database. The commit operation is a way of expressing the behaviour of the Prolog cut operator, a failure in the body of a transaction causes the failure of the transaction.

ii) $trans_i \leftarrow$ **prec** # $t_1, \ldots, t_n$ # **post**

The procedural interpretation of this kind of transaction is that to execute the operation *trans*, the precondition (**prec**) must be verified in the current DB; the clause containing this precondition is then selected, its body executed and the corresponding postcondition verified in the modified databases. A failure in the body or in the postcondition does not cause the failure of the transaction, the computation proceeds by searching for another suitable definition of *trans*. The transaction fails if all its defining clauses fail.

The $t_i$ in the bodies can only be user defined or system predefined transactions.

Preconditions / postconditions in the definitions of transactions denote particular forms of *Controls* which must be checked before/after the execution of a set of operations (body of the transaction). They are introduced to separate global DB controls (*Controls*) from those related to particular transactions, thus reducing the number of necessary global *Control* formulas.

Transaction definitions are searched according to the standard Prolog strategy, where clauses are tried in the order they appear in the program. Thus, the *commitment* will be to the first clause whose precondition part succeeds. The successful evaluation of a transaction causes the *Control* formulas to be checked. The required transaction operation is aborted if the *Control* checking fails. Abortion also occurs upon failure of postconditions or of certain operations of the body. The abortion of a transaction is handled by maintaining a *transition log* which is also useful when the system crashes to restore the previous state.

Transactions can be defined recursively, and two additional control constructs are defined on them: a *for_all(Cond, Trans)* construct and an *iterative(Trans)* construct; their semantics is rather straightforward, the former collects all solutions to *Cond* and executes *Trans* for each of them. Example: for_all((age(Name, X), X>18), update(Name)); the latter collects the set of solutions to the precondition for every clause defining the transaction and executes the body for each of these values. Ex: iterative(update(Name)).

A set of elementary updating operations is provided by the system as a meta-theory with respect to the DB. Such operations also allow *IC*, *Control* formulas and transactions to be added and deleted.

Figure 1 shows a simple example of the kind of database that can be handled by EDBLOG.

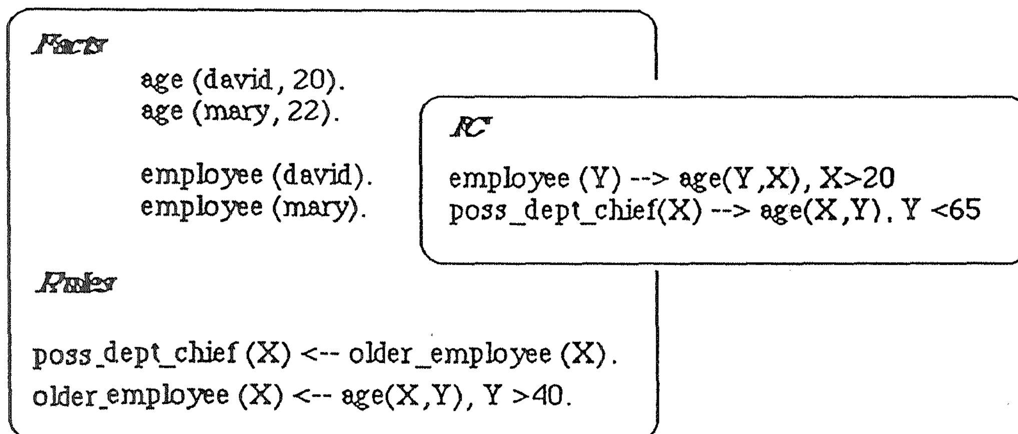


**Figure 1**

## 3 The graphical model

Our model is here introduced by giving a set of definitions together with some explanations.

The model has the following features:

- it is *declarative* and *deductive* i.e. the descriptions of the graphical elements are not given procedurally, but consist in a set of *clausal* definitions;
- it is *compositional*, i.e. complex elements can be described in terms of previously

defined elements;

- it is *fast prototyping oriented*;

The model consists in the following elements:

- *prototypes*, i.e. templates of graphical objects;
- *mechanisms to compose* prototypes;
- *graphical objects*, i.e. instances of prototypes;
- *mechanisms to operate* on graphical objects;
- *a frame*, i.e. the abstract plane in which graphical objects are drawn.

## 3.1 Prototypes

The prototype concept naturally emerges when different views of the same object are considered useful and, furthermore, when it is desired to use a given graphical object as a sub-object in several more complex objects. Thus, the notion of prototype is similar to the notion of (*generic*) type found in high level programming languages such as Ada™; that is, its definition does not define a new object, but is a template that will be used to *create* graphical objects upon instantiation.

**Definition 1**: Attributes are features of a graphical object. Two classes of attributes are provided:

- *contextual* attributes;
- *absolute* attributes. ♦

All prototype descriptions are parametric with respect to contextual attributes and, optionally, also to certain absolute attributes.

**Definition 2**: A *prototype* is the description of a graphical object which is parametric with respect to contextual *attributes*. ♦

**Definition 3**: A *parametric prototype* is a prototype which is parametric with respect to absolute attributes. ♦

**Definition 4**: The *interface* of a prototype definition consists in the set of its parametric components (attributes). ♦

Prototypes are divided into two further classes:

- *basic primitive prototypes*, that represent the usual graphical output primitives (e.g. point, polygon,..). They are system defined and their description cannot be manipulated by the user;
- *user defined prototypes*, which can be:
  - *compound user defined prototypes*; defined as the composition of user defined prototypes and basic primitive prototypes.
  - *primitive user defined prototypes*; defined only in terms of basic primitive prototypes and obtained from user defined prototypes by means of a *compilation* process.

Given that the model is compositional (i.e. a prototype may be defined in terms of other prototypes), the compilation of a compound user defined prototype into a primitive user

defined one eliminates its dependencies on its component prototypes. This means that a primitive user defined prototype only consists of basic primitive prototypes. This class of prototypes has been introduced because it has at least two advantages:

- *encapsulation*: modifications to any component will not affect the compound prototype;
- *efficiency*: the visualization of an instance of a primitive prototype will be more efficient than the visualization of an instance of the same non-primitive prototype.

**Definition 5:** *Contextual attributes* are of two kinds:

*i) geometric* attributes:

- *origin*
- *scale*
- *rotation*

where, origin fixes a point on the plane so that the coordinates of each instance of a prototype have to be computed with respect to this point. The scale and rotation give the scale and the rotation with respect to which instances are created.

*ii) state* attributes:

- *raster function (mode)*, represents a logic connective (and, or,...) to be applied to the source destination pixels (those present on the visualization plane) to obtain the new destination pixels;
- *fill pattern*, represents the pattern that will be used to fill the surface of the given object;
- *write pattern*, represents the pattern that will be used to draw the border of the given graphical object;
- *write style*, represents the style that will be used to draw the border of the given graphical object;
- *write width,* represents the width that will be used to draw the border of the given graphical object;
- *color table*, represents the name of a color table to be used at visualization time. ♦

**Definition 6:** *Absolute attributes* are user definable components. ♦

Absolute components are useful when a prototype is to denote the partial description of a graphical object in which some components, apart from the contextual ones, are left unspecified. For example, it is possible to define prototypes such as the *arch(Length, Height)*, that describe an arch parametrically, with respect to its length and height.

As we have previously pointed out, the description of a prototype defines the graphical structure of the object to be represented parametrically, with respect to some attributes. Sometimes it is useful to limit the range over which these attributes may assume values; in other words, it may be useful, or even necessary, to *type* such attributes. Thus, we introduce the concept of *property*.

**Definition 7**: A *property* is a pair <*name*, *value*>, where *name* denotes an attribute and *value* denotes its value. Each property is associated to the definition of a prototype. ♦

**Definition 8**: Given a prototype P, parametric with respect to an attribute A, the set of properties for A, determines the range of values $V_{P_A}$ for A wrt P. ♦

The set of values $V_{P_A}$ enumerates possible options, and the attribute A of each instance of the prototype P, must have a value in $V_{P_A}$. For example, if we assume that, in the database, the description of prototype P has the following properties for its attribute *origin*: <origin, [10,10]>, <origin, [20,20]>, then all instances of P must have their origin at coordinates [10,10] or [20,20].

### 3.1.1 Compound prototypes

When dealing with compound prototypes, if a graphical object $O_1$ is a structural element of a more complex object $O_2$ then the description of the prototype $P_2$ of $O_2$, must declare the use of the prototype $P_1$ of $O_1$.

**Definition 9** A *use declaration* of a prototype $P_2$ within a prototype $P_1$ specifies the information that is needed to obtain the values of the attributes of $P_1$ from the actual values of the attributes of $P_2$. ♦

The information carried by a use declaration is used to obtain instances of $P_2$ from instances of $P_1$. In particular, the values of the geometric attributes, are considered in relation to the values of the corresponding attributes of the defining prototype, while the values of the other kinds of attributes, when mentioned, are considered to be absolute. The information carried by a use declaration is described below, when discussing the representation of prototypes.

The description of a user defined prototype thus consists of a set of *use declarations* of other prototypes. This permits a prototype to be modelled as the composition of several sub-prototypes.

**Definition 10**: A prototype $P_1$ *depends* on a prototype $P_2$ if the description of $P_1$ contains a use declaration of $P_2$ or, if it contains a use declaration of a prototype $P_3$ that depends on $P_2$. ♦

Given a prototype P, its dependencies can be represented as a graph, the *dependency graph*, where the root denotes P and the other nodes denote all the prototypes on which P depends. An oriented arc, from a node $N_1$ to a node $N_2$, shows that, in the description of the prototype denoted by $N_1$, there is a use declaration of the prototype denoted by $N_2$.

**Definition 11**: A prototype is *definable* if its dependency graph is acyclic. ♦

With this definition, we want to make clear that the only prototypes that can be described in our model are hierarchical, ones.

*Conditional use declaration*

The model we are describing is based on the assumption that descriptions of graphical objects will be stored in a database together with non-graphical information.

In such a situation cases will arise in which the use of a prototype $P_1$, within a prototype $P_2$, is subordinate to the existence of a certain item of information in the database. In a high level programming language framework, this corresponds to the concept of *record with variant*. In practice, taking advantage of our mixed environment we want to be able to model cases in which some information about our data might not be available at definition time, whereas it will be available at query time (retrieval and/or visualization time). Hence, the structure of a graphical object is no longer static and fixed at definition time. For this purpose we introduce the concept of *conditional inclusion*.

**Definition 12**: A *guard* is either a query to the database or a boolean expression. ♦

**Definition 13**: A prototype $P_2$ *conditionally includes* a prototype $P_1$, if the use declaration of $P_1$ contains a guard. ♦

Conditional inclusion allows the definition of graphical objects that could have a different structure at visualization time, depending on the state of the database.

*Representing prototypes as trees*

A prototype P that consists of (uses) N sub-prototypes, can be structurally represented by a tree where the root denotes the prototype itself and the nodes directly connected to the root (depth level 1) denote the N sub-prototypes used by P. Every arc in the tree represents a use declaration (in the prototype associated to the leading node) of the prototype associated to the ending node. The leaf nodes of the tree denote primitive prototypes (basic or user defined).

Every arc in the tree carries a set of information:

- a guard: a condition that has to be evaluated as *true* (when the arc is traversed), for the use declaration to be effective; if no condition is required, the guard is set to *true;*
- the relations that bind the actual attributes of the *using* prototype to the attributes of the *used* prototypes;
- a priority factor that determines the ordering in which the sub-tree has to be visited.

Figure 3 gives an example of a tree representation for a prototype.

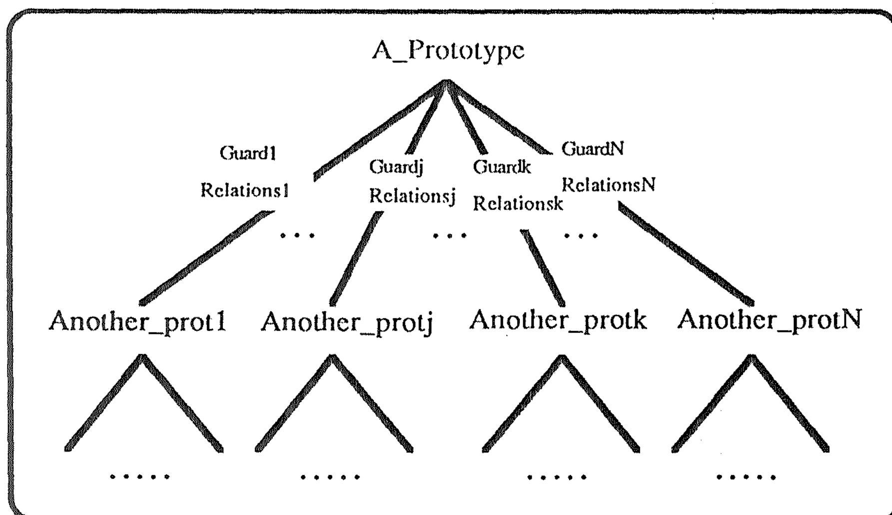


*Figure 3*

*Correctness of non conditional prototypes*

Once the semantics to be associated to the attributes has been chosen, we have to decide when a set of attributes can be asserted on a prototype and how these attributes will affect the use of the prototype within other prototypes.

For this purpose, we introduce a set of examples which will help in understanding the forthcoming definitions of *correctness* in relation to the use that a prototype makes of other prototypes. Such notion of correctness has been defined for non conditional prototype only. In fact, it only makes sense if it is possible to perform a static checking. This cannot, in general, be done when using conditional prototypes where, it may happen that, some information about the use of a prototype can be available only at run time.

Let us consider a user defined prototype P, parametric with respect to parameter $x_1,...x_n$, whose corresponding attributes are $A_1,...A_n$ . Let us also suppose that the description of P contains the use declaration of s prototypes $P_1,...,P_s$. If $P_k$ is one of these prototypes, then in the tree representation of P, there is an arc connecting the node corresponding to P to the node labelled $P_k$. That is, we have:

$P(x_1,...,x_n)$
|
|
$P_k(y_1,...,y_m)$

Such an arc is labelled with m functions $f_{y_1},...,f_{y_m}$, each of which is defined as follows:

$$f_{y_j}(x_{1y_j},..., x_{sy_j}, d_{1y_j}, ..., d_{ry_j}) = y_j \quad \text{with } j\leq m \text{ and } s_{y_j} \leq n \text{ and } r_{y_j} \geq 0$$

i.e., given the set of actual values for the parameters of P and, possibly, a further set of values $(d_{1y_j}, ..., d_{ry_j})$, called $\delta$, each of the $f_{y_j}$ functions computes the actual value of the

corresponding parameter for $P_k$.

For example, if, in the description of *prot*, there is a use declaration of the prototype *another_prot* with parameter values *(15,17)* for *origin_coord* and *2* for *scale*, then the functions associated to the arc connecting the two nodes will include two functions of the following kind:

do_real_origin_coord (P,S,R, (15,17))

do_real_scale (S,2),

The first function, given the position of the origin_coord P, the scale S and the rotation R, that have to be applied to an instance of *prot*, and given the position of the origin_coord of the instance of *another_prot* (15,17) with respect to the instances of *prot*, computes the actual position of the instance of *another_prot*.

The second function, given the actual scale of an instance of *prot* S, and the scale (2) of the instance of *another_prot* with respect to S, computes its actual scale.

Let us now consider one of the functions associated to the arc above in relation to the $y_j$ parameter:

$f_{y_j}(x_{1y_j}, \ldots, x_{s_{y_j}}, d_{1y_j}, \ldots, d_{ry_j})$ with $j \leq m$ and $s_{y_j} \leq n$ and $r_{y_j} \geq 0$

Let us assume that we have an adequate number of values for the $d_{iy_j}$ parameters in $\delta$, and that for all the s prototypes, each attribute has asserted values; that is, for each parameter of every prototype P, $P_1, \ldots, P_s$ the set of values for the corresponding attributes

$V_{PA_1}, \ldots, V_{PA_n}, V_{P_iA_{y_j}}$, $i \in [1, s]$, $j \in [1, m_i]$ exist.

All variables are typed, that is, the range of their values is fixed by the values of the attributes to which they refer. Thus, referring to $P_k$, each application of $f_{y_j}$ denotes a value that can be assigned to $y_j$, along a path which includes the arc connecting P to $P_k$.

**Definition :14** Let $T{\downarrow}PP_{k_{y_j}}$ be defined as follows:

$T{\downarrow}PP_{k_{y_j}} = \{ y \mid \forall x_1, \ldots, x_k \text{ with } x_i \in V_{PA_i} \;\; y = f_{y_j}(x_1, \ldots, x_k, d_1, \ldots, d_r) \}$ ♦

i.e. $T{\downarrow}PP_{k_{y_j}}$ denotes the set of values that the arc $PP_k$ assigns to the $y_j$ component of $P_k$.

We can define the following notion of correct use:

**Definition 15** A prototype P *correctly uses* a prototype $P_k$ if:

$\forall j \in [1 : m], V_{P_{kA_j}} \supseteq T{\downarrow}PP_{k_{y_j}}$ ♦

**Definition 16** A prototype P that uses s prototypes $P_1, \ldots, P_s$ is *correct with respect to attributes* if it correctly uses each prototype $P_i$, $i \in [1,s]$ where each $P_i$ is *correct with respect to attributes.* ♦

Let us now consider the general case where some, (or all) components of P are not typed, while all components of the used prototypes are.

In this case, in order to apply Definition 15, we give a set of Definitions that, roughly speaking, force the nontyped variables to assume the attributes of the *used* prototypes; that

is, we allow for a sort of bottom-up type inheritance mechanism.

Let us suppose, for the moment, that only one component of P is not typed. Let us also consider a used prototype $P_k \in \{P_1,...,P_s\}$, connected by an arc to P.

**Definition 17** Given a prototype $P(x_1,...,x_n)$, with an untyped component $x_p$, such that P uses a prototype $P_k(y_1,...,y_m)$, the set $T\uparrow PP_{Kx_p}$ that denotes the *$x_p$-inheritable attributes set with respect to $PP_k$*. is given by the solution of the following set of equations:

$f_1( x_{1_1},..., x_p,...,x_{1_i}, d_{1_1},...,d_{1_q})=y_1,$

...

$f_r( x_{r_1},..., x_p,...,x_{r_i}, d_{r_1},...,d_{r_q})=y_r$

Where $f_1,...,f_r$, are the functions associated with the arc $PP_k$, such that each one of them computes an actual parameter of $P_k$ and uses the untyped component of P, $x_p$, as parameter.

♦

Thus $T\uparrow PP_{Kx_p}$ denotes the set of potential values for $x_p$.

The intersection of all sets $T\uparrow PPj_{xp}$, $j \in [1,s]$, computed for each arc that leaves P will give the range of values that $x_p$ may assume in each function that uses it as a parameter.

**Definition 18** Given a prototype P using s (correct wrt attributes) prototypes $\{P_1,...,P_s\}$, for each untyped parameter $x_p$ of P, the *inherited attributes set*

$T\uparrow P_{xp} = \cap_{j\leq s} T\uparrow PPj_{xp}$

If no function $fP_{jr}$ uses $x_p$ as parameter, then $T\uparrow P_{xp}=U_{x_p}$, where $U_{x_p}$ denotes the universe of values. ♦

## 3.2 Graphical objects

A graphical object is obtained by fully instantiating a prototype, i.e. providing it with the information needed by its description, thus generating a **ground** (no variables are left uninstantiated) instance of the prototype definition.

The only operations that can be performed on graphical objects are:

*create*
*delete*
*retrieve*
*visualize*

Since the model will be implemented in a logic DBMS (EDBLOG), each of the above operations is defined in terms of operations on the DB that contain prototypes, instances, etc.. Thus, the *create* operation will have the effect of inserting in the DB a fact that, naming a prototype instance, asserts the existence of a new graphical object; *delete*, causes the deletion of a previously inserted object (fact); *retrieve*, is a query to the DB; *visualize* is defined in terms of the retrieve operation and depicts the retrieved object on the screen.

Like prototypes, graphical objects can be represented by trees. In this case, the root denotes the graphical object and there is only one leaving arc which reaches the node denoting the prototype of which the object is an instance. This arc carries the values that must be given to the attributes of the prototype to instantiate it. As an example, if we consider the prototype A_Prototype, represented in Figure 3 above, its instance, the graphical object Example, is represented by the tree of Figure 4.

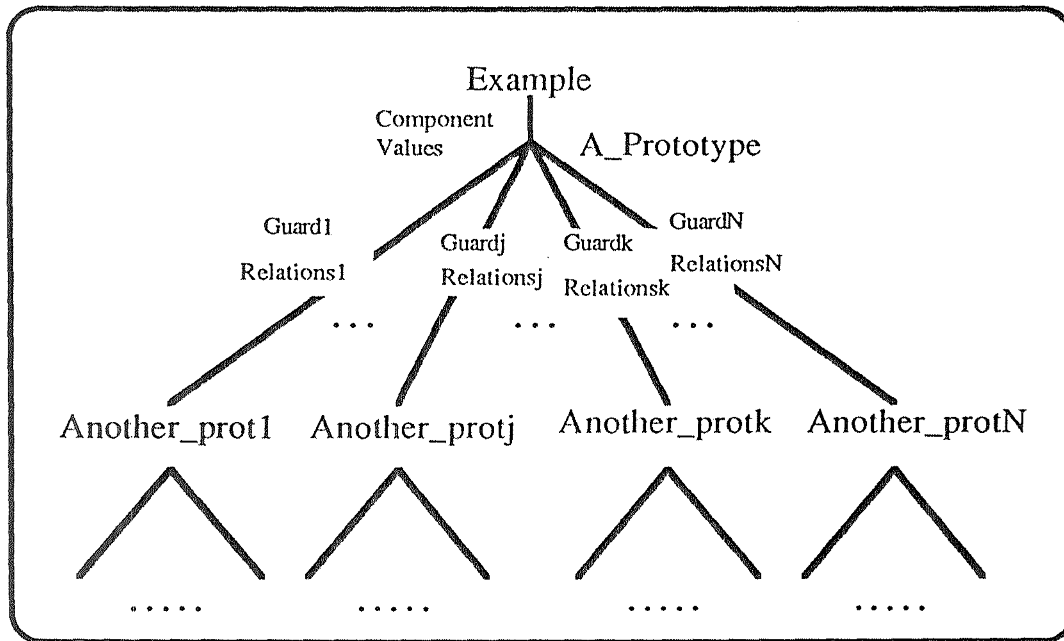


*Figure 4*

Note that when creating an object O, instance of a prototype P, no check that the actual values for an attribute A are in the set $V_{P_A}$ is performed (see Definition 4).

**Definition 19:** An object O, instance of a (correct wrt attributes) prototype P, is correct if, for all its current attributes, the corresponding properties are satisfied. ♦

Retrieval of an incorrect object will fail, i.e. its structure cannot be deduced, thus the object cannot be visualized.

## 3.3 Frame

In the following, we define the concepts of a frame and of its state. These concepts are introduced to make it possible to define integrity constraints on graphical objects at visualization time.

**Definition 20:** The *frame* is the abstract plane on which graphical objects will be drawn. ♦

**Definition 21:** The *state of the frame* consists of the name of the objects and their descriptions (i.e. the name of the prototype and its parameters) currently on the frame. ♦

## 4. Implementing the model by EDBLOG

In this section, we present the implementation of the model within the EDBLOG system.

EDBLOG handles facts, rules, integrity constraints and transactions, as described in Section 2.

The structure of EDBLOG can be depicted as in Figure 4.1, i.e. a logic theory $T_{KB}$, the knowledge base, and a meta-theory $T_{KBMS}$, its management system.

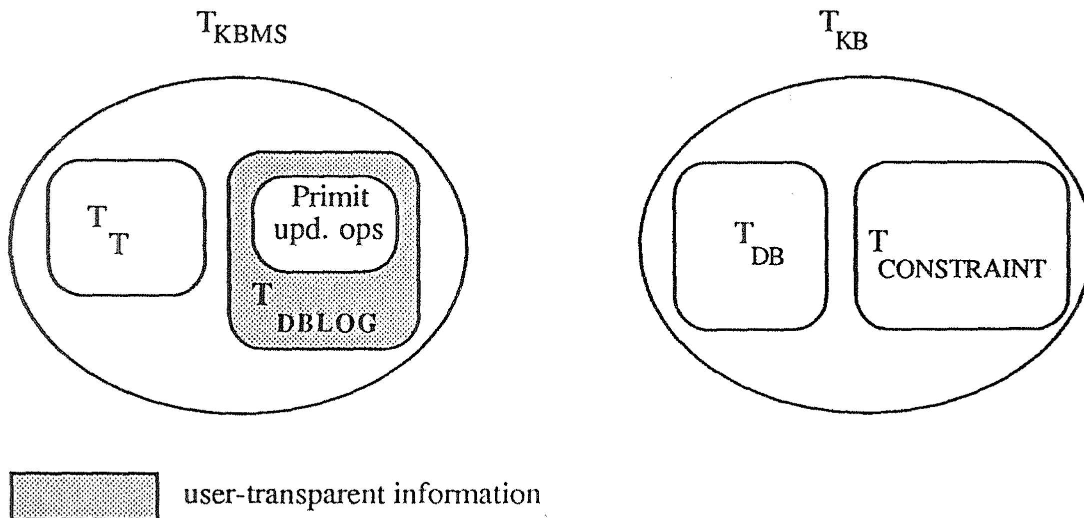


*Figure 4.1.*

In order to obtain the GRAPHEDBLOG system, the above structure has been modified as in Figure 4.2:

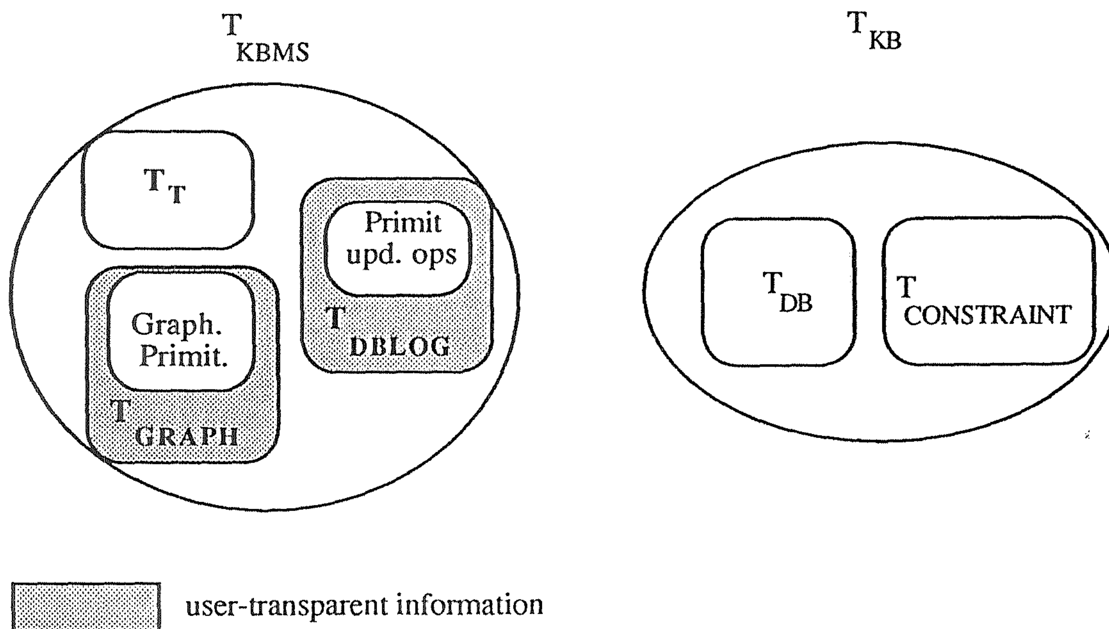


*Figure 4.2*

That is, all information that is set up by the user, such as the representation of user defined prototypes and of graphical objects, will be contained in the theory of the knowledge base, whereas rules needed to interpret the representations and the meta-interpreter to visualize the graphical objects, will be resident on the theory $T_{KBMS}$.

## 4.1 The linguistic constructs used to represent graphical objects

These constructs are of two kinds: i) those corresponding to basic primitive prototypes are defined in the theory $T_{KBMS}$, in particular in $T_{GRAPH}$, and the user can consider them as already existing in his KB; ii) those constructs that the user has to insert in his KB to define his own objects.

***Basic primitive prototypes***

The model provides for basic primitive prototypes that correspond to graphical primitives such as line, circle, etc.. whose description is not explicit in the database, but hidden to the user since they are considered as extending the set of standard system predicates.

Basic primitive prototypes are defined in $T_{GRAPH}$ by *facts* such as:

*basic_prototype(Geometry, State, Abs_Attribute1, ...),*

where

Geometry = *geometry(Origin, Scale, Rotation)* and

State = *state(Mode, ..., Color_Table).*

We have grouped together with the *geometry* functor, the geometric contextual attributes and, with *state*, the contextual state attributes.

Furthermore, for every absolute attribute, we have:

$Abs_Attribute_j$ = *attr_namej (Value).*

The functor $attr_name_j$ has to be considered as the attribute *builder* that, for an absolute attribute, helps in understanding the meaning of its associated parameter. For example:

polyline(Geometry, State, points_list(List))

denotes the polyline basic primitive prototype, parametric with respect to the contextual attributes (Geometry and State, here denoting variables) and also to the absolute attribute points_list, i.e. with respect to a list of points, fixed relatively to the Origin (part of the Geometry information), that contains the polyline vertices that must be connected by lines.

Figure 4.3 below lists all the basic primitive prototypes; the capital letters denote variables:

```
point(geometry(P,S,R),STATE).
line(geometry(P,S,R),STATE,sec_point(S_P)).
path(geometry(P,S,R),STATE,points_list(POINTS_LIST)).
rectangle(geometry(P,S,R),STATE,sec_point(S_P)).
polygon(geometry(P,S,R),STATE,points_list(POINTS_LIST)).
arc(geometry(P,S,R),STATE,point1(P1),point2(P2),point3(P3)).
text(geometry(P,S,0),ST,string(STRING)).
```

*Figure 4.3*

*User defined prototypes*

In the knowledge base ($T_{KB}$), the user inserts information about graphical objects, using the linguistic constructs that we are going to introduce.

Although it sounds more intuitive to represent a tree (see Section 3.1-3.2), in clausal logic, by a set of rules, it turns out that a representation by means of facts is easier to handle, since prototypes will be developed interactively. Another consideration in favor of a representation by means of facts is that we have to handle the priority factor that, in the context of a rule-oriented representation, determines the selection strategy of subgoals in the body. This is more complicated than letting the priority factor be a further parameter in a fact and using unification to perform the right selection order while relying on the fixed selection rule (leftmost) that almost all commercially available Prolog*s* have.

Let us now consider a sub-tree representing a prototype, that consists of two nodes linked by an arc having associated a guard, and a set of relations to bind the attributes of the connected prototypes. In addition, let us suppose that this arc also has a priority factor.

This fragment of tree is represented by the following fact:

*link (Defining_Prot,Used_Prot,Guard,Relations_List,Link_Priority).*

As an example, the following assertion represents an arc that leaves from the prototype *house* and reaches the prototype *window*.

```
link(house(Geometry,State),
    window(delta_geometry([50,100],1,0),delta_state([]),second_corner([30,50])),
    true,
    [],
    3).
```

That is, a use declaration is asserted whose meaning is that all instances of the prototype *house* consist of an instance of the prototype *window*; all instances of *window* will have their Origin coordinates at [50,100] with respect to the Origin of *house*; the second corner has coordinates [30,50] with respect to the window Origin; while, they have the same Scale, Rotation and State of the *house* instances.

It can be noted that the contextual attributes are represented by the following terms:

*delta_geometry(Delta_Origin,Delta_Scale,Delta_Rotation).*

and

*delta_state(List_of_State_Modifications).*

The first term fixes the geometric contextual attributes of the used prototype with respect to the corresponding attribute of the defining prototype.

The second term specifies the way in which the contextual state attributes have to be modified. This term contains a list of pair <state attribute builder, value>. Each pair denotes the change to be made to the state of the defining prototype in order to obtain the state of the instances of the used prototype (actual state).

As an example, if in the use declaration of a prototype, there is a term such as:

delta_state([fill_pattern(black)]),

then, the state of the instances of the used prototype is obtained from the state of the instances of the defining prototype by setting the fill_pattern attribute to black.

***Graphical objects***

Clearly, we do not represent the whole tree (see Section 3.2), but only the arc connecting an object with its prototype, i.e. we represent graphical objects by facts as follows :

*graph_obj(Obj_Name,Prototype_Instance).*

For example,

graph_obj(my_house, house(geometry([250,400],2,30),state(copy,..,standard)).

defines the graphical object my_house as an instance of the prototype house with origin at [250,400], scaled by 2, rotated by 30 degrees and with its own state.

***Properties***

As we stated in Section 3.1, Definition 8, properties have the purpose of limiting the range in which the attributes of a prototype may take a value.

We represent properties by a predicate *property,* and we define them either by rules or facts as follows :

*property(Prototype_Name,Attr_Builder(Value)).*

For example, we can have the following set of properties asserted on the definition of the prototype house :

property(house,origin([0,0])).

property(house,origin([50,40])).

property(house,rotation(X)) $\leftarrow$ X>0, X<30.

The previous clauses force every instance of the prototype house to have its origin at coordinates [0,0] or [50,40] and a positive rotation of less than 30 degrees.

Summarizing, in order to describe a user defined prototype, graphical object and properties, the following relations are available: **link/5**, **graph_obj/2** and **property/2.**

In order to guarantee that insertion of user defined non conditional prototypes is restricted to correct (w.r.t. attributes) prototypes, see Definition 16, a constraint is pre-defined in the

KB:

link (P,X, true, Z,W) $\rightarrow$ correct_wrt_attr(P,X, Y, Z,W).

The definition of correct_wrt_attr is also system pre-defined.

***Representation of the frame***

Let us see how the frame is represented.

The fact that a graphical object is on the frame is represented by a fact as follows:

obj_on_frame(ObjectName),

and by means of the following rule:

```
inst_of_prot_on_frame(S) ← obj_on_frame(ObjectName),
                           graph_object(ObjectName,P),
                           compound(P,S).
```

Where compound(P,X) gives an instance of a sub-prototype, X, that is part of the prototype instance P.

In this way, we are able to derive all instances currently set on the frame, starting from the names of the graphical objects on the frame.

## 4.2. Implementing $T_{KBMS}$

$T_{KBMS}$ consists of three sub-theories: $T_T$, $T_{DBLOG}$ and $T_{GRAPH}$.

Those theories are meta-theories with respect to the $T_{KB}$ theory (object theory).

This means that, by means of a *demo* [15] meta-rule, it is possible to prove facts about the KB (within the KBMS). Our system provides for a, *provable*, meta-predicate that works as a link between these different theories and is defined as follows:

provable(A) ← bridge(A, GOAL), call_edb(GOAL)

When a goal ← provable(B) is issued, then, by means of the bridge predicate, B is split into a set of subgoals that can be of two different kinds:

- subgoals on system predicates;
- subgoals on user defined predicates (EDB=KB);

For example:

let KB be the following:

age(john, 13).

age(mary, 18).

age(jack, 50).

senior(X) ← age(X,Y), Y>=18.

the internal representation in EDBLOG of KB will be :
edb(age(john, 13)).
edb(age(mary, 18)).
edb(age(jack, 50)).
edb(senior(X) ← age(X,Y), Y>=18).

that we denote by EDB.
Thus, once given a goal such as: ← senior(jack), the bridge predicate will split it into:
← edb(age(jack,X)), X>=18.
This can now be proved in the theory of EDB enriched by all system pre-defined predicates.
Given a theory for the knowledge base KB, so that EDBLOG can extend it with a theory for graphical objects, we have extended the set of pre-defined system predicates known by EDBLOG, with all predicates that will enable us to implement the model presented in Section 3. Thus, in $T_{GRAPH}$ the following rules exist:

prototype(P) ← base_prot(P).
prototype(P) ← provable(link(P,_,_,_,_)).

The above definitions state that a prototype is either a primitive or is user defined.
Furthermore, since prototypes must have an acyclic dependency graph, the following constraint is defined and implemented as a demon:

→ depend(P,P).

where depend is defined as follows:

depend(P,S) ← provable(link(P,S,_,_,_)).
depend(P,S) ← provable(link(P,E,_,_,_), depend(E,S)).
Note that the above definition properly works since the check is performed before inserting the new link in a database which only contains acyclic dependency graph.
We allow for an object to be defined in the DB even if it is not correct, see Definition 19.

create(Obj, $P_{IST}$)← Pname($P_{IST}$,Prot_name), prototype(Prot_name),
assert(edb(graph_obj(Obj,$P_{IST}$))).

The *delete* is exactly the same as in EDBLOG.
The *retrieve* is a query to the DB where the entire structure of the graphical object is traversed and checked for correctness of the actual values. We do not permit any "retrieve" operation to be performed on un-correct objects:

retrieve(Obj) ← provable(graph_obj(Obj,$P_{IST}$)), traverse_&_check($P_{IST}$).

Depending on the kind of prototype involved, whether primitive or user defined, the traverse_&_check predicate will check to see if $P_{IST}$ is a correct object, see Definiton 19. This means that, if $P_{IST}$ is user defined, all sub_instances have to be checked too. Thus, it will recursively traverse the whole tree, taking into account any guards. Note that, even if the object is not retrievable, any query whatsoever is always possible on the DB , e.g. it is possible to ask successful queries such as : ?link(P,X,_, _, _).

A graphical object can be *visualized* if it is correct. The visualization of a graphical object is a side effect of the traversing of its structure. This is obtained by meta-interpreting [18] the traverse where, every time an instance of a basic primitive prototype is reached, the corresponding graphical primitive is called.

This definition is expressed by the following rule:

visualize(Obj) ← provable(graph_obj(Obj,$P_{IST}$)), meta_traverse_&_check ($P_{IST}$).

Visualization (meta_traverse_&_check) produces a change to the database so that the logic representation of the frame is updated with the names of the graphical objects present on the screen.

## 5 An application example.

In this section, we show how the model and the language so far defined (i.e. the GRAPHEDBLOG environment) can be used to specify elements of the SADT (*Structured Analysis and Design Technique*) formalism, developed by D. T. Ross [13], [14].

The *SADT methodology* consists of a notation and a set of techniques to formalize complex requirements in a clear and concise manner.

The SADT includes a graphical language, named *Structured Analysis Language (SA)*, and a set of rules to use this language. The main goal of the SA language is to provide the user with a tool such that many complex situations can be specified by using the decomposition and structural relations.

An *SADT model* consists of an ordered set of SA diagrams. The sequence in which these diagrams are built reflects the *top down* approach to problem solving. Every diagram that describes (*expands*, in the SA terminology) a node representing a subproblem is identified by: a title; the name of the node that this diagram expands; a reference number. The diagrams have to be drawn on a single (abstract) page and consist of a set of nodes (more than three, less than six) and a set of arcs connecting the nodes.

Two kinds of SA diagrams are defined: *activity diagrams* (actigrams) and *data diagrams* (datagrams). In an actigram, the nodes denote activities while the arcs specify

the data flow. Vice versa, in a datagram, the nodes specify data and the arcs specify activities. We can say that actigrams and datagrams are *dual*.

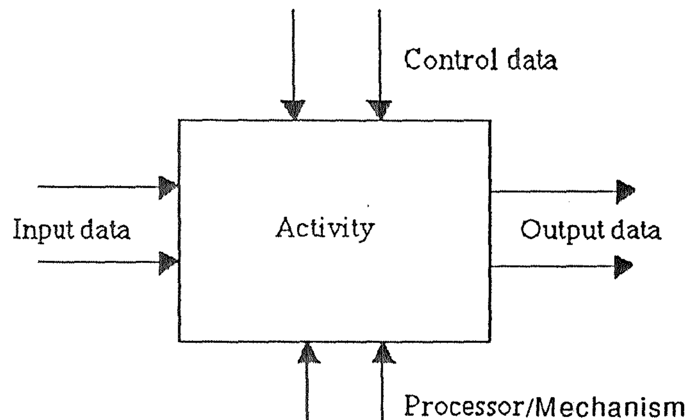


*Figure 5.1*

Figure 5.1 shows the format of an actigram node. Every side has a distinct meaning and there are four different kinds of arcs.

The concepts of input, output, control and mechanism are bound to the context of each node in an SA diagram.

The output supplied by a node can be: an input or a control to other nodes; an output to the external environment of the system described by the diagram. In this second case, there will be an arc that does not reach any node. In the same way, inputs and controls may come in from other nodes in the diagram or from the external environment.

In an actigram, the input and the output represent the flow of data and the mechanisms represent processors. The control represents data that are used, but not modified, by the activity.

The external inputs, outputs, controls and mechanisms of a diagram, that is, the decomposition of a node when used in another diagram, are limited to the inputs, outputs, controls and mechanisms that concern that node.

Each node described by an SA diagram must have at least one control arc leading to it and at least one output arc leaving it.

Figure 5.2 shows an example of an SA diagram.

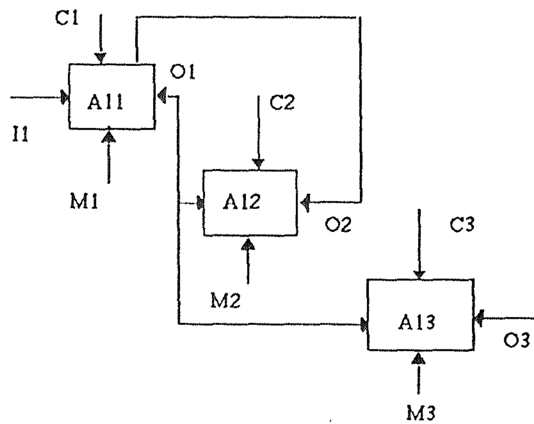


*Figure 5.2*

Given a prototype that describes a graphical object, our goal is to check that it is the description of a correct SA diagram.

We have defined the following prototypes:

**activity(Geometry,State,node_name(N)).;**
**arrow(Geometry,State,label(L),points_list(PL)).**

which must be used in the prototypes that describe SA diagrams.

The definition of the activity prototype is the following:

```
link(activity(Geometry,State,node_name(N)),
    rectangle(delta_geometry([0,0],1,0),delta_state([]),sec_point([400,200])),
    true,[],5).
link(activity(Geometry,State,node_name(N)),
    text(delta_geometry([10,160],1,0),delta_state([]),string(N)),
    true,[],1).
```

The definition of the arrow prototype is the following:

```
link(arrow(Geometry,State,label(L),points_list(PL)),
    path(delta_geometry([0,0],1,0),delta_state([]),points_list(PL)),
    true,[],5).
link(arrow(Geometry,State,label(L),points_list(PL)),
    polygon(delta_geometry([0,0],1,0),delta_state([fill_pattern(black)]),points_list(E)),
    true,[do_end_point_layout(PL,E)],3).
link(arrow(Geometry,State,label(L),points_list(PL)),
    text(delta_geometry(Pos,1,0),delta_state([]),string(L)),
    true,[do_position(PL,Pos)],1).
```

Let us note that in the last two use declarations of arrow, the positions the string associated to the arc and the orientation of the arrow have to be set are obtained, respectively, by means of the predicates **do_position** and **do_end_point_layout**. Their effect is to relate the attribute defining the list of points through which the arc will pass, with parameters of used prototypes.

Let us, finally, note that these prototypes are totally abstract. They only describe a box and an arc to which a string is associated.

We have already stated that an SA diagram is identified by a title, a name (the name of the expanded node) and a diagram number. This information may be denoted in GRAPHEDBLOG by the following kind of facts:

**diagram(Title,NodeName,NodeNumber,SAPrototype).**

From the above facts we assert in the data base the existence of an SA diagram whose title is Title, that expands the node NodeName, whose number is Number and that is described by the prototype named SA prototype.

We have thus defined a set of constraints to express that an SADT data base must not contain a wrong definition of SA diagrams.

A subset of these constraints is given in the following:

- diagram(Title,NodeName,NodeNumber,SAPrototype) -->
  integer(NodeNumber),prototype(SAPrototype).

guarantees that all definitions of diagrams, in the data base, are formally correct;

- diagram(Title,NodeName,NodeNumber,SAPrototype),link(SAPrototype,ProtUsed,_,_,_) -->
  use_of_diagram_element(ProtUsed).

guarantees that the prototypes used to describe SA diagrams consist only of arrows and activities;

- diagram(Title,NodeName,NodeNumber,SAPrototype) -->
  env_arrow_type_for_activity(NodeName,control_side).
- diagram(Title,NodeName,NodeNumber,SAPrototype) -->
  env_arrow_type_for_activity(NodeName,output_side).

guarantee that every node receives at least one control and generates at least one output.

It is important to note that these constraints are structural. That is, they are not concerned with the retrieval or the visualization of instances of the prototypes involved.

For practical reasons it may be useful to replace, in the condition part of each constraint, the term **diagram(A,B,C,D)** with **(diagram(A,B,C,D), inst_of_prot_on_frame(D))**, in order to obtain (making use of the logical representation of the frame) the automatic check

of the constraints every time the visualization of an instance of prototype declared as SA diagram is made, i.e. the state of the frame changes.

We have also defined a prototype that, given the name of a node (in addition to the contextual components), allows us to visualize its expansion in a correct layout. This prototype is **sa_diagram** and its description is the following:

```
link(sa_diagram(Geometry,State,node_name(N)),
  sa_form(delta_geometry([0,0],1,0),delta_state([]),title(T),node_name(N),node_number(R)),
   diagram(T,N,R,_),[],5).
link(sa_diagram(Geometry,State,node_name(N)),SAProtUse
  (diagram(_,N,_,SAPrototype),do_sa_use(SAPrototype,SAProtUse)),[].1).
```

Note that the second prototype (a variable named SAProtUse), which is part of the description of sa_diagram, has not been specified. It cannot be statically determined, but it can be dynamically obtained by evaluating the guard (diagram(_,N,_,SAPrototype), do_sa_use(SAPrototype,SAProtUse)), associated to the use declaration.

Finally , we observe that, thanks to the declarative environment, it has been possible to define, simply and naturally, all the properties that the description of a graphical object must satisfy in order to be considered an SA diagram. Furthermore, by means of the conditional inclusion, it has been possible to define elements whose structure is not static and uniquely fixed but can be modified, according to the state of the knowledge base, during the retrieval of those instances in which a conditional inclusion appears.

## 6. Concluding remarks

We have presented GRAPHEDBLOG, a system to declaratively handle graphical objects and implemented in EDBLOG a logic database management system environment.

The (graphical) data language is an extension of the language used to define a deductive database in EDBLOG: thus the syntax is based on Horn logic (definite clauses) and, as EDBLOG provides mechanisms to handle integrity checking, integrity constraints can be defined on graphical objects and on their visualization.

Since graphical objects are handled within a logic database, all the advantages of logic are exploited so that the resulting system is declarative, deductive and its semantics is well founded.

We should like to stress that the above mentioned features will make the final system very suitable for prototyping graphical applications. Among others, one application area that seems to be very suitable for a system like GRAPHEDBLOG is the visual languages area [4], since it is very natural to associate, to each graphical object, its corresponding operational semantics that, usually, includes also non graphical information.

As an example of application we have presented the definition of the SADT methodology. The example shows that only a very few rules are necessary to define and handle something

as complex this methodology is.

To obtain the final GRAPHEDBLOG system, we have extended EDBLOG by integrating it with the graphical system X-window [16]. The initial prototypical version of the system is implemented in Quintus Prolog and runs on a Sun 3 workstation under Unix 4.2 [Asirelli & al. 89]

The future developments of GRAPHEDBLOG, a part from providing it with a graphical interface for naive users, should include an extension to the model in order to handle the graphical input.

## References

[1] P. Asirelli, P. Castorina, G. Dettori, A **Proposal for a Graphic-Oriented Logic Database System**, Proc. of The Second International Conference on Computers and Applications, Pekin, June 1987.

[2] P. Asirelli, G. Mainetto, **Integrating Logic DataBases and Graphics for CAD/CAM applications**, IEEE WorkShop on Languages for Automation,Vienna, August 1987, pp. 173 - 176

[3] P. Asirelli, D. Di Grande, P. Inverardi, **GRAPHEDBLOG Reference Manual**, IR IEI B4-08 February 1990.

[4] S-K Chang, M.J. Tauber, B. Yu and J-S. Yu, **The SIL_ICON Compiler - An Icon-Oriented System Generator**, IEEE WorkShop on Languages for Automation,Vienna, August 1987, pp. 17 - 22

[5] M. De Santis, **Logic Programming and Databases: Un ambiente di Sviluppo adatto al trattamento dei vincoli di integrità**. Tesi di Laurea , Computer Science Dept., University of Pisa, 1985.

[6] J. D. Foley, A. Van Dam, **Fundamentals of Interactive Computer Graphics**, Addison - Wesley Publishing Company, 1982.

[7] H. Gallaire, J. Minker, J. M. Nicolas, **Logic and Databases: a Deductive Approach**, Computing Surveys, Vol.16, No.2, 1984, pp. 153 - 185.

[8] R. Helm, K. Marriot, **Declarative Graphics**, Lecture Notes in Computer Science No. 225, Springer - Verlag, London, July 1986, pp. 513 - 527.

[9] W. Hubner, Z. I. Markov, **GKS Based Graphics Programming in Prolog**, Computer Graphics Forum, Vol.5, March 1986, pp. 41 - 50.

[10] Lloyd, J. **Foundations of Logic Programming**, 2$^{d}$ edition, Springer-Verlag 1987.

[11] T. Lubinsky, I. Hutzel, An **Object Oriented Graphical Kernel System**, Computer Graphics World, July 1984, pp. 69 - 74.

[12] F. C. N. Pereira, **Can Drawing Be Liberated from Von Neumann Style ?**, Logic Programming and Its Applications, M. van Caneghem e D. H. D. Warren Edd., A.P.C., Norwood, New Jersey, 1986, pp. 175 - 187.

[13] D. T. Ross **Structured Analysis (SA): A LAnguage for Communicating Ideas**, IEEE Trans. Software Engineering, Vol. SE - 3, No 1, January 1977, pp. 16 - 34.

[14] D. T. Ross, **Applications and Extensions of SADT**, IEEE Computer, April 1985, pp 25 - 34.

[15] S. Safra, E. Shapiro : **Mata Interpreters for real**, Inf. Proc. 86. H-J Kugler (Ed.), 1986, pp. 271-278.

[16] R. W. Scheifler, J. Gettys, **The X window system**, ACM Transaction on Graphics, Vol.5, No.2, April 1986, pp. 79 - 109.

[17] D. L. Spooner, **Database Support for Interactive Computer Graphics**, Proc. SIGMOD, 1984, pp. 90 - 99.

[18] L. Sterling, **Expert System = Knowledge + Meta-Interpreter**, Dept. of Applied Mathematics, The Weizmann Institute of Science, Internal Report CS-84-17, 1985.

[19] P.Wisskirchen, **Geo++ - a System for Both Modelling and Display**, EUROGRAPHICS'89, Hamburg, September, 1989, pp.403 - 414.