Consiglio Nazionale delle Ricerche

ISTITUTO DI ELABORAZIONE DELLA INFORMAZIONE

PISA

GRAPHEDBLOG: Reference Manual

P. Asirelli, D. Di Grande, P. Inverardi

B4-08 Febbraio 1990

GRAPHEDBLOG

Reference Manual.

P. Asirelli, D. Di Grande, P. Inverardi

The *Conceptual Database* is the abstract representation of the physical database.

Views are abstractions of parts of the Conceptual Database.

Furthermore, there are another two dimensions to be taken into account, apart from the levels of abstraction we have seen:

- the **instances of the data base**, i.e. the current data in the database;
- the **schema**, i.e.the enumeration of the *entity types* and of *relations* among entity types, according to the level of abstraction referred to by the schema. Thus, for example, we can have a Physical Schema corresponding to the Physical Database, and, corresponding to the different views.

The *Data Model* is a set of logical structures used to describe the Conceptual Schema. The model has to be rich enough to be suitable to describe significant aspects of the real world, but, on the other hand, it has to make it possible to determine, almost automatically, an efficient implementation of the Conceptual Schema (by the Physical Schema).

It is difficult, and very important too, to determine the appropriate Data Model. In fact, the DM defines the general mechanisms to access the data, and, when such mechanisms are not suitable, the resulting Data Base may result in being very inefficient. Research in the field of Data Models is still active, yet the Entity/Relationship model is generally considered to be one of the most advanced, from the point of view of its expressiveness and naturalness. The Entity/Relationship model generalizes and extends the classical models, such as the Relational Model.

Traditionally, the Physical Schema and the Conceptual Schema are expressed by means of different languages, the second one being defined in terms of a programming languages to implement the Conceptual Schema. DBMS's also are implemented, often , using a different programming language and the query language for the external user often has a logic syntax to be interpreted onto the Physical Schema. Thus, often more than one language is involved in a DBMS and appropriate interpreters and algorithms have to be defined. As it will be clearer later on, logic offers a uniform language in which the Data Model can be defined and, such a language, being a programming language too, means that the implementation is immediate (the Conceptual Schema is also the Physical Schema), the query language is the same language used everywhere else and the DBMS too is defined using the same language, providing for definition and implementation. The interpreter and the algorithms are based upon the same mechanism, i.e. Resolution [Robinson 65].

1.2 The syntax used

Let us define the syntax of the logic language we will use so that the examples can be more easily understood. Let us stress that the language we use is exactly the one introduced first by Kowalski and van Enden in [Kowalski 74] and that it is compatible with all Prolog languages commercially available.

A logic program consists of a set of *clauses* (Horn Clauses).

Each clause looks like:

$A \leftarrow$	*facts* (unit clauses)
$A \leftarrow B_1, \ldots, B_n$	*rules*

where $A, B_1, \ldots, B_n$ are literals. A is the consequent, $B_1, \ldots, B_n$ are the premises and they look like $p\,(t_1, \ldots, t_m)$ where: p is a predicate symbol and $t_1, \ldots, t_m$ are *terms*.
The informal interpretation of a clause $A \leftarrow B_1, \ldots, B_n$ is that, A holds if $B_1, \ldots, B_n$ hold.

A term is either

* a *constant* symbol: an identifier beginning with a lower case letter;
* a *variable* symbol: an identifier beginning with an upper case letter;
* a term such as $f\,(t_1, \ldots, t_k)$ where f is a data constructor symbol (functor) and $t_1, \ldots, t_k$ are terms.

For more details on the semantics of the above language and its interpreter (SLD-resolution procedure) we suggest the reading of the following paper and books: [Kowalski 74] and [Lloyd 84].

1.3 Logic DataBases

In [De Santis 85] definitions on Deductive Databases and Logic Databases can be found. There can also be found details on the assumptions that are necessary to represent a Relational Database, by means of logic. In [Giannini & al. 86] it has been shown that, not only the Relational Model, but the Entity/Relationship Model too, can be represented by logic. In particular, an example is presented to show how an E/R model can be mapped onto a logic program.
Deductive Databases (DDB's), are extention of Relational Databases (RDB's) where deduction capabilities are introduced. While RDB's can be seen as a set of facts true in a world that has been

defined, DDB's are defined by a set of facts (Extensional Components) and a set of rules (Intensional Components). Rules permit you to deduce new facts from existing explicit ones. When rules are not recursive, they can be expanded to obtain just a set of facts.Thus an RDB can be straightforwardly represented by a deductive database, with no recursive rules.

A DDB can be seen as a first order theory, in particular, as a Horn clause theory. Thus, given the procedural interpretation of Horn clauses [Kowalski 74], a DDB can be regarded as a logic program where the only facts in the database are just those that can be deduced from the logic program by evaluating goals. This also means that, a logic programming language not only provides for the database definition language but also for the database query language.

Thus, a RDB can be represented (and implemented) by a logic program where rules are not recursive, i.e. by a hierarchic logic program. Referring to *fig. 1*, by using a logic programming language, the Conceptual Schema and the Physical Schema coincide.

Representing a database by a full logic program (with uncontrolled recursion) introduces problems of non-termination of the query evaluation process. On the other hand full logic programming capabilities extends Database capabilities from the point of view of the Data Model, and from the point of view of the Schemas (Conceptual and Physical). A compromise has to be found between the problem of termination and extention of capabilities.

When a LDB has to work as a deductive "question-answering" system for a relational database, three main problems have to be faced:
i - knowledge representation;
ii - knowledge acquisition;
iii - use of knowledge.

A Logic DataBase Management System is thus seen as system for "knowledge management".

While knowledge in such a system is represented by means of Horn Clauses, knowledge acquisistion has to be faced by defining updating operations which guarantee the database integrity consistency and/or redundancy.

The use of knowledge is instead related to the query language interface and the query evaluation process.

1.3.1 Querying the LDB

The most common use of Logic in the database field has been, until recently, confined to the query language and to integrity constraints formulas. In both cases an interpreter is then necessary to transform the formulas into the internal language, say QBE, SQL or the relational algebra language.

On the other hand, logic programs are used via resolution of goals, where the initial goal is considered as the main program. It immediately follows that, when the database is represented by a logic program, a query is nothing else than a goal to be resolved against the program. The query evaluation process is resolution. Integrity constraints are formulas which are properties of the logic program denoting the database and, in some cases resolution can still be used to verify them.

1.3.2 Basic updating operations

Updating operations in a LDB framework are related to knowledge acquisition. Operations are necessary to introduce new facts and rules and, also, integrity constraints formulas.

Furthermore, updating operations must provide for integrity checking. This means that, when a fact or a rule is introduced, the obtained database must be consistent with respect to integrity formulas. The updating request must be denied when it would lead the database into an inconsistent state.

The introduction of new integrity formulas should also cause verification of the actual database against the new formulas.
Updating operations also have to deal with redundancy problems. Such kinds of problems are related to implementation and installation issues. They do not affect the correctness of the system or its logical consistency.

1.3.3 Integrity Constraints handling

As we consider a logic database to be a logic program, integrity constraints (properties which the database must posses), can be considered as properties of logic programs, thus assimilating the problems of integrity constraint checking to that of logic program property proving. In addition, a deductive database system should offer much more than a logic programming system, since its objects are evolving first-order theories (databases), rather than a single fixed one. In particular, the problems of consistency and redundancy must be faced.

Although logic programming offers a straightforward way of implementing deductive databases, some restrictions are needed to guarantee the termination of the query evaluation process and the evaluation of negative queries. Thus the class of logic programs has to be restricted to hierarchical program definitions which do not allow recursive definitions [Clark 78], [Shepherdson 84]. These restrictions can be partially relaxed, at least with respect to negation and to certain kinds of queries [Barbuti & al. 86].

In [Asirelli & al. 85] an approach to integrity constraints handling for hierarchic databases is proposed, in which a database is considered as consisting of a logic program plus a set of formulas, which must be proved to be true in the minimal model of the given program. Since a database will be updated, two approaches are proposed for integrity constraints checking. One approach (**The Modified Program Method**) considers a subset of the given logic formulas, called *IC - Integrity Contraints*, and uses them to modify the logic program automatically so that the given formulas are true in its minimal model (with respect to the model theoretic semantics).This means that all facts which do **not** satisfy *IC* are **not** provable/derivable from the modified logic program/DB (i.e. illegal queries cannot succed). The other approach (**The Consistency Proof Method**) considers a wider class of logic formulas (called *Controls*), and proves that they are true or false using a metalevel proof, on request from the user. The description of the algorithms is sketched in the next section, while a detailed description of them can be found in [Asirelli & al. 85] and in [De Santis 85].
The integrity constraint formulas and the integrity checking algorithms can be extended to work on database which admit some recursion in the spirit of Barbuti. Stratified databases can be considered too.

1.3.4 Redundancy

Redundancy problems are related to excess of information. That is to say that, for example, when a fact is added to the database and the same fact is already derivable, then a choice has to be made depending on time or space considerations.

Time considerations concern time of response in the query evaluation process, while space considerations concern the amount of storage needed for the database.

Generally it is faster to derive information which is explicitely stated than to derive it by rules.
Thus, time considerations encourage the introduction of facts instead of rules.
On the other hand, rules denote a set of facts succinctly. That is, rules allow you to save on the storage space.

The above considerations must be taken into account when adding redundant information. If time

has to be saved then redundant facts are accepted, while if space has to be saved then they have to be rejected.
This all means that an LDBMS should provide for two modes of behaviour, letting the user choose between them depending on the machine being used.

1.3.5 Transactions

When a DBMS becomes something more than a toy system, the user has to be provided with facilities to express compound updating operations. Compound updating operations, in the framework of databases are often called *transactions*.

A transaction definition language is generally defined, often it is yet another language with its own interpreter that is added to a DBMS. Transactions allow the user to define his own operations at a more abstract level, in terms of other transactions or a repetition of basic updating operations.

Execution of transactions involves problems of consistency and redundancy as well as basic updating operations. The database has to remain in a consistent state, or it has to be reset into a consistent state after system crashes or errors occur, thus abortion facilities have to be provided to *undo* the effects of a transaction.

Of course, in a logic framework, the transaction definition language can still be based on logic. This does not require the user to learn a new language and, from the implementation point of view, less effort is needed to build the interpreter using, once more, the basic resolution procedure used throughout the system.

2. The Logic Database Management System (LDBMS) GRAPHEDBLOG

The logic database management system GRAPHEDBLOG [Di Grande 89] is a system which has the capability to manage databases which contain graphical and non graphical information.
GRAPHEDBLOG has been defined as an extension of EDBLOG [Mauro 85] by introducing the capability of manage graphical information. EDBLOG, in its turn, has been defined as an extention of DBLOG [De Santis 85] by introducing transaction definitions and handling facilities. DBLOG is the kernel of the LDBMS.

GRAPHEDBLOG considers the data base system as consisting of four parts:
a) a logic program in which:
 a.1) *the set of facts*, "unit" Horn clauses, are considered to be the Extensional

component of the DB (EDB);

a.2) *the set of deductive rules*, "definite" Horn clauses, are considered to be the Intensional component of the DB (IDB);

b) a set of integrity constraint formulas with:

b.1) *a set of Integrity Constraints* (*IC*), which are formulas of the form:

$\mathbf{A_k} \rightarrow \mathbf{B_1}, \ldots, \mathbf{B_s}$

which can be interpreted informally as: whenever $\mathbf{A_k}$ is true then $\mathbf{B_1}$ and...and $\mathbf{B_s}$ must also be true;

b.2) *a set of Controls formulas* which are either formulas as in b.1) or else

i) $\mathbf{A_1}, \ldots, \mathbf{A_m} \rightarrow \mathbf{B_1}, \ldots, \mathbf{B_n}$

ii) $\rightarrow \mathbf{B_1}, \ldots, \mathbf{B_n}$

iii) $\mathbf{A_1}, \ldots, \mathbf{A_m} \rightarrow$

The informal interpretation for i) is that whenever $\mathbf{A_1}$ and ... and $\mathbf{A_m}$ are true then $\mathbf{B_1}$ and ... and $\mathbf{B_n}$ must also be true; analogously ii) means that $\mathbf{B_1}$ and ... and $\mathbf{B_n}$ must be true and, finally, iii) means that $\mathbf{A_1}$ and ... and $\mathbf{A_m}$ must be false.

Note that for formulas i)-iii), as well as for the formula b.1), all the variables are intended to be universally quantified, apart from the local variables (i.e. variables occurring only on the right hand side) which are intended to be quantified existentially.

c) a set of clauses which define compound updating operations (*transactions*), which are formulas of the form:

i) $\mathit{trans}_i \leftarrow \mathbf{prec} \mid \mathit{trans}_1, \ldots, \mathit{trans}_n \mid \mathbf{post};$

The language used to express this kind of transaction syntactically resembles Concurrent Prolog with no annotated variables [Shapiro & al. 83]. The informal interpretation is that to execute the operation *trans*, the precondition (**prec**) must be first verified, and then the clause containing this precondition must be committed, the body executed and the corresponding postcondition verified. As in Concurrent Prolog, the commit operation is a way of expressing the behaviour of the Prolog cut operator, a failure in the body of a transation causes the failure of the transation.

ii) $\mathit{trans}_i \leftarrow \mathbf{prec}\ \#\ \mathit{trans}_1, \ldots, \mathit{trans}_n\ \#\ \mathbf{post}.$

The informal interpretation of this kind of transaction is that to execute the operation *trans*, the precondition (**prec**) must be first verified, and then the clause containing this precondition must be utilized, the body executed and the corresponding postcondition verified. A failure in the body or in the postcondition not causes the failure of the transaction but the search of anather definition for *trans* with the precondition verified. The transaction fails if all its definitions fail.

Preconditions and postconditions in the definitions of transactions will operate as particular forms of *Controls* which must be checked before/after the execution of the set of operations (body of the transaction).
Since checking for consistency in a DB can be very heavy and time consuming, preconditions and postconditions are introduced to separate global DB controls (*Controls*) from those related to particular transactions, thus reducing the number of necessary global *Controls* formulas.

The operational interpretation of these transaction definitions is the standard Prolog resolution of clauses where clauses are tried in the order they appear in the program.
The successful evaluation of a transaction causes the *Controls* formulas to be checked. The required transaction operation is aborted if this *Controls* checking fails. The abortion of a transaction is automatically handled (by backtracking), by ensuring that elementary updating operations are backtrackable upon failure. Abortion is also started upon the failure of postconditions or upon the failure of some operations of the body, thus obtaining an *and-nondeterministic* behaviour of the clauses.

The system can be seen as an amalgamated theory [Bowen & al. 82], [Bowen 85] consisting of the meta-theory (the theory which handles the evolution of the data base), and the object theory (the logic data base).

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*, *Controls* formulas and transactions to be added and deleted.

2.1 The Logic Database Kernel

The elements described in the points a) and b) form the basic components of the kernel (DBLOG), and can be depicted as in fig. 2.

According to **The Modified Program Method**, IC are used to modify the given set of Facts and Rules, to obtain a new set of facts and rules denoted by Facts1 and M-rules in fig.2, where: Facts1 is a subset of Facts and M-rules consists of both facts which become rules and rules which are modified by the modified program approach algorithm.

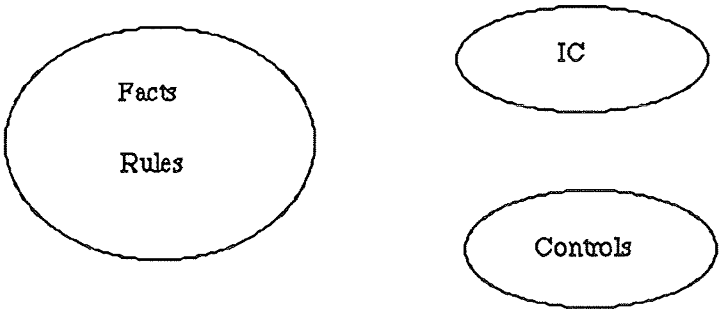


fig. 2

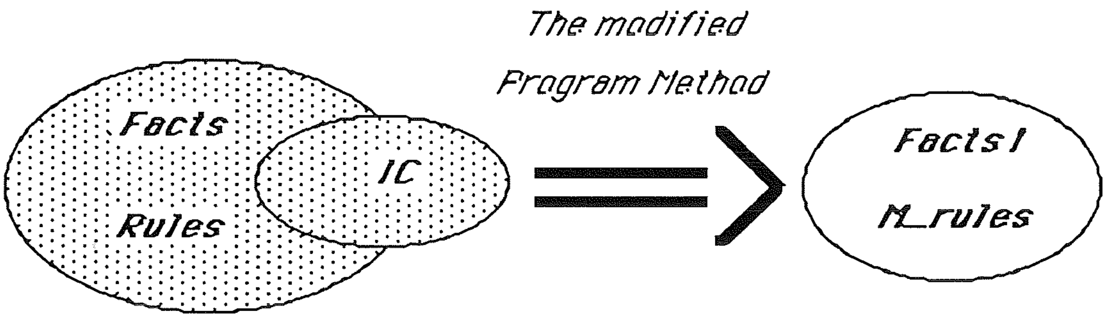
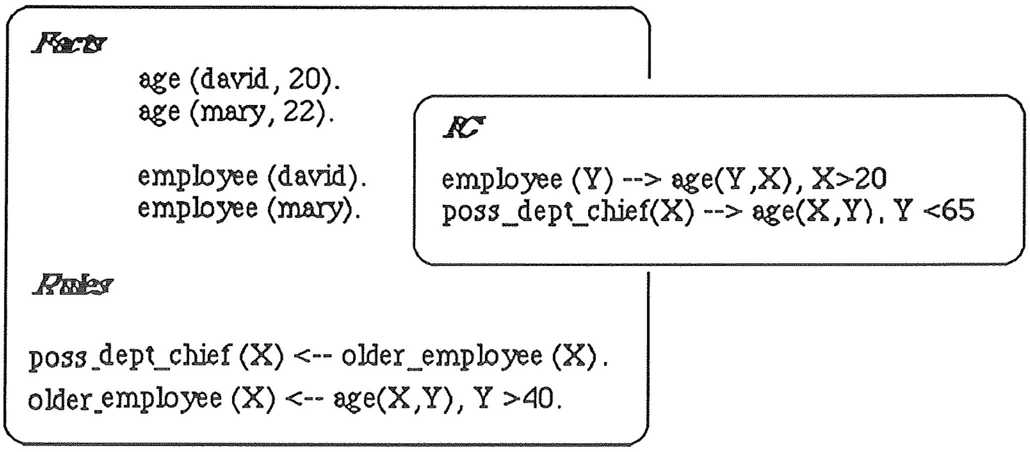


fig. 3

For example:



```
Facts
        age (david, 20).
        age (mary, 22).

        employee (david).
        employee (mary).

Rules

poss_dept_chief (X) <-- older_employee (X).
older_employee (X) <-- age(X,Y), Y >40.

IC
employee (Y) --> age(Y,X), X>20
poss_dept_chief(X) --> age(X,Y), Y <65
```

fig. 4

Then the resulting database to be considered, after running the algorithm for the modified program method, is:

```
Facts
        age (david, 20).
        age (mary, 22).

M_rules

  employee (david) <-- age(david, X), X >20.
  employee (mary) <-- age(mary, X) , X >20.
  poss_dept_chief (X) <-- older_employee (X), age (X, Y), Y < 65.
  older_employee (X) <-- age(X,Y), Y >40.
```

fig. 5

The other component of the kernel system, i. e. Controls are used by **The Consistency Proof Method** algorithm which verifies them against the current database.

This method considers one control formula at a time. Let us consider a formula such as:

$$A_1,..., A_m \rightarrow B_1,...,B_n$$

then the formula $A_1,..., A_m$ is considered a goal and it is resolved in the database, for all values of the variables. Let $J_1, J_2,..., J_n$ be all the answer substitutions for that goal. For each J_i,

$$B_1,...,B_n$$

is rewritten by substituting each variable with its corresponding assignment in J_i, the obtained formula $[B_1,...,B_n] J_i$ is then resolved in the database. If $[B_1,...,B_n]J_i$ succeeds for all $i=1,...,n$ then we say that the database is consistent with respect to that control formula.

As an example, let us consider the formula:

$$a(X, Y, Z)\rightarrow b1(X), b2(X,Y), b3(X,Y,Z)$$

we resolve $\leftarrow$ a(X, Y, Z) for all solutions. Let (X=c, Y=d, Z=f) and (X=a, Y=r, Z=s) be the only solutions, then we look for the success of the two goals:

←b1(c), b2(c,d), b3(c,d,f) and ←b1(a), b2(a,r), b3(a,r,s).

Both the modified program and the proof method algorithms have to be implemented at the metalevel, where the object theory is the database and the set of formulas to be proved.

2.2 The graphical model and its language

GRAPHEDBLOG supports the management of data bases which store graphical data according a model that consists of the following elements:

- *prototypes*, i.e. templates of graphical objects;
- *graphical objects*, i.e. instances of prototypes;
- *properties*, i.e. properties asserted on prototypes definitions;
- *mechanisms* to:
  - compose prototypes;
  - create graphical objects;
- *frame*, i.e. the abstract plane in which graphical objects are drown.

2.2.1 Prototypes

Sometimes, it can be useful to have different views of the same object; furthermore, the same graphical object may be used one or more times as a sub-object in more complex objects. Thus, we introduce the notion of prototype that can be assimilated to the notion of *generic* type in a programming language. That is, its definition does not define a new object, but it is a template that will be used to *create* objects upon instantiation.

Definition 1: A *prototype* consists of the description of a graphical object. This description is parametric with respect to some characteristics (attributes).

Prototypes are divided into two further classes. That is:

- *basic primitive prototypes*, that are the usual graphical output primitives (e.g. point, polygon,..). The description of these prototypes is hidden to the user and they are system primitives, i.e. this class of prototypes is fixed once and for all and cannot be changed by the user.

- *user defined prototypes*, that can be:
  - *compound user defined prototypes*. These are defined as composition of other user defined prototypes or basic primitive prototypes or both. These prototypes must have a hierarchic structure (not recursive).
  - *primitive user defined prototypes*. These elements are defined only in terms of basic primitive prototypes and are obtained from user defined prototypes by *compiling* t hem.

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 prototype, eliminates its dependencies from its composing prototypes; This means that primitive user defined prototype consists only of basic primitive prototypes. This class of primitive user defined 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.

Each prototype description has several attributes for the creation of particular instances, according to user requirements. Two classes of attributes are provided:

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

All prototypes description is parametric with respect to contextual attributes and, optionally, to some absolute attributes, too.

Definition 2: The set of parametric components (attributes) of a prototype definition determines its *interface*.

2.2.1.1 Contextual attributes

The contextual attributes are of two kinds: The former consists of the *geometric* attributes, i.e.:

- *origin*;
- *scale;*
- *rotation.*

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

The latter consists of the *state* attributes, i.e.:

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

2.2.1.2 Absolute attributes

In general, one would like a prototype to denote the partial description of a graphical object where some aspects, a part from the contextual attributes, are left unspecified. For example, someone may wish to define prototypes such as the *arch(Length, Height)*, that describe an arch, parametrically with respect to its length and height. This kind of prototypes are called *parametric prototypes*. Their description is parametric with respect to some aspects that intentionally are not specified (the *absolute attributes* class), apart from the contextual attributes.

2.2.2 Graphical objects

A graphical object is obtained by fully instanciating a prototype i.e. providing it with all the information required by its description, thus generating a **ground** instance of the prototype definition.

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

- create;
- delete;
- retrieve;
- visualize.

Since the model will be implemented in a logic DBMS (EDBLOG), each one of the above operations is defined in terms of operations on the DB that contain prototypes, instances, etc.. Thus, the *create* will have the effect of inserting, in the DB, a fact to give a name to a prototype instance; *delete*, causes the deletion of a previously inserted object (fact); *check*, is a query to the DB; *visualize* is defined in terms of the check operation and, as a side effect it depicts the object on the screen.

2.2.3 Compound prototypes

The description of a user defined prototype consists of a set of *use* declarations of other prototypes. This permits to model a graphical object as the composition of several sub-objects.

If a graphical object O_1 is a structural element of a more complex object O_2 then, the description of prototype P_2 of O_2, must contain the use declaration of prototype P_1 of O_1. The use declaration must specify the information that are necessary to obtain, from the actual values of the attributes of P_2, the values of the attributes of P_1. Such information will be used to obtain from the instance O_2 the instance O_1.

Therefore, in every use declaration, the attributes of the defining prototype must be bound to the attributes of the used prototype. The values, fixed for the geometric attributes, must be expressed in terms of the attributes of the defining prototypes, the other values must be fixed in an absolute way.

Let us introduce the concept of dependency.

Definition 3: A prototype P_1 *depends* on a prototype P_2 if the description of P_1 contains an use declaration of P_2 or, if it contains an use declaration of a prototype P_3 that depends on P_2.

Given a prototype P, its dependencies can be represented as a graph, *dependency graph*, where the root denotes P and, the other nodes, denote all the prototypes P depends on. An oriented arch, from a node N_1 to a node N_2, shows that, in the description of the prototype denoted by N_1, there are one, or more, use declarations of the definition of the prototype denoted by N_2.

If a prototype has an acyclic dependency graph then it is hierarchic, i.e. it does not depend on itself.

Definition 4: A prototype is *correct* if its dependency graph is acyclic.

With this definition we want to underline that the only prototypes that can be described in our model are those that are hierarchies.

2.2.3.1 Conditional use declaration

The model we are describing is based on the assumption that the description of graphical objects must to be stored in a database together with non-graphical information.
This situation allows us to consider cases in which the use of a prototype P_1, within a prototype P_2, is subordinate to the existence of certain information in the database. In programming languages, this corresponds to the concept of *record with variants*.

In practice, taking advantage of the mixed environment we have, we want to be able to model cases in which some information about our data is not available at definition time, while they will be at querying time (checking and/or visualization time).
Hence, the structure of a graphical object is no more static and fixed at definition time. This is why we introduce the concept of *conditional inclusion.*

Definition 5: A prototype P_2 *conditionally includes* a prototype P_1, if the conditions that guard the use definition of P_1 hold when checking the instances of P_2. These guards may be queries to the database or conditions on actual values of the instance of P_2.

This mechanism allows us to define graphical objects that, at visualization time, show a different structure depending on the state of the database.

2.2.4 Properties

As we have previously seen, the description of a prototype defines the graphical structure of an object we want to represent, parametrically with respect to some attributes. Sometimes, it is useful to limit the range over which these attributes may change; to say it in other words, it may be useful, or even necessary, to *type* such attributes. Thus, we have introduced the concept of *property*.

Definition 6: A *property* is a couple <*name*, *value*>, where *name* is an identifier for an attribute and *value* denotes its value. Each property is associated to the definition of a prototype.

Definition 7: Given a prototype **P**, parametric with respect to an attribute **A**, the set of properties for **A**, determines the range of values $\mathbf{V_{P_A}}$ for **A**.

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

Definition 8: An instance of a prototype P that does not satisfy the corresponding properties, for all its actual attributes is *inconsistent.*

Definition 9: A graphical object is *inconsistent* if one of its sub-object is *inconsistent.*

The retrieval of an inconsistent object fails, i.e. its structure cannot be deduced thus, the object cannot be visualized.
If no attribute is defined for a component, then it can be assigned any value.

2.2.5 Frame

In the following, we define the concepts of frame and of its logical representation. As it will be clear in the following section, these concepts are introduced in to make it possible to define integrity constraints on the graphical object at visualization time.

Definition 10: The *frame* is the abstract plane on which the graphical objects will be drawn.

Definition 11: The *logical representation* of the frame is the set of information concerning the graphical objects currently drawn on the frame.

2.2.6 The language

In this section we present the language to represent graphical objects, according to the model introduced in the previous section.
The syntax of the language is Horn logic, thus it is expressed by means of definite clauses. The

environment in which the language is used is a deductive database environment, thus the semantic of the language is expressed in terms of operations on the database containing the representation of graphical objects. Furthermore, the language is defined as an extension of the logic language in which a deductive database is represented within the EDBLOG Logic Database Management System, thus besides facts and rules, integrity constraints (expressed in clausal logic too) can be defined, and are handled by the EDBLOG system.
Since we are using a logical language, it has to be noted that the graphical representation can easily be considered not only as a set of procedure definitions, but as a set of data that are to be interpreted to obtain the visualization of objects as a side effect to the query to the database.
Furthermore, having for granted the possibility of expressing integrity constraints, gives the user a chance to put integrity on the structure of graphical objects and on their visualization.

2.2.6.1 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 and the only information provided is their interfaces.
Basic primitive prototypes are denoted by **terms** such as:

basic_prototype(Geometry,State,Abs_Attribute $_1$***,...)***,

where

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

and

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

We have grouped together, respectively, in *geometry*, the geometric contextual attributes, in *state*, the contextual state attributes.

Furthermore, for every absolute attribute, we consider:

Abs_Attribute$_j$ = *attr_name*$_j$ *(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) and with respect to the absolute attribute points_list, i.e. with respect to a list of points, fixed relatively to its

Origin (part of the Geometry information), that contains the vertices that must be connected by lines.
The list of all the basic primitive prototypes is given in fig. 6

```
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)).
```

fig. 6

2.2.6.2 User defined prototypes

A prototype P that consists of (uses) N sub-prototypes, can be graphically represented by a tree where: the root denotes the prototype itself; the nodes, directly connected to the root (level 1 of depth), denote the N sub-prototypes that P uses. Every arch in the tree represents a declaration of use (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 arch, in the trees, carries a set of information:

- a guard: a condition that has to be evaluated to *true* (when the arch 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 will be visited.

Figure 7 exemplifies a tree representation for a prototype.

The logic description of user defined prototypes, will be given according to such graphical representation. Although it sounds more intuitive to represent a tree, in Horn logic, by a set of rules, it turns out that a representation by rules is not suitable to be really used in an environment where prototypes are interactively developed. In fact, an interactive development of prototypes requires changes to the trees thus resulting in many insertions and deletions of rules in the database. Furthermore, because of the priority factor, we need to evaluate sub_goals in the body of a rule, selecting them according to various selection rules. To this respect, almost all commercially

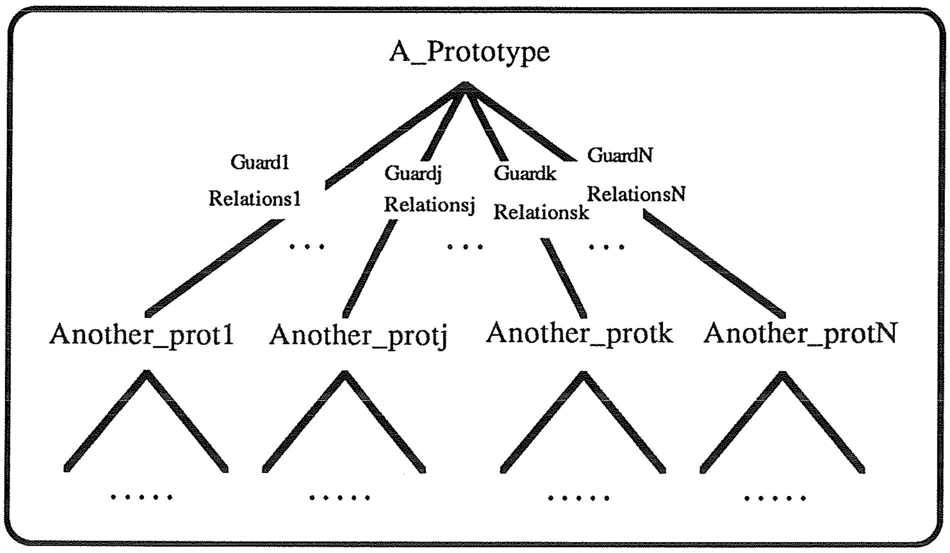


fig. 7

available Prologs have a fixed selection rule (leftmost) thus, we decided to represent a tree by means of assertions (facts).

Let us consider a sub-tree that represents a prototype, consisting of two nodes linked by an arc having associated a guard, and a set of relations to bind the attributes of the involved prototypes. In addition, let us suppose that such an arc has also 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 represent an arch 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, an use declaration is asserted whose meaning is that all instances of the prototype *house* consists 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 relatively 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.

2.2.6.3 Representation of the graphical object

As it should be clear by now, a graphical object is an instance of a defined prototype.
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 that reaches one node, the prototype which the object is an instance of. This arch carries the values that must be given to the attributes of the prototype to instanciate it.
As an example, if we consider the prototype A_Prototype, represented in figure 7 above, its instance, the graphical object Example, is represented by the tree of figure 8.
Clearly, we do not represent the whole tree, but only the arch 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)),

define 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.

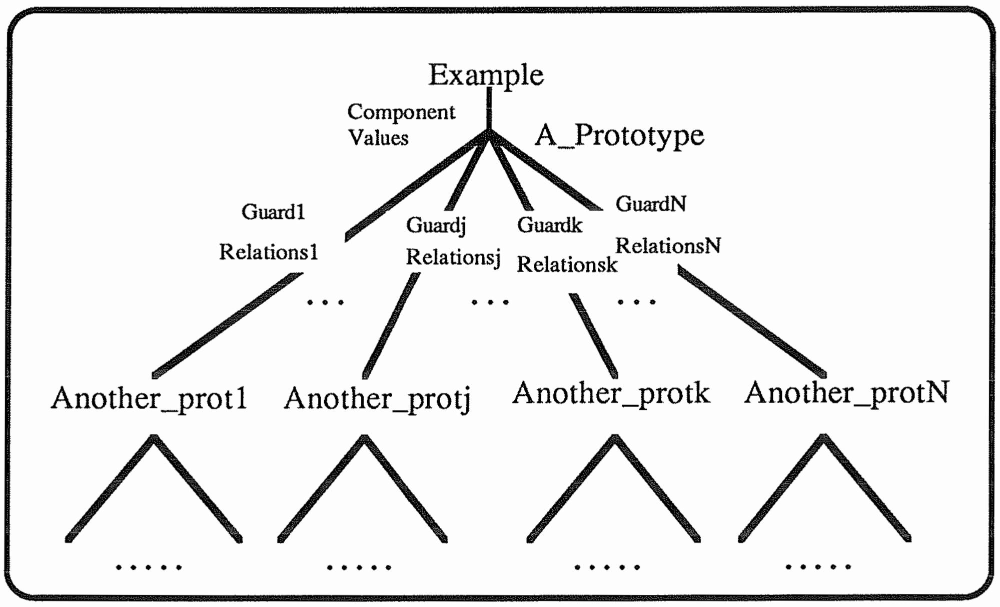


fig 8

2.2.6.4 Defining properties

As we said in section 2.4, properties have the purpose to limit the range in which the attributes of a prototype may change .
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)) ← X>0, X<30.
```

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

3. Interaction with GRAPHEDBLOG

3.1 Starting GRAPHEDBLOG

If you want to access to the GRAPHEDBLOG system, before to do it, you must active the X-Window system with the **xinit** command.
Now if you are connected with a machine without Quintus Prolog, you must activate a connection (**rlogin**) with one that has got it.
After this, you must set the current directory to the one where there is the database that you want to manage or to create.
Now you can start GRAPHEDBLOG with the command **GEDBLOG**, and when the Prolog prompt "yes" appears, type the command "**start.**".
After this, the GRAPHEDBLOG system ask you the database name that you want to manage and the display name where you want the interaction. Answer to it to initiate an interaction.
(Note that the display name of a machine named *name* is *name:0.*)

3.2 GRAPHEDBLOG operations

These are implemeted at metalevel (i.e. they operate on the object theory which corresponds to the database).

3.2.1 Query the database

The query must have the form:

$$q(X_1,...,X_n) \text{ :- } \textit{Goal}$$

where $X_1,...,X_n$ are the variables unbound in *Goal.*
The *Goal*, has to be an *allowed query*, that is:
a) The predicates in Goal must be:
- predicates already defined by the user in the current database;
- *standard* predicates;
- *not* and *;* predicates (metalevel).

b) It must be verified that in every substitution of calculated answer for Goal, there is a value for every instance of $X_1,...,X_n$.
c) The standard predicates must be used with an adequate number of bound variables (safe goal).

If the query is closed the answer will be Yes or No, otherwise the system will list all the solutions if they exist.

User Interaction:

1) If the query window is not mapped:
    1.1) press the left button of the mouse;
    1.2) select the query option in the pannel menu.
   After these operations the "query window" is mapped;
2) Write the query in the "query window";
3) Write the "full stop" and press "return" to execute the operation;
4) If you want to unmap the "query window" clic on the white rectangle in the upper lefthand corner of the same window.

3.2.2 Change mode

GRAPHEDBLOG can work in one of the following two mode:
a) *Time mode*;
b) *Space mode*.

When GRAPHEDBLOG is working in time mode, the insertion of a fact is really executed even if it is already derivable in the database. If the system is working in space mode, the insertion is not really executed. Furthemore, If a rule is inserted, then all the facts that can be derived by the rule are deleted, if the mode is set to 'spacemode'.

User Interaction: To switch from a mode to another:

1) press the left button of the mouse;
2) select the change mode option in the pannel menu.

3.2.3 Fact insertion

A fact must be an expression of this form:

$$p(A_1,...,A_n)$$

that must respect the following conditions:
a) $A_1,...,A_n$ must be ground component or bound to a function;
b) if p is already defined then its arity must be respected.

If these condition are respected, the result of the operation must be:
- the fact is inserted;
- the fact exists already;
- the fact is inserted but isn't derivable because an IC isn't satisfied;

- the fact isn't inserted because it is already derivable and the system is working in space mode.

User Interaction:

1) If the fact insertion window is not mapped:
   1.1) press the left button of the mouse;
   1.2) select the insert fact option in the pannel menu.
   After these operations the "fact insertion window" is mapped;
2) Write the fact in the "fact insertion window";
3) Write the "full stop" and press "return" to execute the operation;
4) If you want to unmap the "fact insertion window" clic on the white rectangle in the upper lefthand corner of the same window.

3.2.4 Rule insertion

The rule must be of this form:

$$A \leftarrow B_1,\ldots,B_n$$

with A, $B_1,\ldots,B_n$ atoms. The rule must respect the following conditions:

a) if A is already defined, then its arity must be respected;

b) the body of the rule must be a correct query, but the metalevel predicates *not* and *;* are forbiddens;

c) the rule must respect the gerarchic constraint;

d) all the variables in the head of the rule must be in the body of the same rule;

If these conditions aren't respected the operation is aborted.
If the rule is inserted when the system is working in space mode and there are facts that can be derived by this rule, then these are deleted.

User Interaction:

1) If the rule insertion window is not mapped:
   1.1) press the left button of the mouse;
   1.2) select the insert rule option in the pannel menu.
   After these operations the "rule insertion window" is mapped;
2) Write the rule in the "rule insertion window";
3) Write the "full stop" and press "return" to execute the operation;
4) If you want to unmap the "rule insertion window" clic on the white rectangle in the upper lefthand corner of the same window.

3.2.5 Integrity Constraint insertion

A Integrity Constraint is e formula as:

$$A \rightarrow B_1,...,B_n$$

The conditions that must be respected are those that could respect the corrisponding rule

$$A \leftarrow B_1,...,B_n$$

In addition to this, before the insertion, the predicate A must be already declared with the same ariety.

User Interaction:

1) If the integrity constraint insertion window is not mapped:
   1.1) press the left button of the mouse;
   1.2) select the insert integrity constraint option in the pannel menu.
   After these operations the "integrity constraint insertion window" is mapped;
2) Write the integrity constraint in the "integrity constraint insertion window";
3) Write the "full stop" and press "return" to execute the operation;
4) If you want to unmap the "integrity constraint insertion window" clic on the white rectangle in the upper lefthand corner of the same window.

3.2.6 Control insertion

A check is a formula of this kind:

c1) Condition $\rightarrow$ Conclusion

c2) always $\rightarrow$ Conclusion

c3) never $\rightarrow$ Condition

with Condition and Conclusion conjunctions of atoms.
For c1) the condition that must be respected are:

i) the predicates in Condition must be already defined in the database;
ii) the predicates in Conclusion must be already defined in the database;
iii) Condition must be a safe goal (see query operation).
iv) The goal made by juxtaposing Condition and Conclusion must be safe.

For c2) the condition that must be respected are: i) and iv).
For c3) the condition that must be respected are: i) and iii).
If one of these condition isn't respected, the insertion fails.

User Interaction:

1) If the check insertion window is not mapped:

1.1) press the left button of the mouse;
1.2) select the insert check option in the pannel menu.
After these operations the "check insertion window" is mapped;
2) Write the check in the "check insertion window";
3) Write the "full stop" and press "return" to execute the operation;
4) If you want to unmap the "check insertion window" clic on the white rectangle in the upper lefthand corner of the same window.

3.2.7 Transaction insertion

A transaction must be of a form of one of the following clauses:

i) trans_name ← Prec | T | Postc.

ii) trans_name ← Prec | T .

iii) trans_name ← Prec # T # Postc.

iv) trans_name ← Prec # T #.

v) trans_name ← T.

Prec and Postc are equivalent to queries.
T must be a series of operations on the database, which form an atomic operation.
Two kind of transactions can appear in T:
- standard transactions;
- transactions defined by the user.

Every element in T that isn't a standard transaction is considered a user defined transaction.

User Interaction:
1) If the transaction insertion window is not mapped:
1.1) press the left button of the mouse;
1.2) select the insert transaction option in the pannel menu.
After these operations the "transaction insertion window" is mapped;
2) Write the transaction in the "transaction insertion window";
3) Write the "full stop" and press "return" to execute the operation;
4) If you want to unmap the "transaction insertion window" clic on the white rectangle in the upper lefthand corner of the same window.

3.2.8 Fact deletion

If the deleting fact isn't ground, it must be specified in a ground form, replacing every variable that is present in the term with a constant. The first fact that matches with it, is deleted.

The operation fails if the specified fact doesn't match with a fact already inserted in the database.

User Interaction:

1) If the fact deletion window is not mapped:
   1.1) press the left button of the mouse;
   1.2) select the fact deletion option in the pannel menu.
   After these operations the "fact deletion window" is mapped;
2) Write the fact in the "fact deletion window";
3) Write the "full stop" and press "return" to execute the operation;
4) If you want to unmap the "fact deletion window" clic on the white rectangle in the upper lefthand corner of the same window.

3.2.9 Rule deletion

To delete a rule you must specify its number, a number that you can read in the rule view window. The operation fails if the specified number is wrong.

User Interaction:

1) If the rule deletion window is not mapped:
   1.1) press the left button of the mouse;
   1.2) select the rule deletion option in the pannel menu.
   After these operations the "rule deletion window" is mapped;
2) Write the rule number in the "rule deletion window";
3) Write the "full stop" and press "return" to execute the operation;
4) If you want to unmap the "rule deletion window" clic on the white rectangle in the upper lefthand corner of the same window.

3.2.10 Integrity Constraint deletion

To delete an Integrity Constraint you must specify its number, a number that you can read in the Integrity Constraint view window.
The operation fails if the specified number is wrong.

User Interaction:

1) If the Integrity Constraint deletion window is not mapped:
   1.1) press the left button of the mouse;
   1.2) select the Integrity Constraint deletion option in the pannel menu.

After these operations the "Integrity Constraint deletion window" is mapped;
2) Write the Integrity Constraint number in the "Integrity Constraint deletion window";
3) Write the "full stop" and press "return" to execute the operation;
4) If you want to unmap the "Integrity Constraint deletion window" clic on the white rectangle in the upper lefthand corner of the same window.

3.2.11 Control deletion

To delete a check you must specify its number, a number that you can read in the Integrity Constraint view window.
The operation fails if the specified numbere is wrong.

User Interaction:

1) If the check deletion window is not mapped:
   1.1) press the left button of the mouse;
   1.2) select the check deletion option in the pannel menu.
   After these operations the "check deletion window" is mapped;
2) Write the check number in the "check deletion window";
3) Write the "full stop" and press "return" to execute the operation;
4) If you want to unmap the "check deletion window" clic on the white rectangle in the upper lefthand corner of the same window.

3.2.12 Transaction deletion

To delete a transaction you must specify its number, a number that you can read in the transaction view window.
The operation fails if the specified number is wrong.

User Interaction:

1) If the transaction deletion window is not mapped:
   1.1) press the left button of the mouse;
   1.2) select the transaction deletion option in the pannel menu.
   After these operations the "transaction deletion window" is mapped;
2) Write the transaction number in the "transaction deletion window";
3) Write the "full stop" and press "return" to execute the operation;
4) If you want to unmap the "transaction deletion window" clic on the white rectangle in the upper lefthand corner of the same window.

3.2.13 Fact Modify

When you modify a fact, its arguments can be specified in six ways:

- m(Value): to modify the corrisponding argument, replacing it with Value. Value can be a constant or a function;
- m_add(Number): to modify the corrisponding argument, adding the value of the Number to the precedent. The value and the precedent component must be a number;
- m_sub(Number): to modify the corrisponding argument, subtracting the value of the Number to the precedent. The value and the precedent component must be a number;
- _ : if the Value of the component must remain the same;
- Variable;
- constant.

When variables are present among the arguments of the fact that must be modified, the operation involves the first instance for that predicate.
The ground arguments act as keys for that relation.
The operation fails if:

- the predicate that must be modified isn't defined or is defined with different arity;
- the fact that must be modified isn't derivable;
- the modified fact is already present (if the system is working in time mode);
- the modified fact is already derivable (if the system is working in space mode).

User Interaction:

1) If the fact modify window is not mapped:
   1.1) press the left button of the mouse;
   1.2) select the fact modify option in the pannel menu.
   After these operations the "fact modify window" is mapped;
2) Write the fact with its modification in the "fact modify window";
3) Write the "full stop" and press "return" to execute the operation;
4) If you want to unmap the "fact modify window" clic on the white rectangle in the upper lefthand corner of the same window.

3.2.14 Check the database consistency

With this operation all the checks are tested.
For each check tested the system can answer:

- "check OK";
- "negative check" (if the check of "always onclusion" type or the Condition doesn't have variables);
- "negative check" and the list of wrong values for which is true Condition$\wedge\neg$Conclusion

User Interaction:

1) If the checking window is not mapped:
    1.1) press the left button of the mouse;
    1.2) select the check option in the pannel menu.
   After these operations the "checking window" is mapped;
2) Write the "full stop" and press "return" to execute the operation;
3) If you want to unmap the "checking window" clic on the white rectangle in the upper lefthand corner of the same window.

3.2.15. Disable Integrity Constaints

You can disable an Integrity Constrant without deleting it. The result of this operation has the same effect as the deletion of the Integrity Constraint.
To disable an Integrity Constrant you must specify its number, a number that you can read in the Integrity Constraints view window.

User Interaction:

1) If the disable IC window is not mapped:
    1.1) press the left button of the mouse;
    1.2) select the disable IC option in the pannel menu.
   After these operations the "disable IC window" is mapped;
2) Write the Integrity Constraint number in the "disable IC window";
3) Write the "full stop" and press "return" to execute the operation;
4) If you want to unmap the "disable IC window" clic on the white rectangle in the upper lefthand corner of the same window.

3.2.16 Enable Integrity Constaints

To enable an Integrity Constaint you must specify its number, a number that you can read in the sleep view window.

User Interaction:

1) If the enable IC window is not mapped:
    1.1) press the left button of the mouse;
    1.2) select the enable IC option in the pannel menu.
  After these operations the "enable IC window" is mapped;
2) Write the sleeping Integrity Constraint number in the "enable IC window";
3) Write the "full stop" and press "return" to execute the operation;
4) If you want to unmap the "enable IC window" clic on the white rectangle in the upper lefthand corner of the same window.

3.2.17 Execution of a transaction

The transactions that you can execute are:

- user defined transactions;
- system top level executable transaction (we will see below);
- set_debug transaction (to receive more detailed system messages);
- reset_debug transaction.

User Interaction:

1) If the execute transaction window is not mapped:
    1.1) press the left button of the mouse;
    1.2) select the execute transaction option in the pannel menu.
  After these operations the "execute transaction window" is mapped;
2) Write the transaction goal in the "execute transaction window";
3) Write the "full stop" and press "return" to execute the operation;
4) If you want to unmap the "execute transaction window" clic on the white rectangle in the upper lefthand corner of the same window.

3.2.18 Graphic operations

You can change the view on the graphic frame. After the change, the graphic window will be redrawn according to the new view.

User Interaction:

1) If the graphic operations window is not mapped:
    1.1) press the left button of the mouse;
    1.2) select the graphic operations option in the pannel menu.

After these operations the "graphic operations window" is mapped;
2) Make the changes of the view in the "graphic operations window";
3) Clic the confirm or the annul button;
4) If you want to unmap the "graphic operations window" clic on the white rectangle in the upper lefthand corner of the same window.

3.2.19 Viewing

The system maps a series of windows that allows yow to see the database state.
These windows are:

- Facts:	to view the facts present in the database;
- Rules:	to view the rules present in the database;
- IC:	to view the enabled Integrity Constraints;
- Checks:	to view the checks present in the database;
- Sleeps:	to view the IC that have been disabled;
- Graphic:	to view the contents of the frame.

User Interaction:
1) If the involved window is iconized:
   1.1) clic on it to deiconoze
2) Clic on the scroll bar (if it exists) to scroll the content of the window;
3) If you want to iconify the window clic on the left button in the title bar of the same window.

3.2.20 Quit

With this operation, the session of interaction with GRAPHEDBLOG ends.

User Interaction: To quit:
1) press the left button of the mouse;
2) select the quit option in the pannel menu;
3) when the Prolog prompt appears, type "CTRL-c e" to exit from the GRAPHEDBLOG environment.

Appendix A.

Below we list the standard predicates that can be used in GRAPHEDBLOG's queries and rules. We also explain the way of using them to have safe goals or rules.

A1 +X is -Expression.

What it does:
X is Expression, evalutes Expression as an arithmetic expression and unifies the resulting number to X. Expression may be an expression made up of numbers and/or variables bound to numbers, but may not contain variables bound to expressions. If Expression is not an arithmetic expression, an error message is sent to the standard error stream, and the goal fails.

GRAPHEDBLOG use:
All the variables appearing in Expression must to be used in the predicates that precede the "is" predicate in the goal or in the body of the rule in which it appears.

A2 ?X = ?Y.

What it does:
X = Y unifies X and Y. If "="/2 is not able to unify X and Y, it will simply fail.

GRAPHEDBLOG use:
The use of this predicate at least must to have one of the two parameters instantited with a constant. If it doesn't happen, and the two parameters are instantited with variables, at least one of the two variables must to be used in the predicates that precede the "=" predicate in the goal or in the body of the rule in which it appears.

A3 +X == +Y.

What it does:
X == Y succeeds if the terms currently instantiating X and Y are literally identical..

GRAPHEDBLOG use:
The use of this predicate must to have the two parameters instantited with constants. If it doesn't

happen, and one or all the parameters are instantited with variables, these variables must to be used in the predicates that precede the "==" predicate in the goal or in the body of the rule in which it appears.

A4 +X =:= +Y.

What it does:
X =:= Y evalutes X and Y as arithmetic expressions. The goal succeds if the results are equal. If X and Y are not arithmetic expressions, an error message is sent to the standard error stream, and the goal fails.

GRAPHEDBLOG use:
The use of this predicate must to have the two parameters instantited with ground expressions. If it doesn't happen, all the variables that appear in X and in Y must to be used in the predicates that precede the "=:=" predicate in the goal or in the body of the rule in which it appears.

A5 +X =\= +Y.

What it does:
X =\= Y evalutes X and Y as arithmetic expressions. The goal succeds if the results are not equal. If X and Y are not arithmetic expressions, an error message is sent to the standard error stream, and the goal fails.

GRAPHEDBLOG use:
The use of this predicate must to have the two parameters instantited with ground expressions. If it doesn't happen, all the variables that appear in X and in Y must to be used in the predicates that precede the "=\=" predicate in the goal or in the body of the rule in which it appears.

A6 +X \== +Y.

What it does:
X \== Y succeeds if the terms currently instantiating X and Y are not literally identical..

GRAPHEDBLOG use:
The use of this predicate must to have the two parameters instantited with constants. If it doesn't happen, and one or all the parameters are instantited with variables, these variables must to be used

in the predicates that precede the "==" predicate in the goal or in the body of the rule in which it appears.

A7 +X <u>&lt;</u> +Y.

<u>What it does:</u>
X < Y evalutes X and Y as arithmetic expressions. The goal succeds if the result of evaluating X is strictly less than the result of evaluating Y. If X and Y are not arithmetic expressions, an error message is sent to the standard error stream, and the goal fails.

<u>GRAPHEDBLOG use:</u>
The use of this predicate must to have the two parameters instantited with ground expressions. If it doesn't happen, all the variables that appear in X and in Y must to be used in the predicates that precede the "<" predicate in the goal or in the body of the rule in which it appears.

A8 +X <u>&gt;</u> +Y.

<u>What it does:</u>
X > Y evalutes X and Y as arithmetic expressions. The goal succeds if the result of evaluating X is strictly greater than the result of evaluating Y. If X and Y are not arithmetic expressions, an error message is sent to the standard error stream, and the goal fails.

<u>GRAPHEDBLOG use:</u>
The use of this predicate must to have the two parameters instantited with ground expressions. If it doesn't happen, all the variables that appear in X and in Y must to be used in the predicates that precede the ">" predicate in the goal or in the body of the rule in which it appears.

A9 +X <u>&gt;=</u> +Y.

<u>What it does:</u>
X >= Y evalutes X and Y as arithmetic expressions. The goal succeds if the result of evaluating X is greater than or equal to the result of evaluating Y. If X and Y are not arithmetic expressions, an error message is sent to the standard error stream, and the goal fails.

<u>GRAPHEDBLOG use:</u>
The use of this predicate must to have the two parameters instantited with ground expressions. If it doesn't happen, all the variables that appear in X and in Y must to be used in the predicates that

precede the ">=" predicate in the goal or in the body of the rule in which it appears.

A10 +X =< +Y.

What it does:
X =< Y evalutes X and Y as arithmetic expressions. The goal succeds if the result of evaluating X is less than or equal to the result of evaluating Y. If X and Y are not arithmetic expressions, an error message is sent to the standard error stream, and the goal fails.

GRAPHEDBLOG use:
The use of this predicate must to have the two parameters instantited with ground expressions. If it doesn't happen, all the variables that appear in X and in Y must to be used in the predicates that precede the "=<" predicate in the goal or in the body of the rule in which it appears.

A11 ?Pred =.. ?List.

What it does:
Pred =.. List unifies List with a list whose head is the atom corresponding to the principal functor of Pred and whose tail is a list of the arguments of Pred. If Pred is unistantiated, the List must be instantiated and the invers is made.
If Pred and List are both unistantiated, or if either is not what is expected, "=.."/2 will fail.

GRAPHEDBLOG use:
The use of this predicate at least must to have one of the two parameters instantited with a constant. If it doesn't happen, and all the parameters are instantited with variables, at least one of the two variables must to be used in the predicates that precede the "=.." predicate in the goal or in the body of the rule in which it appears.

A12 dist(+X,+Y,?Z).

What it does:
Dist(X,Y,Z) unifies X and Y with two points (a point is represented by a list of two elements in which the first corresponds to the X coordinate and the second corresponds the Y coordinate) and unifies Z with the distance between them. If X or Y are not bound to points, the predicate fails.

GRAPHEDBLOG use:
The use of this predicate at least must to have the two parameters X and Y instantited with points. If it doesn't happen, and one or the two parameters are instantited with variables, these variables must to be used in the predicates that precede the dist/3 predicate in the goal or in the body of the rule in which it appears.

A13 last_seg(+List,?X,?Y).

What it does:
Last_seg(List,X,Y) unifies List with a list of points and unifies rispectly X and Y with the last two points of the list. If List is not bound to a list of points, the predicate fails.

GRAPHEDBLOG use:
The use of this predicate at least must to have the List parameter instantited with a list of points. If it doesn't happen, and the List parameter instantited with to a variable, it must to be used in the predicates that precede the last_seg/3 predicate in the goal or in the body of the rule in which it appears.

A14 number_of_use(+A,+B,?N).

What it does:
Number_of_use(A,B,N) unifies A and B with two prototypes and unifies X with the number of the use of B in the definition of A. If A or B are not bound to prototypes, the predicate fails.

GRAPHEDBLOG use:
The use of this predicate at least must to have the A and B parameters instantited with prototypes. If it doesn't happen, and the parameters are instantited with variables, these must to be used in the predicates that precede the number_of_use/3 predicate in the goal or in the body of the rule in which it appears.

A15 integer(+X).

What it does:
Integer(X) succeds if X is currently instantiated to an Integer; otherwise it fails.

GRAPHEDBLOG use:

The use of this predicate isn't restrict.

A15 <u>float</u>(+X).

<u>What it does:</u>
Float(X) succeds if X is currently instantiated to a Float; otherwise it fails.

<u>GRAPHEDBLOG use:</u>
The use of this predicate isn't restrict.

A17 <u>base_prot</u>(?X).

<u>What it does:</u>
Base_prot(X) succeds if X is instantiated to a basic prototype; if X is unbound, it unifies X with a basic prototype; otherwise base_prot/1 fails.

<u>GRAPHEDBLOG use:</u>
The use of this predicate isn't restrict.

A18 <u>prototype</u>(?X).

<u>What it does:</u>
Prototype(X) succeds if X is instantiated to a basic prototype or to an used defined prototype; if X is unbound, it unifies X with a prototype; otherwise prototype/1 fails.
<u>GRAPHEDBLOG use:</u>
The use of this predicate isn't restrict.

A19 <u>visualizable_object</u>(?X).

<u>What it does:</u>
visualizablebject(X) succeds if X is instantiated to a visualizable graphic object (see [Di Grande 89]) ; if X is unbound, it unifies X with a visuazable graphic object, if there are; otherwise it fails.

<u>GRAPHEDBLOG use:</u>
The use of this predicate isn't restrict.

A20 <u>obj_on_frame</u>(?X).

<u>What it does:</u>
obj_on_frame(X) succeds if X is instantiated to a graphic object currently putted on the frame; if X is unbound, it unifies X with a graphic object currently putted on the frame, if there are; otherwise it fails.

<u>GRAPHEDBLOG use:</u>
The use of this predicate isn't restrict.

A21 <u>ist_of_prot_on_frame</u>(?X).

<u>What it does:</u>
ist_of_prot_on_frame(X) succeds if X is instantiated to an istance of a prototype currently putted on the frame; if X is unbound, it unifies X with an istance of a prototype currently putted on the frame, if there are; otherwise st_of_prot_on_frame/1 fails.

<u>GRAPHEDBLOG use:</u>
The use of this predicate isn't restrict.

A22 <u>retrievable_instance</u>(+X).

<u>What it does:</u>
retrievable_instance(X) succeds if X is instantiated to an istance of a prototype visualizable; ; otherwise it fails.

<u>GRAPHEDBLOG use:</u>
The use of this predicate must to have the X parameter instantited with an istance of a prototype. If it doesn't happen, and X is instantited with a variable, it must to be used in the predicates that precede the retrievable_instance predicate in the goal or in the body of the rule in which it appears.

A23 <u>compound</u>(?X,?Y).

<u>What it does:</u>
Compound(X,Y) given X, an istance of a prototype, unifies Y with another istance of a prototype that compounds the first. If X is not bound and Y is, the inverse is made; otherwise compound

fails.

GRAPHEDBLOG use:

The use of this predicate at least must to have one of the two parameters instantited with constants. If it doesn't happen, and one or all the parameter are instantited with variables, at least one ot these must to be used in the predicates that precede the compound/2 predicate in the goal or in the body of the rule in which it appears.

A24 depend(?X,?Y).

What it does:

Depend(X,Y) given a prototype X, unifies Y with another prototype from which X depends. If X is not bound and Y is, the inverse is maded; otherwise compound fails.

GRAPHEDBLOG use:

The use of this predicate at least must to have one of the two parameters instantited with constants. If it doesn't happen, and one or all the parameter are instantited with variables, at least one ot these must to be used in the predicates that precede the depend/2 predicate in the goal or in the body of the rule in which it appears.

A25 last_in_list(?X,?Y).

What it does:

Last_in_list(X,Y) unifies X with a list and unifies Y with the last element of the list. If List is not bound and X is, unifies X with a list with only one element thay is Y; otherwise last_in_list fails.

GRAPHEDBLOG use:

The use of this predicate at least must to have one of the two parameters instantited with constants. If it doesn't happen, and one or all the parameter are instantited with variables, at least one ot these must to be used in the predicates that precede the last_in_list/2 predicate in the goal or in the body of the rule in which it appears.

Appendix B.

GRAPHEDBLOG offers to the users a series of predefined transactions.
With same of these you can modify the sets of facts and rules that are present in the currently managed database. With athers you can do graphic operations on the frame.
With athers again you can perform other operations.
You can execute directly only a subset of the predefined transactions, but you can utilize all of these in the body of the user defined transactions.

Below we list the predefined transactions of GRAPHEDBLOG.
We also explain the way of using them .

B1 insert_f(Mode, Fact).

What it does:
With this transaction you can specify the insertion of a fact in the database.

The Mode component can be bound to the following values:
- t (time) : the insertion is maded in time mode;
- s (space): the insertion is made in space mode.

If Fact respect the conditions supposed in 3.2.3, the result of the insertion can be:
- the fact is inserted;
- the fact is already present in the database, the Mode is t: the transaction doesn't have effect but doesn't fail.
- the fact is already derivable in the database, the Mode is s: the transaction doesn't have effect but doesn't fail.

GRAPHEDBLOG use:
This transaction can not be directly executed, it can only be used in the body of the user defined transactions.

B2 insert_r(Mode, Rule).

What it does:
With this transaction you can specify the insertion of a rule in the database.

The Mode component can be bound to the following values:

- t (time) : the insertion is maded in time mode;
- s (space): the insertion is made in space mode.

If Rule respect the conditions supposed in 3.2.4, the result of the insertion can be:

- the rule is inserted;
- the rule is inserted and all the explicit facts that can be derived by the rule are deleted, if Mode is bound to the constant s.

GRAPHEDBLOG use:
This transaction can not be directly executed, it can only be used in the body of the user defined transactions.

B3 delete_f(Fact, Mode).

What it does:
With this transaction you can specify the deletion of a fact in the database.
The Mode component can be bound to the following values:

- 1: the deletion of the fact is phisically maded but if there are rules such that the fact can still derived, nothing is maded;
- 2: the deletion of the fact is phisically maded but if there are rules such that the fact can still derived, these are deleted;
- 3: the deletion of the fact is phisically maded but if there are rules such that the fact can still derived, a IC is inserted to impede the derivation of the fact.

If Fact respect the conditions supposed in 3.2.8, the result of the deletion can be:

- the fact is deleted;
- the fact not exist and the transaction doesn't have effect, but doesn't fail.

GRAPHEDBLOG use:
This transaction can not be directly executed, it can only be used in the body of the user defined transactions.

B4 delete_r(Index).

What it does:
With this transaction you can specify the deletion of a rule in the database.
If Index (e.g. the rule number) respect the conditions supposed in 3.2.9, the result of the deletion

can be:

- the rule is deleted.

GRAPHEDBLOG use:
This transaction can not be directly executed, it can only be used in the body of the user defined transactions.

B5 modify(Mode,Fact).

What it does:
With this transaction you can specify the modification of a fact in the database.
The Mode component can be bound to the following values:

- t (time) : the modification is maded in time mode;
- s (space): the modification is made in space mode.

If Fact is specied as supposed and required in 3.2.13 , the result of the modification can be:

- the fact is modified;
- the fact modified is already present in the database, the Mode is t: the transaction delete only the modified fact.
- the fact modified is already derivable in the database, the Mode is s: the transaction delete only the modified fact.

GRAPHEDBLOG use:
This transaction can not be directly executed, it can only be used in the body of the user defined transactions.

B6 forall(Vars_list,Goal,Trans).

What it does:
This transaction drops the Goal in the theory that represent the database and collects all the finded solutions for the variables that are present in Vars_list.
For each solution, the transaction Trans is executed.
This transaction fails if:

- there aren't solutions for Goal;
- the transaction Trans doesn't exist;
- all the executions of Trans fail.

GRAPHEDBLOG use:
This transaction can be directly executed, it can also be used in the body of the user defined

transactions.

B7 iterative(Trans).

What it does:
This transaction execute Trans for all the instances of the free variables in the head of Trans when this is used.
Iterative(Trans) can be compiled in

forall(Free_vars_in_Trans,(P_1;..;P_n),Trans)

where in the list Free_vars_in_Trans are present the free variables that are present in the use of Trans and (P_1;..P_n) is a goal maded of the disgiuntion of all the preconditions that are present in the definitions of Trans.
This transaction fails if:

- Trans not exist;
- all the definition of Trans doesn't have preconditions;
- all the excution of Trans fail.

GRAPHEDBLOG use:
This transaction can be directly executed, it can also be used in the body of the user defined transactions.

B8 on_screen(Graph_object).

What it does:
This transaction put on the frame the graphic object Graph_object. It fails if Graph_object isn't a correct graphic object.
GRAPHEDBLOG use:
This transaction can be directly executed, it can also be used in the body of the user defined transactions.

B9 off_screen(Graph_object).

What it does:
This transaction put off the frame the graphic object Graph_object. It fails if Graph_object currently isn't on the frame.

GRAPHEDBLOG use:
This transaction can be directly executed, it can also be used in the body of the user defined transactions.

B11 clear_screen.

What it does:
This transaction put off the frame all the graphic object currently on it. It never fails.

GRAPHEDBLOG use:
This transaction can be directly executed, it can also be used in the body of the user defined transactions.

B10 rewrite_screen.

What it does:
This transaction acts as a revision operation. It clear the frame and puts again on it the graphic objects that previously were on it.
All the modifications of the description in the database of theese graphic objects are thus rendered visible.

GRAPHEDBLOG use:
This transaction can be directly executed, it can also be used in the body of the user defined transactions.

B11 fail.

What it does:
This transaction has the same meaning as in Prolog. When it is executed a failure happens.

GRAPHEDBLOG use:
This transaction can not be directly executed, it can only be used in the body of the user defined transactions.

B12 true.

<u>What it does:</u>
This transaction has the same meaning as in Prolog. When it is executed a success happens.

<u>GRAPHEDBLOG use:</u>
This transaction can not be directly executed, it can only be used in the body of the user defined transactions.

B13 X is Expression.

<u>What it does:</u>
This transaction has the same meaning as in Prolog. When it is executed the value of Expression is bound to X. This fails if Expression is not an arithmetic expression..

<u>GRAPHEDBLOG use:</u>
This transaction can not be directly executed, it can only be used in the body of the user defined transactions.

Bibliography

[Agnello 87] A. Agnello, **Un Linguaggio di Interfaccia Grafico per il Sistema di Gestione di Basi di Dati Logiche EDBLOG**, Tesi di Laurea, Dip. di Informatica, Università di Pisa, Aprile 1988.

[Assirelli & al. 85b] P. Asirelli, M. De Santis, M. Martelli, **Integrity Constraints in Logic Data Bases**, Journal of Logic Programming, Vol. 2, No. 3, Ottobre 1985.

[Barbuti & al. 86] R. Barbuti, M. Martelli, **Completeness of the SLDNF Resolution for a Class of Logic Programs**, Proc. of the 3rd Int. Conf. on Logic Programming, London, 1986.

[Bowen & al. 82] K. A. Bowen, R. A. Kowalski , **Amalgamating Language and Metalanguage in Logic Programming**, Logic Programming, Academic Press, Londra 1982, pp. 159 - 172,

[Bowen 85] K. A. Bowen, **Meta - Level Programming and Knowledge Representation**, Technical Report, CIS - 85 - 1, School of Computer & Information Science, Syracuse University, Agosto 1985.

[Di Grande 89] D. Di Grande, **Un modello per la rappresentazione degli oggetti grafici basato su un DB logico**, Tesi di Laurea, Dip. di Informatica, Università di Pisa, Aprile 1989.

[De Santis 85] M. De Santis, **Logic Programming e Database: un ambiente di sviluppo adatto al trattamento dei vincoli di integrità**, Tesi di Laurea, Dip. di Informatica, Università di Pisa, Gennaio 1985.

[Gallaire & al. 78] H. Gallaire, J. Minker, J. M. Nicolas, **Logic and Databases**, Plenum Publishing Co., New York, N. Y., 1978.

[Gallaire 83] H. Gallaire, **Logic Databases vs. Deductive Databases**, Logic Programming Workshop, Albufeira, Portogallo 1983, pp. 608 - 622.

[Gallaire & al. 84] H. Gallaire, J. Minker, J. M. Nicolas, **Logic and Databases: a Deductive Approach**, Computing Surveys, Vol.16, No.2, 1984, pp. 153 - 185.

[Giannini & al. 86] F. Giannini E. Grifoni, **Programmazione Logica in Ambiente di Sviluppo Software: Data Base Logici come Data Base di Progetto**, Tesi di Laurea, Dip. di Informatica, Università di Pisa, Ottobre 1986.

[Green 69] C. Green, **Theorem proving by resolution as a basis for question-answering systems**, Machine Intelligence 4, B. Meltzer, D. Michie Edd., American Elsevier Pub. Co. , New York, n. Y. , 1969.

[Helm & al.86] R. Helm, K. Marriot, **Declarative Graphics**, Lecture Notes in Computer Science No. 225, Springer - Verlag, Londra, Luglio 1986, pp. 513 - 527.

[Hubbold & al. 88] R. J. Hubbold, W. T. Hewitt, **GKS-3D and PHIGS Theory and Practice**, EUROGRAPHICS'88, Tutorial Cours No. 1, settembre 1988.

[Julien 82] S. M. P. Julien, **Graphical in Micro-Prolog**, Research report DOC