Consiglio Nazionale delle Ricerche

ISTITUTO DI ELABORAZIONE
DELLA INFORMAZIONE

PISA

Gedblog Reference Manual

Revised Version

P. Asirelli, P. Inverardi,

D. Aquilino, D. Apuzzo,

G. Bottone, M.C. Rossi

Nota Interna B4-18

Aprile 1995

Gedblog

Reference Manual

P. Asirelli, P. Inverardi, D. Aquilino,
D. Apuzzo, G. Bottone, M.C. Rossi.

Indice

Chapter 1

Introduction

In the paper we describe GEDBLOG, a database system for the design, validation and execution of graphical applications.
The main feature of the system is its declarativity which allows users to develop their own applications in a compositional and consistent (with respect to the assessed requirements) fashion.
This permits the graphical representation of concepts and their declarative semantics to be combined within a uniform, logic framework.
Another important feature of the system is the provision of an integrity constraints checking mechanism which permits properties of the application under development to be proved.
A graphic application is designed through a step by step refinement of the knowledge base, i.e. the application is developed consistently with respect to the assessed integrity constraints that can be considered as those requirements that the application must satisfy.
The knowledge concerning the application can be defined by means of:

- facts and rules to express the general knowledge;
- integrity constraints formulas to express exceptions to the general knowledge, or as general properties or requirements of the application being developed;

The capability that permits a graphic representation to be associated with a concept means that the overall semantics can be affected by constraints imposed only on the graphic representation and, vice versa, the graphic representation can be affected by semantics.
The relationship between the semantics and the graphic denotation of concepts within the same context make the tool particularly suitable for developing and prototyping applications in the areas of visual language, graphic interfaces and some aspects of CAD.

At the end of this section, we give a brief example to allow the reader to better grasp the ideas behind GEDBLOG and the approach to prototyping that it encourages and supports.

GEDBLOG is the result of the integration of a deductive (logic) database with graphics.

Our aim is to use a logic database approach to the problem of prototyping applications.

To justify this strategy, our basic assumption is that requirements can easily be expressed as integrity constraints and the development of the application can be seen as an incremental knowledge base updating process.

Although the use of a logic language for expressing databases, queries and integrity constraints is quite natural since logic provides a formal specification for the resulting application, deductive capabilities that allow a straightforward definition of a knowledge base and a formal base for the definition and checking of integrity constraints, the representiation of graphic objects presents some problems and cannot simply be handled by interfacing the logic database system with a graphic package.

Graphic objects, their relations, and the set of operations that the user can perform, represent the model of the real world implemented by the application.

Our strategy is to use a logic language for the definition of graphic objects and their operations [3, 4, 13, 17, 21, 22]. Since the representation is rigorously declarative, the graphic shape of an object is fully determined by the values of its attributes (such as colour and position); there is no notion of global attributes.

This is fundamental in our approach as we want to amalgamate graphic and non graphic components n the same declarative context.

GEDBLOG permits the definition of objects where the inclusion of other objects is controlled by a condition (an atomic formula).

This is called Conditional inclusion and the effect is the definition of graphic objects that can have different structures at visualization time depending on the value of the condition, i.e.depending on the state of the database.

A concept of "frame" and of "frame state" have also been introduced in order to be able to define the visualization operation and to be able to perform integrity constraint checking on graphic objects at visualization time.

In the following, we introduce a (data) model, and the corresponding linguistic constructs to manage graphic objects.

The model has been defined in terms of the Logic Data Base Management System EDBLOG in order to obtain **GEDBLOG** .

Chapter 2

Logic and Databases

2.1 Logic DataBases

There is quite a lot bibliography on the subject of deductive databases.
Some of the basic definitions we refer to can be found in [13,14,15,37,46].
In [12] 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 EntityRelationship 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 [21], 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.
That is 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:

- knowledge representation;
- knowledge acquisition;
- use of knowledge.

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

2.1.1 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.
In the last few years there has been a growing interest in deductive databases, i.e. databases where "deduction" is the basic mechanisms to get information from the database.
The deductive engine and language are used to define and run the query to the underneath database, thus the query is considered as a logic program and the engine to interprete it can be "top-down" (based on Resolution), "bottom-up" (such LDL- based on the T_P operator)[60] or "mixed" [59].
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.

2.1.2 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.

2.1.3 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 .
These restrictions can be partially relaxed, at least with respect to negation and to certain kinds of queries [8].

In [3] 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 [3] and in [12].
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.

2.1.4 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.

2.1.5 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.

Chapter 3

The basic idea of the system

Here we want to give a grasp of the system, before getting into details. Thus we first introduce the formal language of definite clasues on which the system is based, then we show a brief example problem.

3.1 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 [21] 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 facts
- $A \leftarrow B_1\&...\&B_n$ rules

where A,$B_1, ..., B_n$ are literals.
A is the consequent,$B_1, ..., B_n$ are the premises and they look like $p(t_1, ..., t_n)$ where:
p is a predicate symbol and $t_1, ..., t_n$ are terms.
The informal interpretation of a clause $A \leftarrow B_1\&...\&B_n$ is that:
" A holds if $B_1, ..., B_n$ 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$f(t_1, ..., t_k)$ where f is a data constructor symbol (functor) and $t_1, ..., t_k$ 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:
[21]and[26].

3.2 A little example problem

In the following we present a simple example to illustrate the use of **GEDBLOG**.
Let us define the knowledge about Jan being the father of Ann and Charles, of Ann being a girl and Charles being a boy which is expressed as follows:

- father (jan, ann).
- father (jan, charles).
- girl (ann).
- boy (charles).

We then want to express a general law concerning a daughter being a girl and a son being a boy:

- daughter(X)← girl(X), father(Y,X).
- son(X)← boy(X), father(Y,X).

The next step is to express a knowledge of requirements such as :
if X is a daughter she cannot be a son;
if X is a boy he cannot be a girl; nobody can have two different fathers:
if X is father of Y and if Z is father of Y then it must be the case that X and Z must be the same.

- daughter (X), son (X) → false.
- boy (X), girl (X) → false.
- father(X,Y), father (Z,Y) → X==Z.

The next step is to add a graphic denotation for all elements and relations in the knowledge base.
Thus we define (this is just a sketched definition, not an exact one):

- prototype (girl, "graphic definition of a girl").
- prototype (boy,"graphic definition of a boy").
- object (ann, girl).
- object(charles,boy).

We can now represent the link for the father relation:

- prototype(father_link (X,Y), "a line that connects X and Y").
- object(jan_ann_link, father_link(jan, ann)).
- object(jan_charles_link, father_link(jan, charles)).

Then we can put constraints that bind the graphic representation to the semantics, i.e. constrain the graphic representation of boys, girls and father_link to the existence in the base of the corrisponding knowledge:

- object(X,girl) → girl(X).
- object(X,boy) →boy(X).
- object(X,father_link(Z,Y)) → father(Z,Y).

GEDBLOG allows us to build this knowledge base, proving the constraints when it is updated and also visualizing objects on the screen when such a visualization is possible, i.e. no violation is detected.

Chapter 4

The Logic Database Management System (LDBMS) of GEDBLOG

The logic database management system **GEDBLOG** [11] is a system which has the capability to manage databases which contain graphical **GEDBLOG** and non graphical information.
has been defined as an extension of **EDBLOG** [27] by introducing the capability of manage graphical information.
EDBLOG, in its turn, has been defined as an extention of **DBLOG** [12] by introducing transaction definitions and handling facilities.
DBLOG is the kernel of the **LDBMS**.
GEDBLOG considers the data base system as consisting of four parts:

- a logic program in which:
  1. the *set of facts*, "unit" Horn clauses, are considered to be the Extensional component of the DB (EDB);
  2. the *set of deductive rules*, "definite" Horn clauses, are considered to be the Intensional component of the DB (IDB);
- a set of integrity constraint formulas with:
  1. a *set of Integrity Constraints* (IC), which are formulas of the form:
     $A_k \rightarrow B_1 \& ... \& B_s$
     which can be interpreted informally as:
     " whenever A_k is true then $B_1 and ... and B_s$ must also be true";
  2. a set of Controls formulas which are either formulas as in 1 or else

(a) $A_1\&...\&A_m \rightarrow B_1\&...\&B_n$

(b) $\rightarrow B_1\&...\&B_n$

(c) $A_1\&...\&A_m \rightarrow$

The informal interpretation for (a) is that:
" whenever $A_1\&...\&A_m$ are true then $B_1\&...\&B_n$ must also be true".

The informal interpretation for (b) is that:
"$B_1\&...\&B_n$ must be true"

The informal interpretation for (c) is that:
" $A_1\&...\&A_m$ must be false".

Note that for formulas (a_c), as well as for the formula 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 .

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

1. $trans_i \leftarrow prec|trans_1, ..., trans_n|post;$

   The language used to express this kind of transaction syntactically resembles Concurrent Prolog with no annotated variables [33].
   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.

2. $trans_i \leftarrow prec\#trans_1, ..., trans_n\#post;$
   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 [9], [10] 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.

4.1 More about integrity constraints and transactions

The two forms of integrity constraints correspond to two different integrity checking algorithms.
The first (*The Modified Program Method*) considers a subset of the given logic formulas, called *IC - Integrity Constraints*, and uses them

to modify the logic program automatically so that all facts which do not satisfy IC are not provable/derivable from the modified logic program/DB (i.e. illegal queries cannot succeed).

The second (*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 [2].

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 and/or IC) from those related to particular operations, 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-query 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.

The structure of **EDBLOG** can be depicted as in Figure 1, i.e. a logic theory TKB, the knowledge base, and a meta-theory TKBMS, its management system.

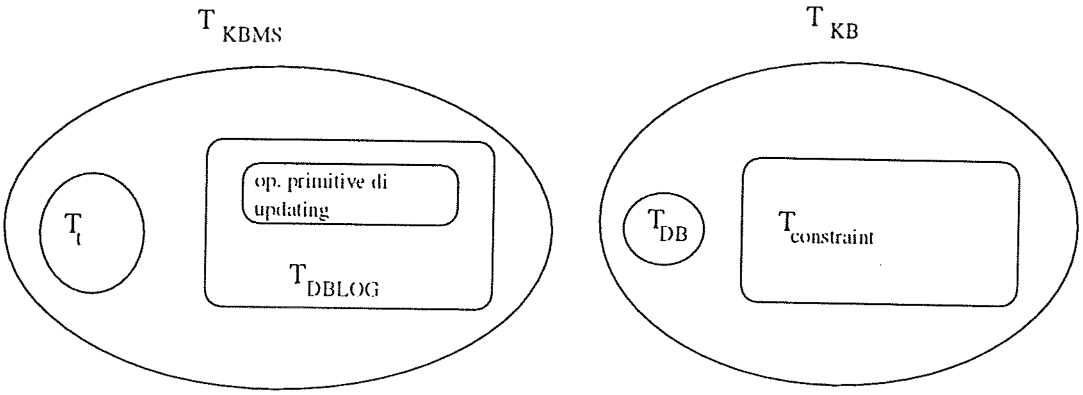


fig. 1

In order to obtain the **GEDBLOG** system, the above structure has been modified as in Figure 2:

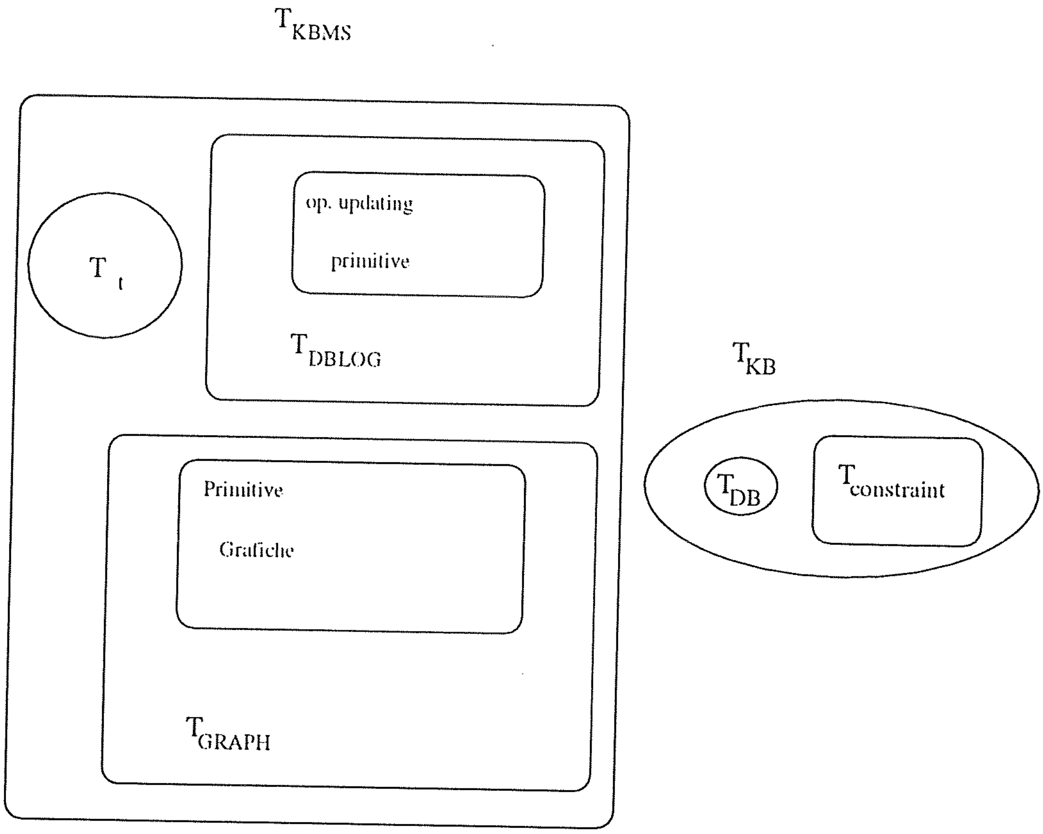
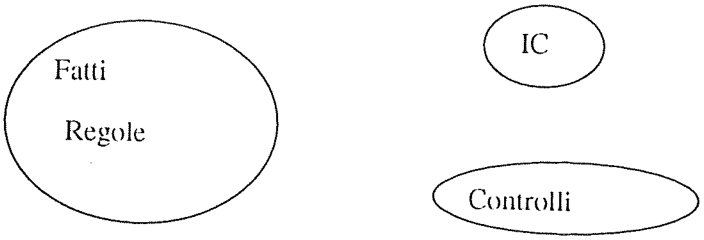


fig.2

That is, all information that is set up by the user, such as the representation of user defined prototypes and of graphic 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 graphic objects, will be resident on the theory T_{KBMS}.

4.1.1 The Logic Database Kernel

The elements described in the first two points form the basic components of the kernel (**DBLOG**), and can be depicted as in fig. 3.



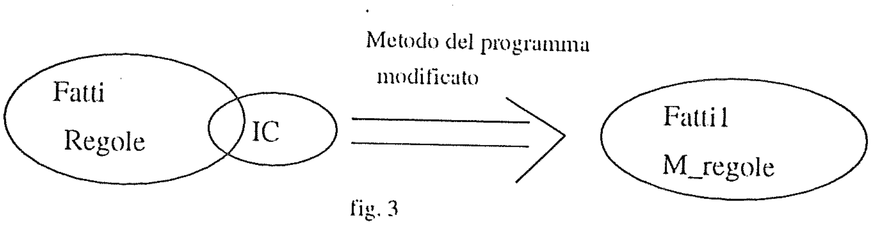


fig. 3

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.4, 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.
For example:

Facts

age(david,20).
age(mary,22).
employee(david).
employee(mary).

IC

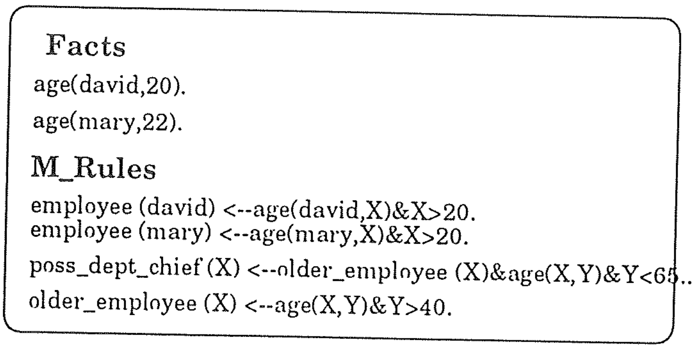
employee (Y) ==>age(Y,X)&X>20.
poss_dept_chief (X) ==> age(X<Y)&Y<65.

Rules

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

fig4.

Then the resulting database to be considered, after running the algorithm for the modified program method, is described in Fig. 5:



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.
```

fig5.

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:

$a(X, Y, Z) \rightarrow b_1(X), b_2(X, Y), b_3(X, Y, Z)$ 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:

$\leftarrow b_1(c), b_2(c, d), b_3(c, d, f)$ and $\leftarrow b_1(a), b_2(a, r), b_3(a, r, s)$. 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.

Chapter 5

The graphic model

In this section we introduce the concepts our model is based on.
The model consists of the following elements:

- *prototypes*, i.e. templates of graphic objects;
- *mechanisms to compose* prototypes;
- *graphic objects*, i.e. instances of prototypes;
- *mechanisms to operate* on graphic objects;
- *constraints to define* the semantics of prototypes and graphic objects.

5.1 Prototypes

The prototype concept naturally emerges when different views of the same object are considered useful and, furthermore, when it is desirable to use a given graphic 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 graphic objects upon instantiation.
The information with respect to which all prototype descriptions are parametric is represented by means of the notion of attribute.
Therefore, attributes represent features of a graphic object.
In our model two classes of attributes are provided:

- contextual attributes:

    1. geometric attributes (origin, scale, rotation);
    2. state attributes (raster function, fill patterns etc.);

- absolute attributes: user definable attributes.

Contextual attributes represent the usual topological information which is connected with a graphic object.
The absolute attributes, instead, permit the definition of partial description of graphic objects where 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.
A *prototype* is the description of a graphic object which is parametric with respect to contextual attributes.
A *parametric prototype* is a prototype which is also parametric with respect to some absolute attributes, i.e. it acts as a template.
The set of attributes of a prototype definition constitutes its interface.
Prototypes are divided into two further classes:

- *basic primitive prototypes*, that represent the usual graphic 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.

As we have previously pointed out, the description of a prototype defines the graphic structure of the object to be represented parametrically, with respect to some attributes.
The range over which these attributes can assume values can be limited by introducing the concept of **property** which serves to type the attributes.
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.
Given a prototype **P**, parametric with respect to an attribute name,

the set of values associated to it by all properties (tuples) determines the range for name wrtP.
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].

5.2 Compound prototypes

When dealing with compound objects, i.e. a graphic object O_1 is a structural element of a more complex object O_2 , the description of the prototype $P_2 of O_2$, must declare the use of the prototype $P_1 of O_1$.
This is achieved by means of the notion of use declaration:
" 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 on 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 on 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.
A prototype P_1 depends on a prototype P_2 if the description of P_1 contains a use declaration of P_2.
The relation depends is transitive.
For any given prototype P its dependencies can be represented as a graph, the dependency graph, where the root denotes P and the other nodes denote all prototypes P depends on.
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.
Therefore, a prototype is definable if its dependency graph is acyclic.
With this definition, we want to make it clear that the only prototypes that can be described in our model are hierarchical ones.

5.2.1 Conditional use declaration

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

In such 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, we want to be able to model cases in which some piece of 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 graphic object is no longer static and fixed at definition time.

For this reason we introduce the concept of **conditional inclusion.**

A guard is either a query to the database or a boolean expression.

A prototype P_2 conditionally includes a prototype P_1, if the use declaration of P_1 contains a guard.

Conditional inclusion makes it possible to define graphic objects that can have a different structure at visualization time, depending on the value of the guard, i.e depending on the state of the database.

5.2.2 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 to *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 6 gives an example of a tree representation for a prototype.

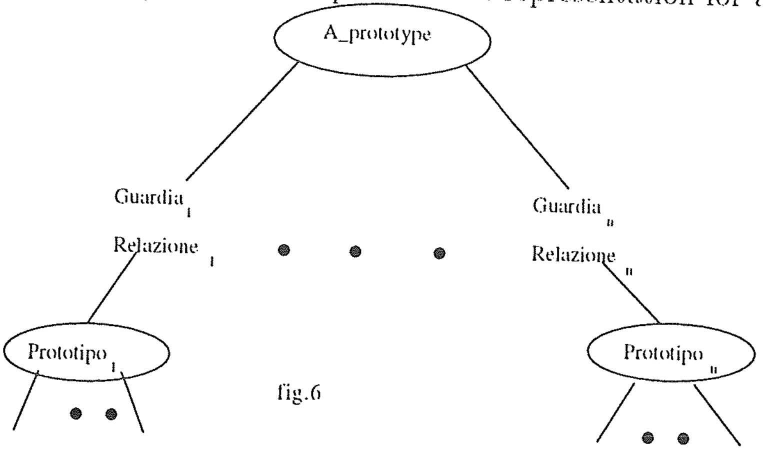


fig.6

5.3 Graphic objects

A graphic object is obtained by fully instantiating a prototype by means of a create operation which provides the information needed by the prototype description, thus generating a **ground** (no variables are left uninstantiated) instance of the prototype definition.
The only operation that can be performed on graphic objects is the visualize operation which depicts the object on the *(virtual) screen.*
We model the screen by means of the frame concept, in terms of which the semantics of the visualize operation is defined.
The frame represents the abstract plane on which graphic objects will be drawn.
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.
Then the semantics of the visualize operation consists of changing the current state of the frame by adding the name and the description of the graphic object to be visualized.
The concepts of frame and of its state have been introduced in order to be able to define the visualize operation and to be able to perform integrity constraints checking on graphic objects at visualization time.

Like prototypes, graphic objects can be represented by trees.
In this case, the root denotes the graphic 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 6 above, its instance, the graphic object Example, is represented by the tree of Figure 7.

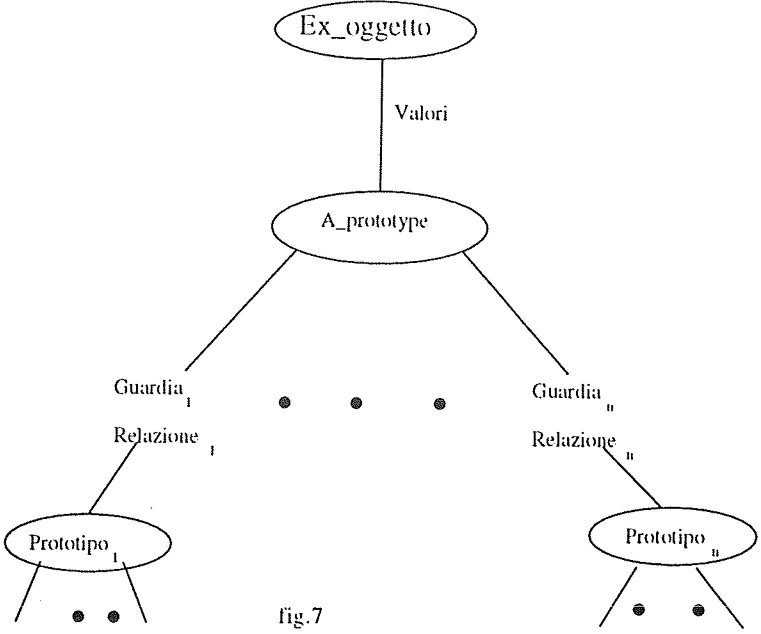


fig.7

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 correct range is performed.
An object O, instance of a (correct wrt attributes) prototype P, is correct if, for all its current attributes, the corresponding properties are satisfied.
An object cannot be visualized if its structure is not correct, i.e. if it is not correct the use it makes of sub-objects.

5.4 Integrity Constraints

In this section we introduce constraints as a means to define invariant properties of the objects under definition.

An integrity constraint formula is a closed first order formula in prenex conjunctive normal form, where the predicate symbols occurring in the formula denote the basic elements of our model, i.e. prototypes, graphic objects, frame, state of the frame, etc; and the elements of the graphic application under development.
Note that, thanks to the notion of frame, we are able to define constraints which will became active at visualization time.
Integrity constraints represent invariant properties of the application, thus changes to the application can only be made if the stated constraints are satisfied.

5.5 Syntax for prototype

- $prototype(Proto\_name(ParForm_1, ..., ParForm_n), Proto)$.
- Proto ::= Base_proto(Resources, Operations, Links).
- Resources::= [(resource_name, Val), ...].
- Operations ::= [Goal & ... & Goal].
- Links ::= $[(link(link\_name, Proto\_name(Par\_Act_1, ..., Par\_Act_n)), |link(link\_name, Proto)), ...]$.

Base_Proto is a basic type graphic; **Proto_name** is the name of user's prototype defined.
Val defined the value of a resource of a basic type.

Val::= $file(File\_name)|pixmap(Pixmap\_name)|string(String\_value)$ $|callback(Transaction\_name)|Numeric\_value$.

5.6 Syntax for object

object(Object_name,Base_proto).
Define a graphical object *Object_name* from basic prototype *Base_proto.*

$object(Object\_name, Proto_name(Par\_Act_1, ..., Par\_Act_n))$.
Define a graphical object from user's prototype *Proto_name.*

link(Child_name,Object,Base_proto).
Make subobject (child) of name *Child_name* and type *base Base_proto.*

$link(Child\_name, Object, Proto\_name(Par\_Act_1, ..., Par\_Act_n))$.
Make subobject (child) of name $Child\_name$ and type $Proto\_name$ definito dall'utente.

Example:
prototype(labelled_window, formDialog([],[], [link(button,pushButton([],[],[]))])).
object(demowin,labelled_window).

5.7 Resources Value

L'aspetto di un oggetto grafico e' caratterizzato, nell'ambito del tipo grafico scelto per rappresentarlo, dall'insieme di valori delle sue risorse.
Come meccanismo di accesso a questi valori , Gedblog mette a disposizione un predicato predefinito:
$get\_par(Object\_name, [res\_name_1, Var_1, ..., res\_name_n, Var_n])$, che istanzia le variabili $Var_1, ..., Var_n$ ai valori che le risorse
$res\_name_1, ..., res\_name_n$ hanno nell'oggetto grafico di nome Object_name.
E' fornita anche una transazione predefinita :
$set\_par(Object\_name, [res\_name_1, Val_1, ..., res\_name_n, Val_n])$ che setta le risorse$res_name_1, ..., res_name_n$ nell'oggetto grafico di nome Object_name ai valori$Val_1, ..., Val_n$.
La semantica di questa transazione e' equivalente a quella di una operazione di cancellazione dell'oggetto ed una successiva operazione di inserzione con i valori delle risorse stabiliti.

5.8 Accessing Sub-Components

Dato che un oggetto grafico puo' essere strutturato internamente in componenti (figli) sia tramite i links nella definizione del suo prototipo che tramite link s esterni, e' necessario prevedere meccanismi per accedere alle sue parti (necessari ad esempio se si vuole variare o testare i valori delle risorse di un sotto-oggetto).
L'accesso alle sottoparti di un oggetto avviene tramite l'operatore '!', che permette di costruire cammini di accesso alle parti di un oggetto strutturato.

Example:
prototype(labelled_window, formDialog([],[],[link(button,pushButton([],[],[])),

```
link(counter,label([(labelstring,string(0))],[],[]))])).
object(demowin,labelled_window).
```

In questo caso e' stato definito un prototipo labelled_window di tipo formDialog e con due sottoprototipi, button e counter, tutti di tipo primitivo (rispettivamente pushButton e label).
L'oggetto di nome demowin di tipo labelled_window avra' quindi due sottoparti (figli) accessibili come demowin!button e demowin!counter rispettivamente. Ad esempio, per istanziare la variabile Count al valore delle risorsa labelstring della label counter e' sufficiente scrivere un goal del tipo:

```
get_par(demowin!counter,[labelstring,string(Count)])
```

5.9 Calling Transactions as Objects Callback

Un oggetto grafico visualizzato e' sensibile ad eventi di input (da mouse, tastiera od altre periferiche).
Il meccanismo utilizzato in Gedblog per catturare gli eventi di input e' quello dei widgets, e si basa sull'attivazione automatica di una procedura (callback) destinata a gestire il tipo di evento che si e" verificato sull'oggetto che lo ha ricevuto.
Settando le risorse callback degli oggetti grafici a nomi di transazioni, si ottiene la chiamata automatica di una transazione in risposta ad un evento di input.

Per gestire l'evento di pressione del bottone nell'esempio precedente definiamo una nuova versione del prototipo labelled_window:

```
Example
prototype(labelled_window,formDialog([],[],[
link(button,pushButton([(activateCallback,callback(setlabel))],[],[])),
link(counter,label([(labelstring,string(0))],[],[]))])).
object(demowin,labelled_window).
```

In questo modo la transazione setlabel gestira' l'evento di attivazione del bottone demowin! button. La transazione setlabel puo' essere ad esempio la seguente:

setlabel := get_par(demowin ! counter,[labelstring,string(L)])# NewL is L+1, set_par(demowin!counter,[labelstring,string(NewL)])# true.

In questo modo, ad ogni attivazione (evento di press-and-release) del bottone, il valore della label demowin!counter sara' incrementato di una unita'.

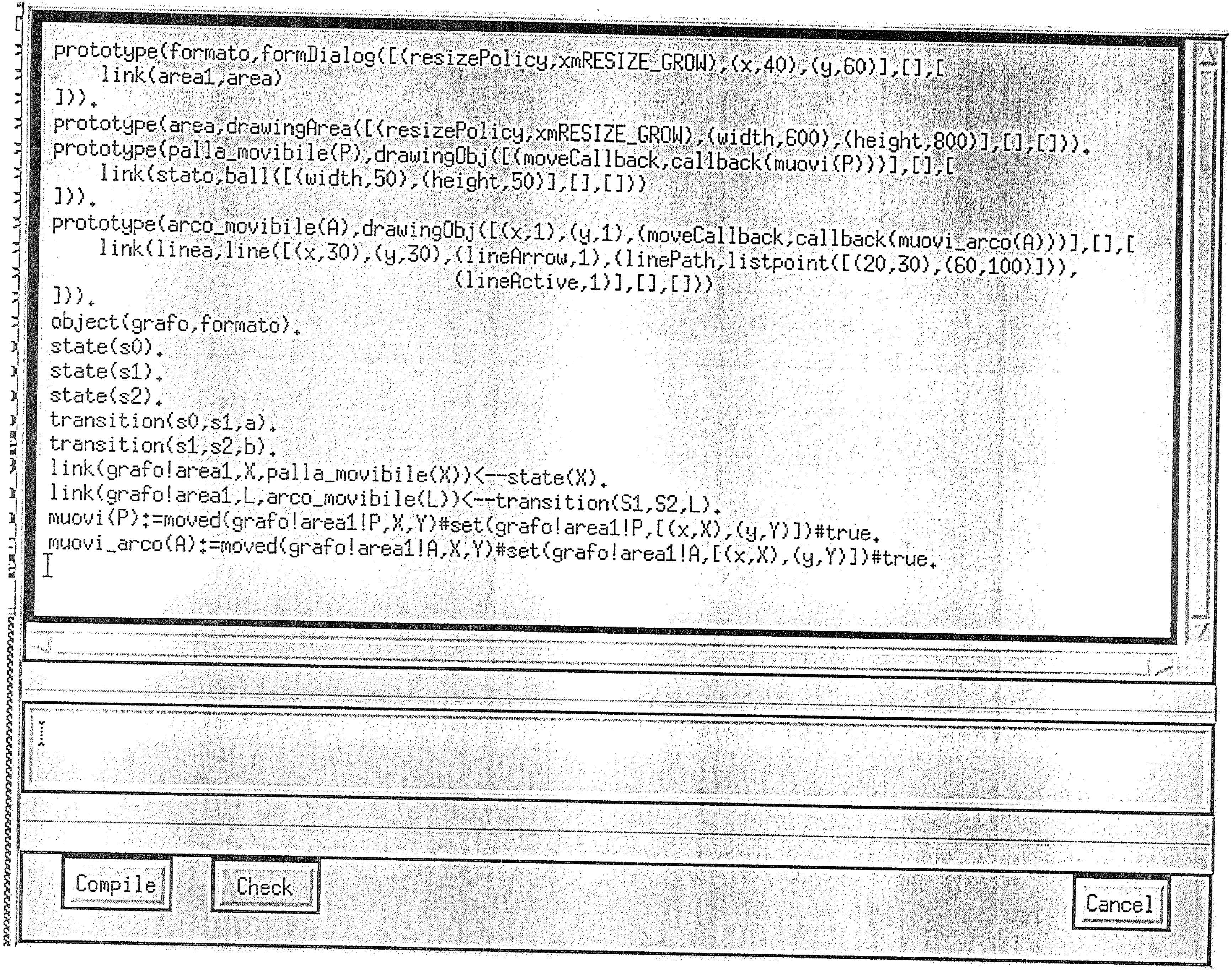
5.10 Getting what's happened after an event

Notiamo che un evento porta esclusivamente l'informazione che una qualche azione e' avvenuta sull'oggetto che lo ha catturato.
Per rilevare eventuali modifiche che l'evento ha prodotto, GEDBLOG mette a disposizione dei predicati che permettono di gestire la variazione di certi valori di risorse degli oggetti a seguito di eventi particolari .

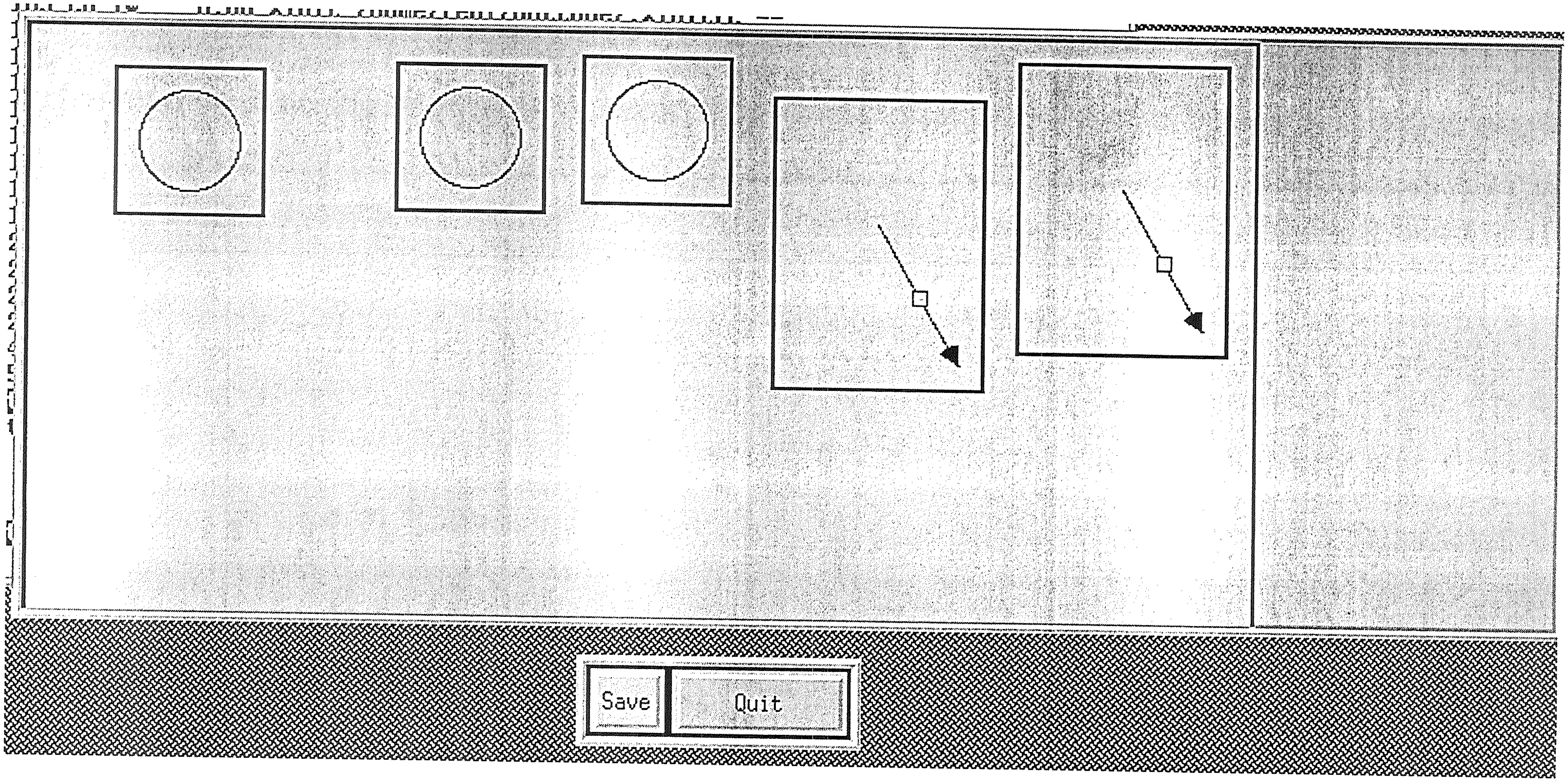
Questi sono:
moved(Obj_name, (X,Y)).
Ha successo se l'oggetto Obj_name e' stato spostato ed istanzia le variabili X ed Y alle coordinate della nuova posizione dell'oggetto .

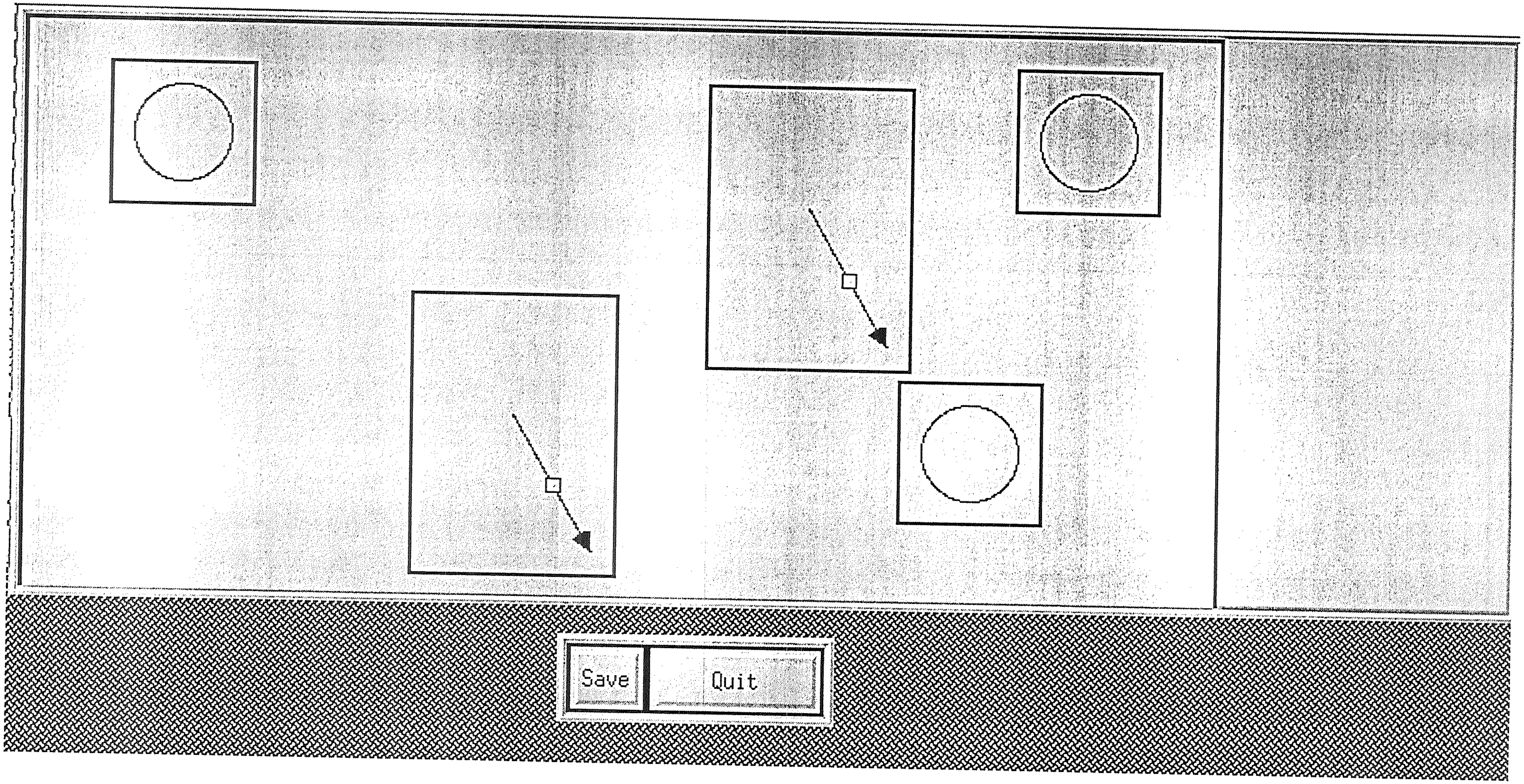
resized(Obj_name, (L,H)). Ha successo se l'oggetto Obj_name e' stato ridimensionato ed istanzia le variabili L ed H alle nuove dimensioni dell'oggetto.

selected(Obj_name, (X,Y)). Ha successo se l'oggetto Obj_name e' stato selezionato ed istanzia le variabili X ed Y alle coordinate in cui e' avvenuto l'evento di selezione dell'oggetto.



```
prototype(formato,formDialog([(resizePolicy,xmRESIZE_GROW),(x,40),(y,60)],[],[
     link(area1,area)
])).
prototype(area,drawingArea([(resizePolicy,xmRESIZE_GROW),(width,600),(height,800)],[],[])).
prototype(palla_movibile(P),drawingObj([(moveCallback,callback(muovi(P)))],[],[
     link(stato,ball([(width,50),(height,50)],[],[]))
])).
prototype(arco_movibile(A),drawingObj([(x,1),(y,1),(moveCallback,callback(muovi_arco(A)))],[],[
     link(linea,line([(x,30),(y,30),(lineArrow,1),(linePath,listpoint([(20,30),(60,100)])),
                                        (lineActive,1)],[],[]))
])).
object(grafo,formato).
state(s0).
state(s1).
state(s2).
transition(s0,s1,a).
transition(s1,s2,b).
link(grafo!area1,X,palla_movibile(X))<--state(X).
link(grafo!area1,L,arco_movibile(L))<--transition(S1,S2,L).
muovi(P):=moved(grafo!area1!P,X,Y)#set(grafo!area1!P,[(x,X),(y,Y)])#true.
muovi_arco(A):=moved(grafo!area1!A,X,Y)#set(grafo!area1!A,[(x,X),(y,Y)])#true.
```


Save
Quit


Save
Quit

5.11 Use of Theories

The database editor provide a graphicalinterface to edit **GEDBLOG** databases as a collection of multiple, heterogeneous theories.
Each theory is visualized in a View_Window, divided into four sectionswhich respectively contains the four differents classes of formulas: facts,rules,constraint,transactions.
Follow we present an example use of Theories:

The OIKOS Process Editor

A very interesting and practical experiment in using GEDBLOG has been carried out by realising a graphical editor for OIKOS. OIKOS [Mo84] is an environment that provides a set of functionalities to easily construct process-centred software development environment. In the following a brief description of OIKOS is provided. This description will highlight the characteristics that have a direct impact with the realisation of the OIKOS editor.

Specification

In OIKOS a software process model is a set of hierarchical entities. Each entity is an instance of one of the OIKOS classes and represent a modelling concept. An entity can be either structured (i.e. formed by other entities) or simple (i.e. a leaf in the model structure).
OIKOS defines a top-down method to construct process models. This method uses two descriptions of the entities: abstract and concrete. The abstract entities are introduced first and then refined in the concrete ones. At the end of the modelling activity an enactable model is obtained by adding to the defined entities the needed details. The method establishes some constraints about the entities (e.g. a coordinator entity has only the concrete representation) and their use as sub-entities during the model refinement (i.e. constraints in the model structure, for example a desk cannot have, among its sub-entity, a process). The entity classes defined in OIKOS are the following: Process, Office, Environment, Desk, Cluster, Session, Role and Coordinator.

The OIKOS Process editor should provide the modeller with an environment to easily specify software process models following the OIKOS method. It should permit to edit and store different models for different users. Different modelers must operate with the editor with the same set of static constraints (i.e. the basic constraints of the method that cannot be changed because they are an integral part of the method) but they can work with different dynamic constraints (i.e. constraints on alternative ways to develop models). The editor interface layout is required to provide: a menu for selecting operations, a top level entity to store the edited process, windows to present the informations of different entitiy kinds and dialogs for user inputs.

OIKOS Process Editor Database

This section describes the database that implements the OIKOS Process Editor [Ap94]. An overview of the theories that compose the OIKOS Editor database is given, with some explanations of the semantics they express.
Theories have different purposes: to map the OIKOS model into logic predicates, to hold constraints on the modelling process, to define a graphical counterpart for OIKOS abstract entities, and to represent the editor layout using the suitable GEDBLOG mechanisms.
From the user side, the theories are black-boxes. Users are just required to instantiate two theories, in order to load their personal instance of the editor database (oikos.editor theory) and to manage the data specific to the process they create during editor sessions (this theory will be referred to as oikos.<process_to_create>).

The Database Theories

The OIKOS Editor Database consists of the following theories: oikos.editor, oikos.model, oikos.menu, oikos.proto, oikos.<process_to_edit>

oikos.editor

This theory declares the imported theories and instantiates a window to display the root process. The theory is also used to record facts regarding the layout of the graphical frame (exposed objects, selected objects, etc.). These facts are inserted by transactions along the course of execution sessions.

Figure 3 shows the content of this theory. The predicate *theory* is used to load all the theories that compose the OIKOS editor database, while the predicate *object* instantiates a window that stores the top_level entity of the process model.

```
theory('oikos.model').
theory('oikos.menu').
theory('oikos.proto').
theory('oikos.mini_dctu').
object(process_row,formDialog([(dialogTitle,string(top_level))],[],[
        link(the_row,rowColumn([(orientation,xmHORIZONTAL)],[],[]))])).
```

Figure 3: Oikos Editor Theory

oikos.model

This theory introduces the predicates which define an abstract representation of OIKOS entities, and the constraints to be satisfyed by OIKOS process structures.

```
kindc(process).kindc(office).kindc(env).kindc(desk).kindc(cluster).
kinda(ang_process).kinda(ang_office).kinda(ang_env).kinda(ang_desk).
kinda(ang_cluster).kinda(ang_role).kinda(ang_ses).
may(process,ang_desk).
conc(Name,Kind,Limbo) ==> kindc(Kind).
conc(Name,Kind,Angel,Res) ==> kinda(Kind).
part_of(Name1,Kind1,Name2,Kind2) ==> conc(Name1,Kind1,Limbo).
part_of(Name1,Kind1,Name2,Kind2) ==> may(Kind1,Kind2).
conc(Name,Kind,Limbo) ==> part_of(Name,Kind,Coord,coord).
```

Figure 4: Oikos Model Theory

The facts with predicates *kindc* and *kinda* distinguish angelic entity kinds from concrete ones respectively. A set of instances of predicate *may* is used to define the allowed inclusion relations among different kinds of entities. Then a set of static constraints is given to model OIKOS process models. The first and second constraints state that concrete and abstract entities must have respectively concrete and abstract kind. Notice the use of the *may* predicate in the fourth constraint formula. It states that an entity can be used as a part of another one only if the two entity kinds are in the *may* relation.

Notice also that the theory defines only static constraints. However, dynamic constraints on the process construction methodology can be added by adding a separate theory.

oikos.menu

The menu theory builds the graphical object corresponding to the OIKOS Editor main menu and attaches transactions to the menu items in order to perform the corresponding actions.The menu-bar is realized by a fact in the theory that instantiate a menu-bar prototype, a set of facts that defines the prototypes for the pull-down items of the menu and the transactions to be performed in response to selection events.

```
object(main_dialog,formDialog([(dialogTitle,string(menu))],[],[
    link(oikos_menu,menuBar([],[],[
        link(gen,cascadeButton([(labelString,string(general))],[],[
            link(gen_pull,gen_pull)])),
        link(edit,cascadeButton([(labelString,string(edit))],[],[
            link(edit_pull,edit_pull)])),
        link(obj,cascadeButton([(labelString,string(objects))],[],[
            link(obj_pull,obj_pull)])),
        link(manag_obj,cascadeButton([(labelString,string(managers))],[],[
            link(manag_obj_pull,manag_obj_pull)])),
        link(serv,cascadeButton([(labelString,string(services))],[],[
            link(serv_pull,serv_pull)])),
        link(util,cascadeButton([(labelString,string(utilities))],[],[
            link(util_pull,util_pull)])))]))])).
...
prototype(obj_pull,pulldownMenu([],[],[
    link(coord,pushButton([(labelString,string(coord)),
        (activateCallback,callback(sel(coord)))],[],[])),
    link(role,pushButton([(labelString,string(role)),
        (activateCallback,callback(sel(ang_role)))],[],[])),
    ...
    ...])).

sel(Entity):= selected(Old) #out(selected(Old)) &in(selected(Entity))
        #true.
sel(Entity):= true#in(selected(Entity))#true.
```

Figure 5: Oikos Menu Theory

The *sel* transaction is activated when an entity kind from the objects menu is selected. Its effect is to insert a new fact in the logic database (selected(Entity)) that records the entity kind that will be created next time an object create action is performed. The following picture shows the menu as it is visualized on the OIKOS Editor layout.

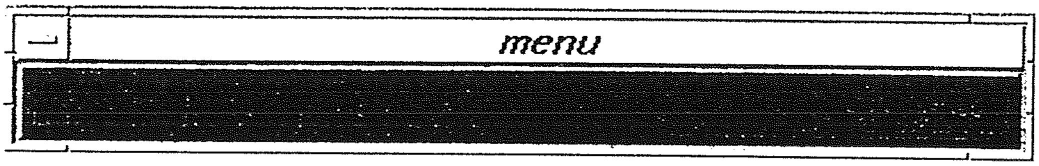


Figure 6: The Menu-bar

Notice that, due to the use of prototypes, the menu theory can be easely imported into a different databases and customized for.a different application.

oikos.proto

Oikos.proto deals with the graphic presentation of entities. It declares the graphical prototypes, the rules upon which the entities visualization depends and the transactions to be activated in response to events occourring on them.
According to the specification section , the entities that make up an OIKOS process description are: roles, coordinators, desks, clusters, environments, offices and processes. All but coordinators have an angelic counterpart. In order to attach a graphic representation to OIKOS Entities, consider they have a closed and opened status. When they are closed, only the icon representing the entity and the entity name are visible. The icons for entities and their angelic counterparts are given by the following Table:

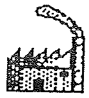
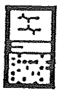
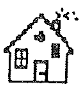
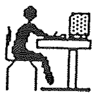
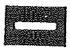
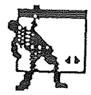
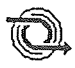
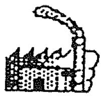
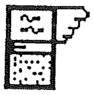
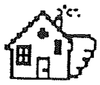
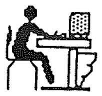
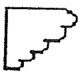
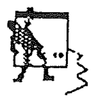
	Process	Office	Environ ment	Desk	Cluster	Role	Coordin ator
Concrete							
Angelic							

Table 1: OIKOS Entities Icons

Closed entities are graphically represented by the following prototype definitions.

```
prototype(microf(Path,Kind,Name),form([(marginHeight,1)],[],[
     link(pb1,pushButton([(labelType,xmPIXMAP),labelPixmap,pix-
map(Kind)),
        (activateCallback,callback(open(Name)))],[],[])),
     link(lbl,pushButton([(labelString,string(Name)),
        (activateCallback,callback(set_obj(Path,Name)))],[],[]))])).
```

This is a compund prototype that contains two buttons. One buttom shows the entity icon while the other one reports the entity name. Transactions are attached to button's activate events. The activate event on the icon button is linked to the *open* transaction by means of the callback mechanism.

As regards open entities, the graphical representation is different, depending on the following classification:

Compound Concrete Entities: Processes, Offices, Environments, Clusters and Desks are concrete compound entities. They are represented by a specification file written in the *Limbo* language and by the set of parts (sub-entities) that define their structure

```
prototype(generalconc(Name,Kind,Limbofile),
  formDialog([(dialogTitle,string(Name))],[],[
  link(fig,pushButton([(labelType,xmPIXMAP),
    (labelPixmap,pixmap(Kind)),
    (activateCallback,callback(close(Name)))],[],[])),
  link(descr,pushButton([(labelString,string(Name)),
    (activateCallback,callback(set_active(Name,Kind)))],[],[])),
  link(parts,partgrid(Name,Kind)),
  link(limbolab,pushButton([(labelString,string(limbo_update)),
    (activateCallback, callback(save_file(Name!limbodef!limbodeftxt,
      'oikos.mini_dctu',Limbofile)))],[],[])),
  link(limbodef,limbodefp(Name,Limbofile))
  ])).
```

Figure 7: Compound Entity Prototype

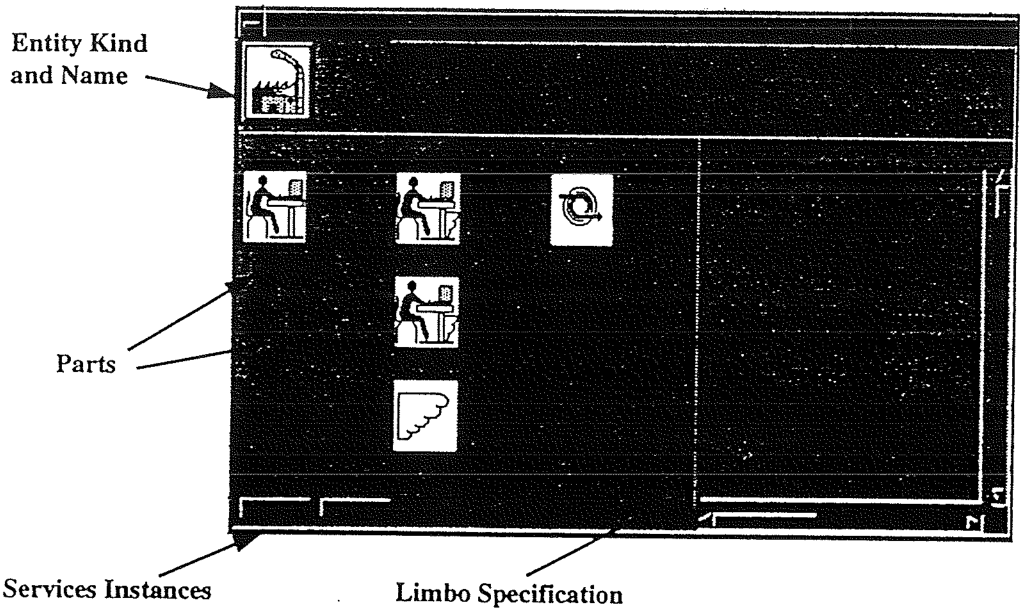


Figure 8: Compound Entity Instance

Angels: Angels represent the abstract specification of the entity. In Figure 9 an instance of an angelic entity is depicted.

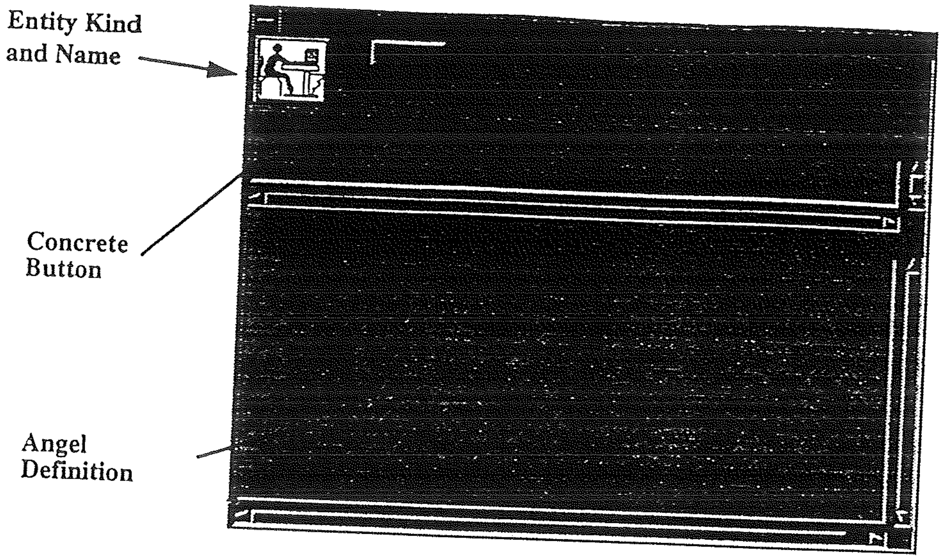


Figure 9: Angelic Entity Instance

Concrete Entities: Coordinators and Roles represent low-level entities in the OIKOS process description. Their definition is directly given into the entity of which they are parts.

The Oikos.proto theory connects also entities abstract representation to their graphical counterpart by the following rules:

```
object(Name,generalconc(Name, Kind, Limbofile)) <--
   conc(Name, Kind, Limbofile) & exposed(Name).
object(Name,generalabs(Name, Kind, AngSpec)) <--
   abs(Name, Kind, AngSpec) & exposed(Name).
object(Name, concrole(Name, Actor, Req, Behav)) <--
   role(Name, Actor, Req, Behav) & exposed(Name).
object(Name, conccoord(Name, Dests, Intheory)) <--
   coord(Name, Dests, Intheory) & exposed(Name).

link(Name1!parts!grid1,Name,microf(Name1!parts!grid1!Name,Kind,Name))<--
   part_of(Name1,Kind1,Name,Kind) & kindc(Kind) & exposed(Name1).
link(Name1!parts!grid2,Name,microf(Name1!parts!grid2!Name,Kind,Name))<--
   part_of(Name1,Kind1,Name,Kind) & kinda(Kind) & exposed(Name1).
```

Figure 10: Objects Instantiation Rules

The first group of rules states that a graphical object is on frame (see section 2.3.2 and 2.4) if the corresponding entity belongs to the knowledge base and the entity is in opened status (exposed). The second group of rules builds graphical objects corresponding to the parts of an open entity (refer to Figure 8).

Notice that these rules define the abstract data model of OIKOS process structure. A compound concrete entity is defined as an instance of the *conc* predicate. An angelic entity is modelled by the *abs* predicate, while the parts of a compound entity are described by the *part_of* predicate.

The motivations for using two rules to put parts into the graphic grid of a compound entity is that we want to keep angelic and concrete sub-parts into different columns.

oikos.<process_to_edit>

This theory records the abstract representations of process entities. In general, this theory will be empty when the editor is started for the first time on a process. It will be enriched along to the user interaction with the editor.
The following example of this theory defines only the top level entity of a process model that is called *mini_dctu.*

```
top_level(mini_dctu).
conc(mini_dctu,process,mini_dctu).
part_of(mini_dctu,process,manager_desk,desk).
```

Figure 11: Oikos Mini_dctu Theory

3.3 Execution of the OIKOS Editor Database

In the following execution simulation, we start the editor on the mini_dctu process as it is described by the theory of Figure 11. At the start-up, the execution engine calls the frame manager to obtain the graphic objects to display. Recall that the frame manager returns the list of graphic objects that can be proved in the logic knowledge-base. The objects that will be visualized in this case are the menu bar and the process root icon., as they are facts of the database.

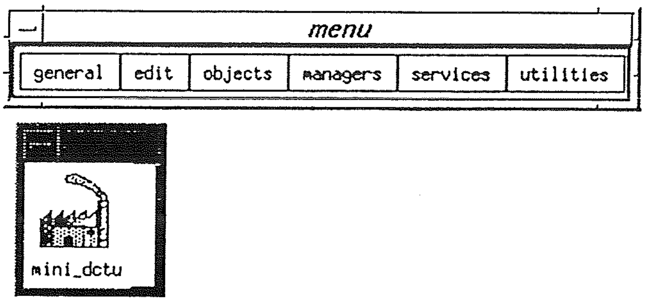
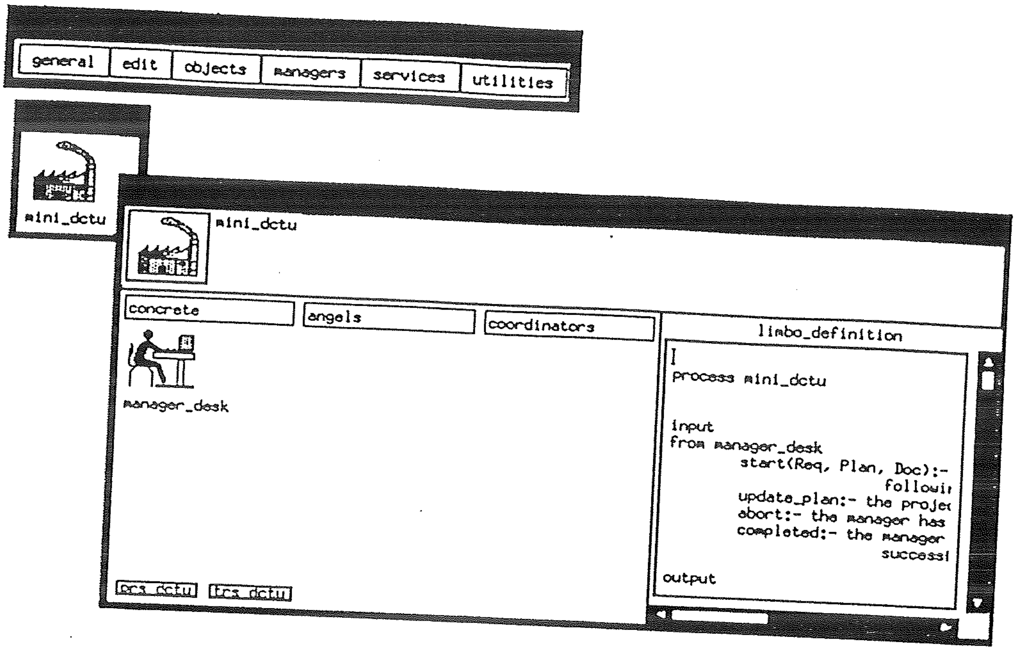


Figure 12: An editing session at start-up

Figure 12 shows the top-level representation of process *mini_dctu.* The process is represented by an icon, and is closed, in the sense that its internal definition is not visible. To open *mini_dctu,* the user clicks upon the icon button (the smoking factory).

The *activate* event on the button calls the *open* transaction of the oikos.proto theory. This transaction inserts a new fact in the knowledge base: `exposed(mini_dctu)`. Now, look at the first rule of Figure 10. It states that a compound concrete graphic object is deducible if the corresponding entity is defined in the database and it is exposed. So, exposing the mini_dctu entity causes this rule to fire, as the fact `conc(mini_dctu, process, mini_dctu)` is true in the knowledge-base (refer to oikos.mini_dctu theory in Figure 11).



```
I
Process mini_dctu


input
from manager_desk
        start(Req, Plan, Doc):-
                        followi
        update_plan:- the projec
        abort:- the manager has
        completed:- the manager
                        successl

output
```

Figure 13: Opening the Top_level Process

The system pops-up the graphical representation of the mini_dctu entity as a compound concrete entity. The window shows the internal structure of the mini_dctu process entity. The only entity that appears in the mini_dctu structure is the manager desk. This is due to the fact `part_of(mini_dctu,process, manager_desk, desk)` in theory oikos.mini_dctu (Figure 11), and to rule 5 in theory oikos.proto (Figure 10). To insert a new entity among the mini_dctu parts, first select the entity on which to operate (in this case the mini_dctu process) by clicking its name. This event activate a transaction that inserts the fact `active(mini_dctu, process)` in the database.

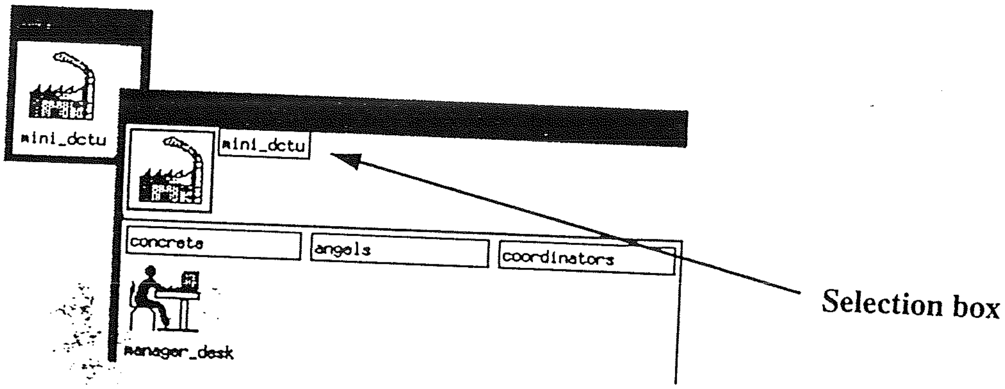
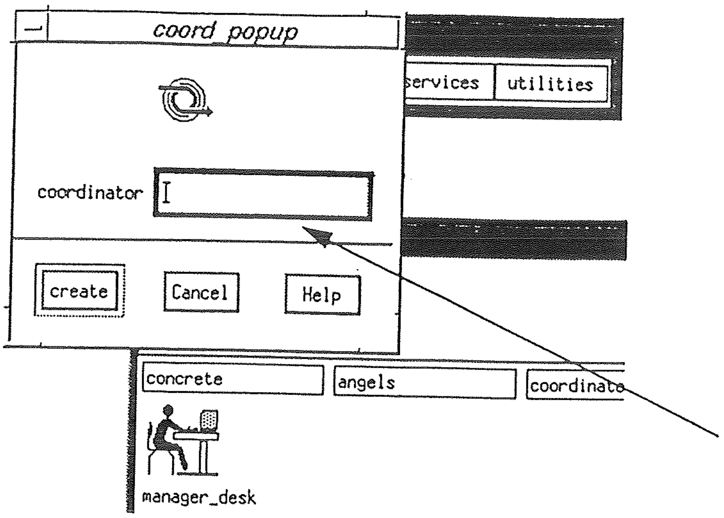


Figure 14: Selecting the active entity

Then chose the entity kind from menu *objects*. In the following example, a *coordinator* for the mini_dctu process is created by selecting the *coord* item from the *object* menu. A dialog window pops up, because the menu transaction asserts the fact `selected(coord)` and the rule

```
object(Kind,dialog(Entity,Etype,Kind))<--
     selected(Kind) & active(Entity,Etype).
```

is defined in the theory oikos.proto.



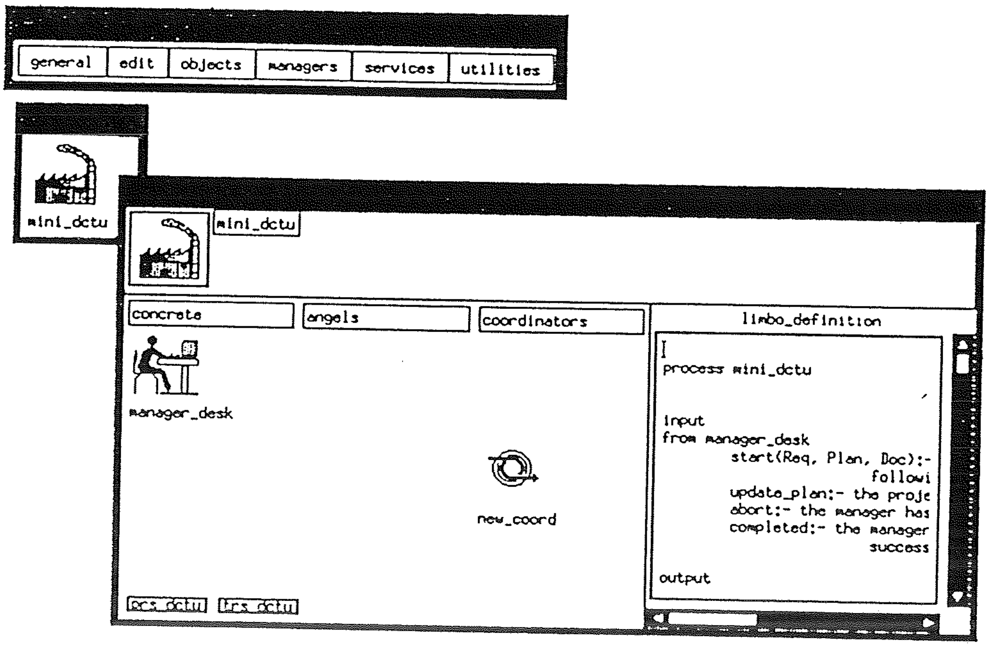
Input field for the entity name

Figure 15: Creating a sub-entity

The dialog asks for the name of the new entity. After having filled up the name field with the name *new_coord*, the user clicks the create button. This action starts a transaction

```
mk_conc_entity(Mother,Kind,Type):=
     selected(Etype) & get_par(Type!mess!row1!nametxt,[(value,Name)]) &
     kindc(Etype)
     # out(selected(Etype)) & in(conc(Name,Type,Name) &
     in(part_of(Mother,Kind,Name,Type))
     # true.
```

that in this case instantiates the new facts `conc(new_coord,coord,Name)` and part_of(mini_dctu, process, new_coord, coord). Furthermore, the transaction removes the fact `selected(coord)`, so the dialog is dismissed. Now, the new coordinator is shown among the parts of the mini_dctu process by means of rules in oikos.proto theory of Figure 10. The correctness of the new schema is automaticaly granted by the constraints, which participate to the deduction process.

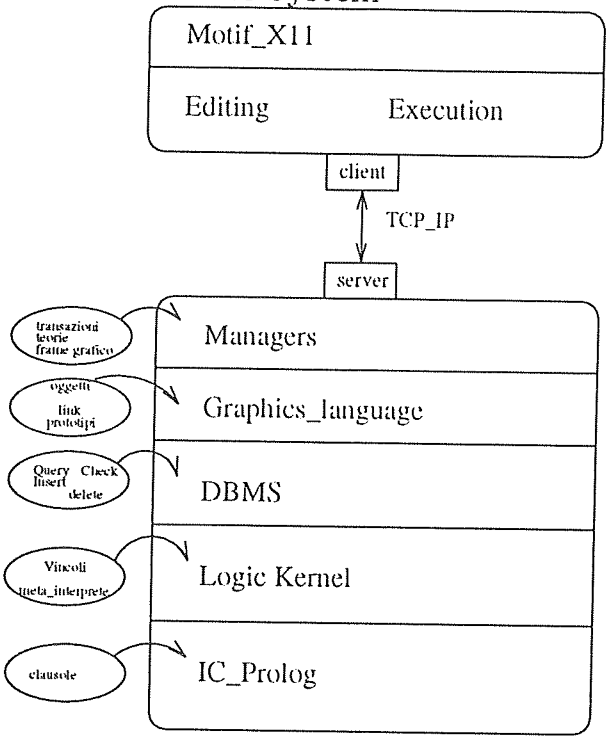


```
process mini_dctu


input
from manager_desk
          start(Req, Plan, Doc):-
                            followi
          update_plan:- the proje
          abort:- the manager has
          completed:- the manager
                            success

output
```

Figure 16: The Final Result

Appendice

Architetture of System



Bibliography

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

[2] R. Ahad & A. Basu . ESQL: A Query language for the Relation Model Supporting Image Domains, Proc. of the 7th Int. Conf. on Data Eng., Japan, April 1991, pp. 550-559.

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

[4] Asirelli, P. Castorina, G. Dettori . A Proposal for a Graphic-Oriented Logic Database System, IEEE Proc. of The 2nd Int. Conf. on Computers and Applic., Pekin, June 1987.

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

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

[7] Asirelli P. Inverardi and V.Raffaelli. - Using Constraints and Parallelism in Queries Evaluation: a top-down/bottom-up computation model . Draft .

[8] 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.

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

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

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

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

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

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

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

[16] 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, Universita' di Pisa, Ottobre 1986.

[17] 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.

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

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

[20] S. M. P. Julien Graphical in Micro-Prolog, Research report DOC 8217, Imperial College, London, 1982.

[21] R. A. Kowalsky Predicate Logic as Programming Language, Proc. IFIP - 74 Congress, 1974, pp. 569 - 574.

[22] R. A Kowalsky Logic and Data Bases, Logic Programming Meeting, Imperial College, London, Maggio 1976.

[23] R. A Kowalsky Logic for Problem Solving, Artificial Intelligence Series, N. J. Nillson Ed., Int. Symp. on Logic Programming, Atlantic City, 1984, pp. 118 - 125.

[24] R. A Kowalsky Logic Programing, IFIP, 1983, pp. 133 - 145.

[25] U. W. Lipeck Trasformation of Dynamic Integrity Constraints into Transaction Specifications, Lecture Notes in Computer Science No. 326, Springer Verlag, Bruges, Settembre 1988.

[26] J. Lloyd Foundations of Logic Programming, Springer Verlag, New York, 1984.

[27] F. Mauro Basi di Dati Logiche: un Approccio al Trattamento delle Transazioni, Tesi di Laurea, Dip. di Informatica, Universita' di Pisa, Novembre 1985.

[28] J. M. Nicolas Logic for Improving Integrity Cheching in Relational Data Base, Acta Informatica No. 18, 1982, pp. 227 - 253.

[29] F. C. N. Pereira Can Drawing Be Liberated fom 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.

[30] V.Raffaelli

- La programmazione Logica ed it metodi di valutazione di query logiche ricorsive: confronti e proposte.Thesis of the University of Pisa, 1990.

[31] J. A. Robinson A Machine-Oriented Logic Based on the Resolution Principle, JACM, Vol.1, No.12, Gennaio 1965, pp. 23 - 41.

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

[33] E. Y. Shapiro, A. Takeuchi An Object-Oriented Programming in Concurrent Prolog, New Generation Computing, Vol.1, No.1, 1983, pp. 25 - 48.

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

[35] P. Whiederhold Database Design, Computer Science Series, McGraw Hill Book Company, 1983.

[36] S-K Chang Visual Languages: a tutorial and survey, IEEE Software 4, 1987, pp. 29-39 .

[37] C. Beeri Data Models and Languages for Databases, Proc. ICDT'88, LNCS No. 326, Springer - Verlag, pp. 19-40.

[38] R.R. Berman, M. Stonebraker, GEO-QUEL: A System for the Manipulation and Display of Geographic Data, Computer Graphics 11 (1977), 186-191.

[39] N.S. Chang, K.S. Fu A Relational Database System for Images, In: N.S. Chang and K.S. Fu (Ed.), Pictorial Information Systems, Springer, 1980, 288-321.

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

[41] R.H.Goting Geo-Relational Algebra: A Model and Query Language for Geometric Database Systems, Proc. EDBT'88, LNCS No. 303, Springer - Verlag, pp. 506-527.

[42] R. Helm, K. Marriot Declarative Specification of Visual Languages, Proc. 1990 IEEE Workshop on Visual Languages, IEEE, pp. 98 - 103.

[43] C. Herold Spatial Management of Data, ACM Trans. on Database Systems Vol. 5 No. 4, Dicembre 1980, pp. 493 - 514.

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

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

[46] Lloyd, J. Foundations of Logic Programming, 2d edition, Springer-Verlag 1987.

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

[48] J.A. Orenstein Spatial Query Processing in an Object-Oriented Database System, Proc. ACM SIGMOD '86, pp. 326-336.

[49] M. J. Prospero F. C. N. Pereira On Programming an Interactive Graphical Application in Logic, Computer&Graphics Vol. 14, No. 1, pp. 7-16, 1990.

[50] 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.

[51] 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.

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

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

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

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

pert System = Knowledge + Meta-Interpreter, Dept. of Applied Mathematics, The Weizmann Institute of Science, Internal Report CS-84-17, 1985.

[56] C. N. Waggoner GKS - based graphic software adapts to changing tecnologies, EDN, marzo 1984, pp. 127 - 133.

[57] D. Weller, R, Williams Graphics and Database Support for Problem Solving, ACM SIGGRAPH Computer Graphics, Vol.10, 1976, pp. 183 - 189.

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

[59] Vieille L. "Recursive Query Processing: the Power of Logic", in Theoretical Computer Science, vol. 69,pp. 1-53, 1989.

[60] C. Zaniolo Deductive Databases: Theory Meets Practice, (Invited Paper) Proc. EDBT'90, Lecture Notes in Computer Science No. 416, Springer - Verlag, pp. 1-15.

[61] P. Asirelli, P.Inverardi, D. Aquilino, D. Apuzzo GEDBLOG Reference Manual (Edizione vecchia).

[62] P. Asirelli,D. Di Grande, P.Inverardi, F. Nicodemi Graphics by a logic Data Base Management System

[63] D. Apuzzo, D. Aquilino, P. Asirelli ADeclarative Approach to the Design and Relation of Graphic Interfaces. Ottobre 1994 Nota Interna B439.