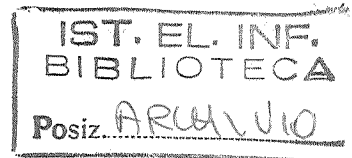


Consiglio Nazionale delle Ricerche

**ISTITUTO DI ELABORAZIONE
DELLA INFORMAZIONE**

PISA



**A Declarative Approach to the Design and
Realization of Graphic Interfaces**

D. Apuzzo, D. Aquilino, P. Asirelli

**Nota Interna B4-39
Ottobre 1994**

A Declarative Approach to the Design and Realization of Graphic Interfaces

D. Apuzzo[♥], D. Aquilino[♥], P. Asirelli[♠]¹

Abstract

This paper presents the design and realization of a software Process model Editor for the OIKOS environment. This editor has been realized using GEDBLOG, a multi-theory deductive database management system. GEDBLOG allows applications which heavily rely on graphic user interaction to be developed and enacted according to a declarative definitional style. It supports the consistent design and prototyping of graphic applications through an incremental development and/or by combining pre-defined theories. The case study we present highlights the GEDBLOG system features.

Keywords: *Deductive Databases, Graphic Interfaces, Logic Programming, Prototyping.*

Topics: visual user interfaces management systems, iconic systems.

1. Work partially supported by Progetto Finalizzato Sistemi Informatici e Calcolo Parallelo

[♥] Intecs Sistemi S.p.A. - Via Gereschi, 32 Pisa; e-mail: {aquilino,mimmo}@sole.intecs.it

[♠] Istituto Elaborazione Informazione C.N.R. - Via S. Maria, 46 Pisa; e-mail: asirelli@iei.pi.cnr.it

1 Introduction

The main goal of this paper is to give some evidence that declarativeness supports graphic applications prototyping quite well. In this perspective, the GEDBLOG system [As94] has been used to realize the presented approach. Beside the outlining of new concepts and techniques provided by GEDBLOG, the main contribution relies in the realization of a complex, non-toy application by means of it. The application presented here is a graphic editor for OIKOS [Ap94] that allows to handle software process model schemata.

GEDBLOG is a deductive database management system for the design, validation and execution of interactive 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. An important feature of the system is the provision of an integrity constraints checking mechanism which permits application properties under development to be proved. A graphic application is designed through a step by step refinement of the knowledge base. The application is developed consistently with respect to the defined integrity constraints that can be considered as those requirements that the application must satisfy.

The application knowledge is expressed as a deductive database [Sp84, Za90, We76] spread into multiple extended logic theories [Bro90, Mo89]. Each theory entails a piece of application data model and control by means of

- *facts* and *rules* to express the “general” knowledge;
- *integrity constraints* formulas to express either “exceptions” to the general knowledge, or general “requirements” of the application being developed;
- *transactions* to express atomic update operations that can be performed on the knowledge-base.

Graphic capabilities are handled by the system so that a graphic representation can be associated with a concept. The strategy is to use a logic language for the definition of graphic objects and their operations [As87a, As87b, He86, Hu86, Pn90, Pe86]. Since the representation is rigorously declarative, the graphic shape of an object is fully determined by the values of its attributes; there is no notion of global attributes. The goal is to combine graphic and non graphic components in the same declarative context.

Graphic objects, their relationships, and the set of operations that the user can perform will be included in the knowledge of the application. The result of this integration will be that the overall semantics of the application can be affected by constraints imposed only on the graphic representation and, vice versa, the graphic representation can be affected by semantics.

Combining the concepts semantics and their graphic denotation within the same context it makes the tool particularly suitable for developing and prototyping applications in the areas of visual languages [Ch87, He90], graphic interfaces and some aspects of CAD.

2 The GEDBLOG System

The system is implemented following a client-server approach as it is shown in Figure 1. The client side realizes the user interface and thus it deals with the database editing capability. It requires a window management system to run (in the actual version we use Motif 1.2 [OS] on X11R5). However, GEDBLOG does not depend on any particular support. A different window system can quite easily be connected.

The server is the part of the system that deals with the knowledge-base management. The actual version of the system is implemented in IC-Prolog [Co93] which forms the bottom layer. On this layer the Logic Kernel and the DBMS are built. These support knowledge handling at the logical level.

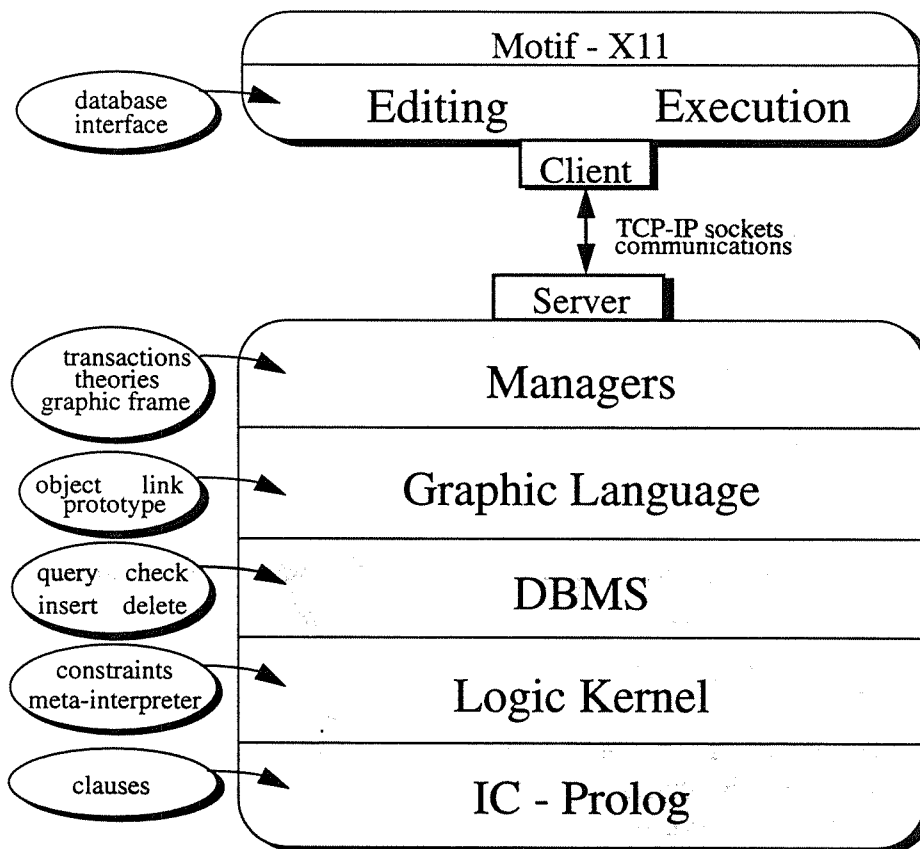


Figure 1: GEDBLOG Architecture

The DBMS is extended by the graphic component and by the managers. These make available to the clients the following features:

- multiple theories management, i.e. the capabilities to combine different pre-existing databases and load them as a single logical database;
- transactions management
- graphic objects management.

2.1 The Logic Kernel

In the following, some familiarity with Logic Programming and database terminology is assumed. Details can be found in [Ll87, Ga84].

The kernel of the system consists of a logic theory $T_{KB} = T_{DB} + T_{CONSTRAINT}$ where: T_{DB} is a logic program in which:

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

$T_{CONSTRAINT}$ are integrity constraint formulas of two kinds:

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

$$A_k \gg B_1, \dots, B_s$$

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

$$A_1, \dots, A_m \implies B_1, \dots, B_n$$

The connective "," stands for the logical connective \wedge .

Integrity checking is performed according to [As85]. The two forms of integrity formulas are used respectively to guarantee i) the construction of "correct with respect to constraints" applications (no illegal query will be answered) and ii) to perform controls at the user request. 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*, 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, *Controls*, and proves that they are true or false using a metalevel proof, on request from the user.

2.2 The DBMS

The DBMS has been realized following a meta-programming approach [Sa86, St85]. It consists of the implementation of primitive updating operations plus a language interpreter and a theorem prover (a Goal evaluator). The language interpreter is used to evaluate user-defined updating operations (i.e. transactions).

The logic theory T_{KB} , described in the previous section is augmented by a meta-theory $T_{KBMS} = T_T + T_{PO}$ that realizes the management system.

T_T is the theory of transaction definitions: a set of clauses that define compound updating operations (*transactions*) with the following syntax and procedural interpretation:

$$trans := prec \# t_1, \dots, t_n \# post$$

To execute a transaction *trans*, the precondition (prec) must be verified in the current DB. If it is verified, t_1, \dots, t_n are executed and the corresponding postcondition (post) verified in the modified database. A failure makes the computation to proceed by searching for another suitable definition of *trans*. A transaction fails if all its defining clauses fail.

The t_i in the transaction bodies can only be user defined or system predefined transactions. Preconditions / postconditions in the definitions of transactions denote particular forms of *Controls* which must be checked before/after the execution of a set of operations (body of the transaction).

T_{PO} is a set of elementary operations provided as a meta-theory with respect to the T_{KB} .

2.3 The Graphic Language

The graphic language layer supports a language for the definition and visualization of graphic objects thus enriching the theory T_{KBMS} by T_{GRAPH} . This theory contains the definition of graphic primitives, the rules needed to interpret the representations of graphic definitions and the meta-interpreter to visualize graphic objects.

A pre-defined windowing system has been exploited thus assuming the existence of low level consolidated mechanisms to define and manipulate graphic entities. This does not mean that the obtained system is tightly bound to a particular graphic system, but it allows to rise the power and make easier the specification of objects and interactions.

GEDBLOG uses the technology supported by Motif [OS] and X11 [Sc86], and integrates it into a completely declarative environment. This means that *graphic objects* and *interactions* are specified by a language provided by GEDBLOG and based on a clausal representation according to the following elements:

- *prototypes*, i.e. templates of graphical objects;
- *graphic objects*, i.e. instances of prototypes;

2.3.1 Prototypes

Prototypes in GEDBLOG can be either primitive or user-defined:

Primitive prototypes

A primitive prototype is directly implemented as a GEDBLOG primitive. Primitive prototypes are grouped into two classes: simple and structured. Simple prototypes cannot contain sub-components, while structured ones are able to include prototypes as sub-parts.

- Simple: *line*, *ball*, *text*, *label*, *separator*, *pushButton*, *arrowButton*, *toggleButton*, *drawn-Button*
- Structured: *form*, *rowColumn*, *formDialog*, *mainWindow*, *messageBox*, *drawingObj*, *drawingArea*, *bulletinBoard*, *list*, *cascadeButton*, *scrolledWindow*, *fileSelectionBox*

The two lists above define names of types that correspond to Motif [OS] widgets, apart from *ball*, *line* and *drawingObj* that are a GEDBLOG extension to Motif widgets in order to support the representation of low-level graphic objects.

User-Defined prototypes

The above defined prototype can be used to define new graphic objects that in turn can be used to define new objects. To this end GEDBLOG provides the following defined predicates: *prototype*, *object* and *link*.

A user-defined prototype is defined in GEDBLOG by means of the predicate *prototype* :
`prototype(Proto_name(List_ParForm), Proto).`

```
Proto ::= Base_proto(Resources, Operations, Links).
Resources ::= [(resource_name, Val), ..., (resource_name, Val)]
Operations ::= [Goal & ... & Goal]
Links ::= [(link(link_name,Proto_name(List_ParAct)),
           | link(link_name,Proto)),...]
```

Base_Proto is a graphic primitive-type while *Proto_name* is the name of user-defined prototype. Goals are queries to the logical knowledge base. They can be used to instantiate variables used in the resources values.

Val defines the value associated to resources . To handle the different type of values that

can be used for resources, the following type constructor have been defined:

```
Val ::= file(File)| pixmap(Pixmap)| string(String)|callback(Transaction)
      | Numeric_value | Variable.
```

The names of the resources, that can be set up for each basic prototype, correspond to Motif widgets resources.

2.3.2 Graphic Objects

A graphic object is instantiated by means of the predicates *object* and *link*:

`object(Object_name,Base_proto)`

that defines the graphic object *Object_name* starting from the basic prototype *Base_proto*.

`object(Object_name,Proto_name(List_Par_Act))`

that defines a graphic object given the user-defined prototype *Proto_name*

`link(Child_name,Object,Base_proto)`

builds a sub-object (child) of *Object* whose name is *Child_name* and whose type is the basic prototype *Base_proto*.

`link(Child_name,Object,Proto_name(List_Par_Act))`

builds a sub-object (child) of *Object* whose name is *Child_name* and whose type is the user-defined prototype *Proto_name*.

2.4 The Server Managers

The managers layer defines the services that are provided to access, modify and load the knowledge stored into the server logical database. These services are used by the clients to edit GEDBLOG databases, and execute the resulting specifications.

Theory Manager

It allows to load multi-theories databases into the server logic database. GEDBLOG supports this by the pre-defined predicate *theory*. By this predicate, different theories can be included in a database definition. The knowledge expressed by the actual database is considered to be the union of the knowledge expressed by the imported theories.

Graphic Frame Manager

The frame is the abstract logical space where objects are visualized and available for user interaction. In GEDBLOG the frame is managed in a strictly declarative way. This means that the objects on the frame at a certain instant are all and only those that can be proved (deduced) at that time in the knowledge base.

The frame manager can be used along to the specification execution to access the list of all objects that are on the frame at a certain instant.

Transactions Manager

The mechanism that GEDBLOG uses to catch the input events that happen on visualized objects is the same used in Motif widgets. It is based on the automatic activation of a procedure (callback) that handles the kind of event the object receives. Transaction manager enables client applications that catch an event on an object to give the control to the transaction execution feature of the DB management system of GEDBLOG.

2.5 The GEDBLOG client side

The GEDBLOG client supports the database editing graphical interface for GEDBLOG database editing and the execution of GEDBLOG applications. To do this, it interacts with the managers level of the server by means of a communication protocol based on messages exchange built on top of UNIX sockets.

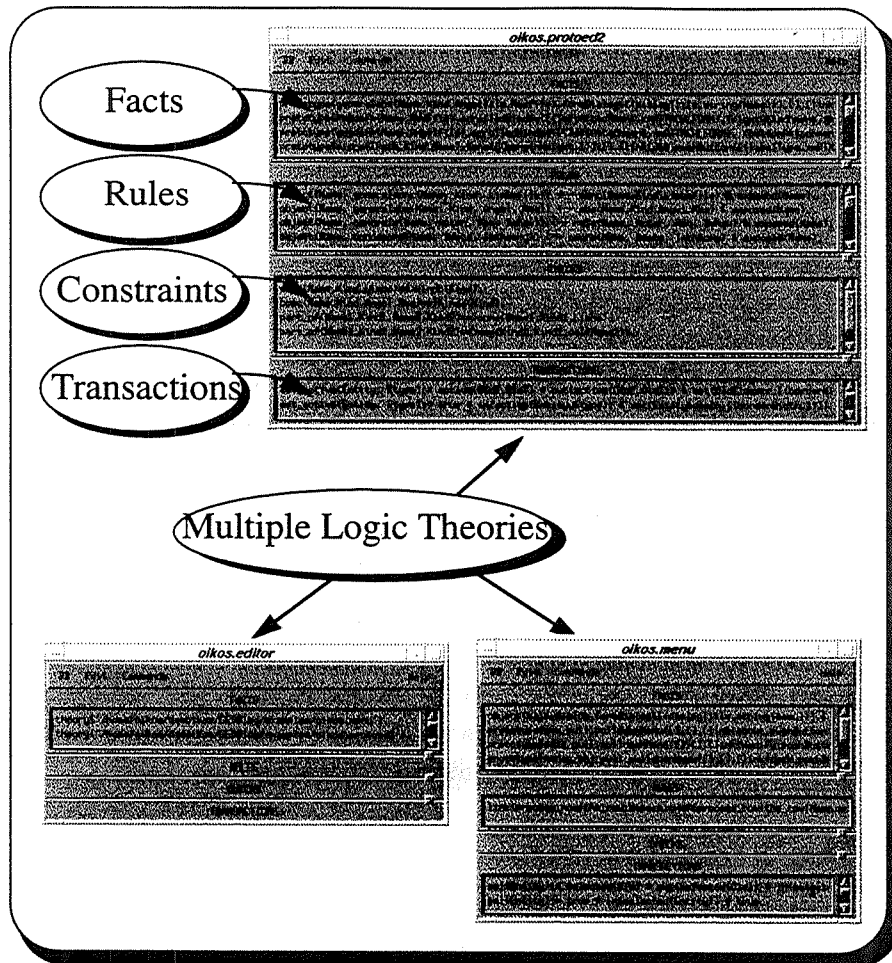


Figure 2: A GEDBLOG database

The database editor provides a graphical interface to edit GEDBLOG databases as a collection of multiple, heterogeneous theories. Each theory is visualized in a *View-Window*, divided into four *sections* which respectively contains the four different classes of formulas: facts, rules, constraints and transactions.

Once a database is loaded into the server knowledge-base, the logic specification it defines can be executed. The execution client realizes the support to this phase. It shows the graphical objects that are stored in the logical frame and calls the transactions activated by the interaction of the user with the objects.

As transaction execution may update the knowledge-base, the execution client is also responsible to keep up-to-date the visualized objects and their attributes.

To execute a GEDBLOG database the execution client performs a cycle made of the following steps:

- obtain from the server the list of graphic objects on frame and update the display

- when an event on a visualized object is caught, call the associated transaction through the server transactions manager
- wait for the server to perform the knowledge-base update operations.

The cycle can be aborted by quitting the execution client and returning to database editing

3 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.

3.1 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 entity kinds and dialogs for user inputs.

3.2 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>).

3.2.1 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 satisfied 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.

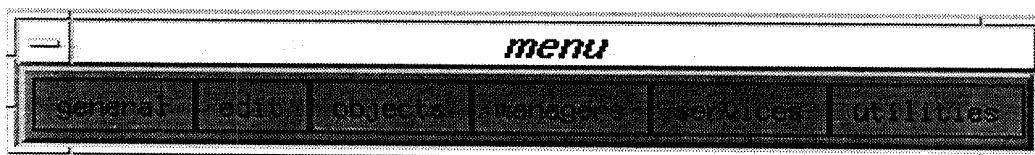


Figure 6: The Menu-bar

Notice that, due to the use of prototypes, the menu theory can be easily 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 occurring 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	Environment	Desk	Cluster	Role	Coordinator
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(lb1,pushButton([(labelString,string(Name)),
  (activateCallback,callback(set_obj(Path,Name)))]),[],[])))).
```

This is a compound prototype that contains two buttons. One button 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, limbodef (Name, Limbofile))
  ])).
  
```

Figure 7: Compound Entity Prototype

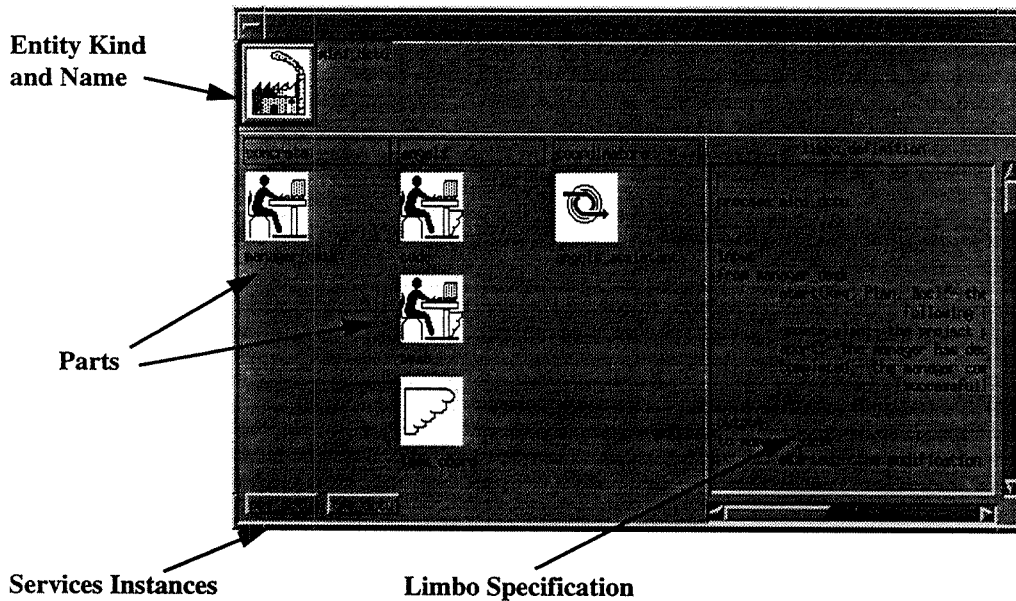


Figure 8: Compound Entity Instance

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

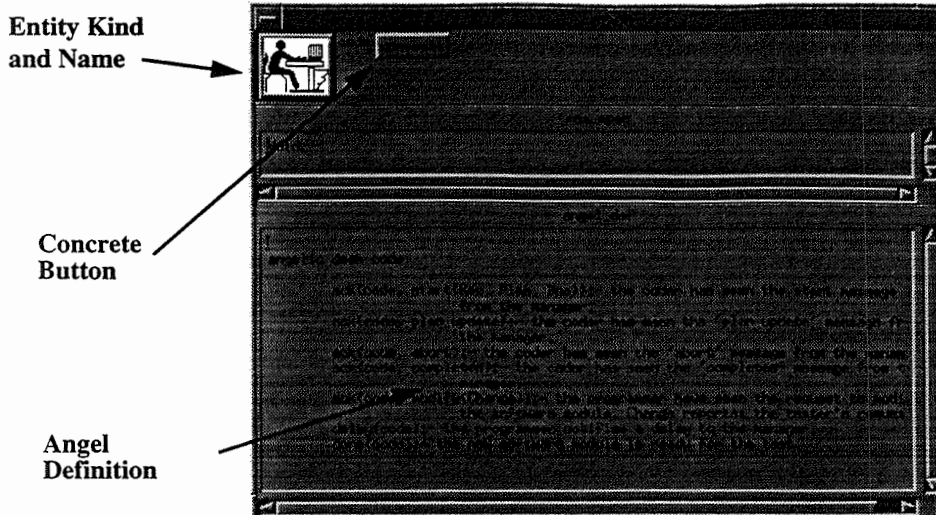


Figure 9: Angelic Entity Instance

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.

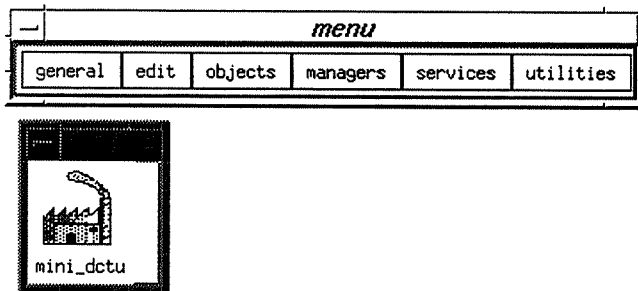


Figure 12: An editing session at start-up

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).

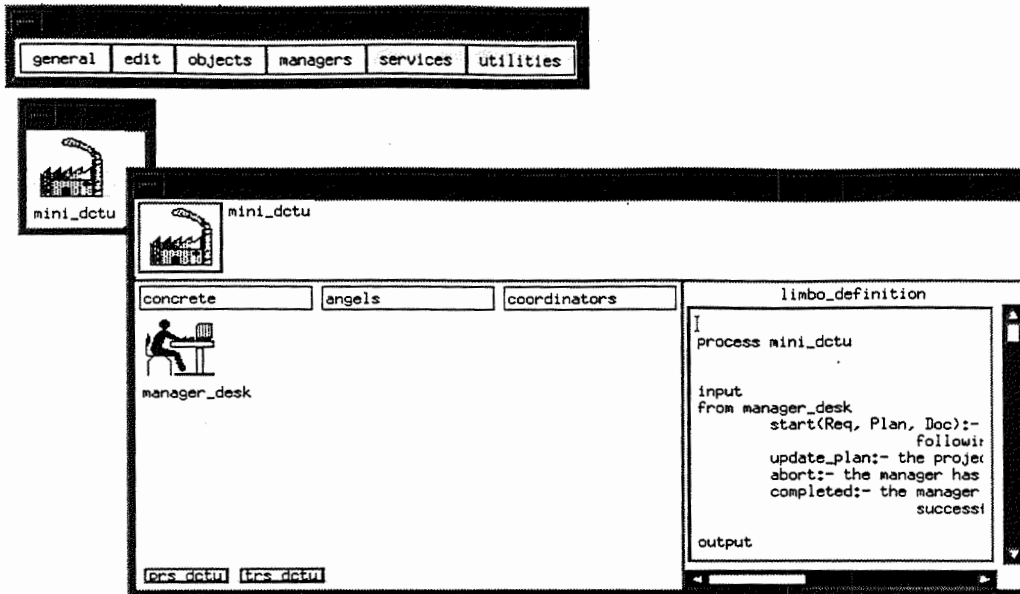


Figure 13: Opening the Top_level Process

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.

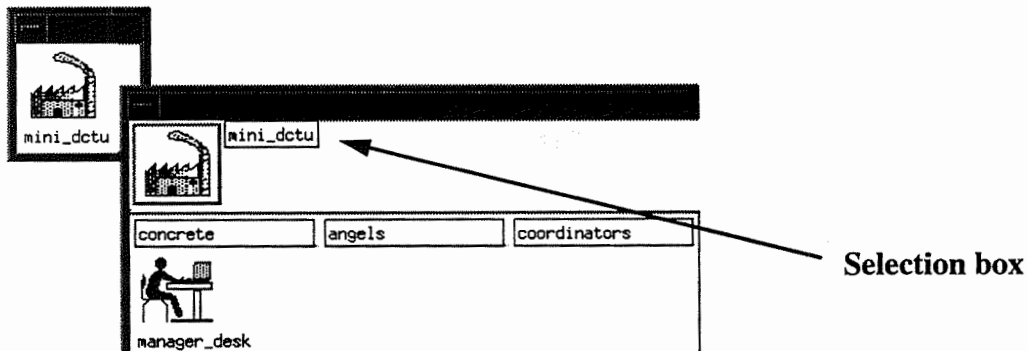


Figure 14: Selecting the active entity

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`.

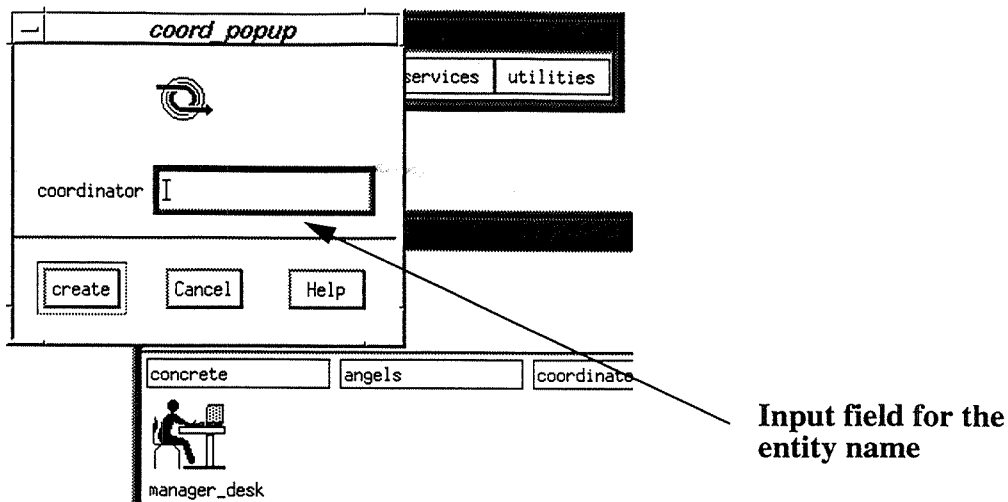


Figure 15: Creating a sub-entity

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!row!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 automatically granted by the constraints, which participate to the deduction process.

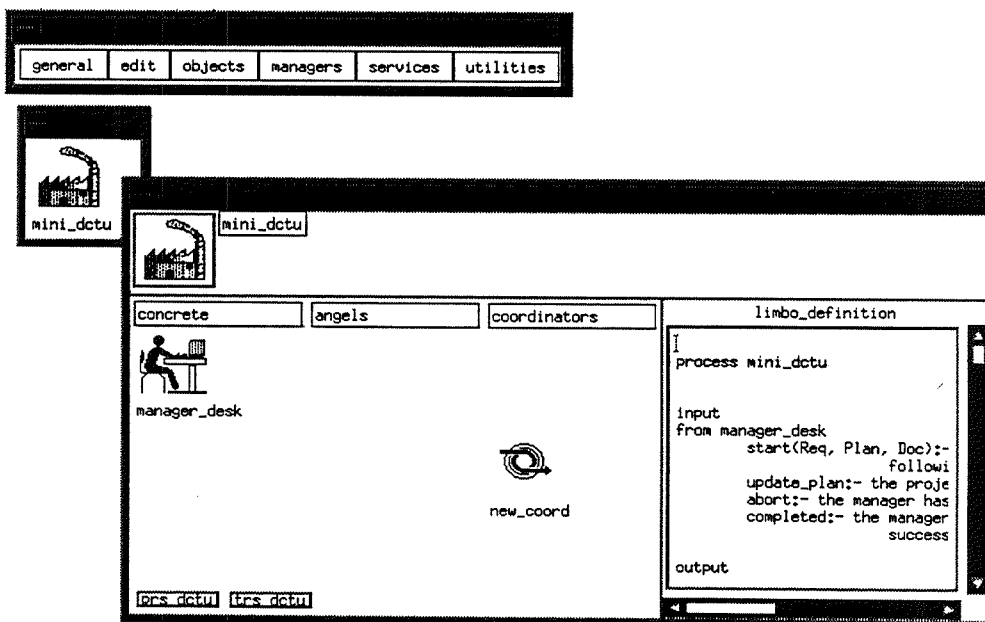


Figure 16: The Final Result

4 Conclusions

We have presented GEDBLOG, a system to handle graphic objects declaratively by means of a logic database management system environment.

The (graphic) data language is an extension of the language used to define a deductive database. The syntax is based on Horn logic (definite clauses) and a mechanism is provided to handle integrity checking. Integrity constraints can be defined on graphic objects and on their visualization. Since graphic objects are handled within a logic database, all the advantages of logic are exploited so that the resulting system is declarative, deductive and its semantics is well founded.

One application area that seems to be very suitable for a system like GEDBLOG is the visual languages area [Ch80, Ch87], since it is very natural to assign, to each graphic object, its corresponding operational semantics that, usually, includes non graphic information as well. As an example of an application, we have presented the definition of the OIKOS process editor. The example shows that only a very few rules are needed to define and handle something as complex as this application.

To obtain the final GEDBLOG system, we have extended the logic database management system by integrating it with a graphic language and mapping it to Motif and X11 primitives [Sc86]. At the present time, the system is implemented in C and IC-Prolog [Co93] and runs on a Sun 4 workstation under Unix 4.1.x [As90, As94b].

Acknowledgements

We thank Prof. Carlo Montangero, for his help in the specification of the OIKOS process editor, and Dr. Chiara Renso, for her careful review of this paper.

References

- [Ap94] D. Apuzzo, D. Aquilino, C. Montangero. OIKOS Process Editor Users Manual. Intecs Sistemi, Internal Report, 1994.
- [As85] P. Asirelli, M. De Santis, M. Martelli. Integrity Constraints in Logic Data Bases. *Journal of Logic Programming*, Vol. 2, No. 3, Ottobre 1985.
- [As87a] P. 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.
- [As87b] P. Asirelli, G. Mainetto. Integrating Logic DataBases and Graphics for CAD/CAM applications. *IEEE WorkShop on Lang. for Automation*, Vienna, August 1987, pp. 173 - 176
- [As90] P. Asirelli, D. Di Grande, P. Inverardi. GRAPHEDBLOG Reference Manual. I.E.I. Internal Report B4-08 February 1990.
- [As94a] P. Asirelli, D. Di Grande, P. Inverardi, F. Nicodemi. Graphics by a Logic Database Management System. to appear in *Journal of Visual Languages*, 1994.
- [As94a] P. Asirelli, P. Inverardi, D. Aquilino, D. Apuzzo. GEDBLOG Reference Manual. in preparation.

- [Bro90] A. Brogi, P. Mancarella, D. Pedreschi, F. Turini. Composition Operators for Logic Theories. In Computational Logic, Symposium Proceedings, editor J.W. Lloyd, Springer-Verlag, 1990.
- [Ch80] 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.
- [Ch87] S-K Chang. Visual Languages: a tutorial and survey. IEEE Software 4, 1987, pp. 29-39
- [Co93] Y. Cosmadopoulos, D. Chu. IC Prolog User's guide. *available by ftp from: src.doc.ic.ac.uk in /computing/programming/languages/prolog/icprolog.*
- [Ga84] H. Gallaire, J. Minker, J. M. Nicolas. Logic and Databases: a Deductive Approach. Computing Surveys, Vol.16, No.2, 1984, pp. 153 - 185.
- [He86] R. Helm, K. Marriot, Declarative Graphics. Lecture Notes in Computer Science No. 225. Springer - Verlag, London, July 1986, pp. 513 - 527.
- [He90] R. Helm, K. Marriot. Declarative Specification of Visual Languages. Proc. 1990 IEEE Workshop on Visual Languages, IEEE, pp. 98 - 103.
- [Hu86] W. Hubner, Z. I. Markov. GKS Based Graphics Programming in Prolog. Computer Graphics Forum, Vol.5, March 1986, pp. 41 - 50.
- [Ll87] J. Lloyd. Foundations of Logic Programming. 2^d edition, Springer-Verlag 1987.
- [OS] OSF/Motif Programmers Guide. Open Software Foundation, Cambridge, MA.
- [Mo84] C. Montangero, V. Ambriola. Oikos: Constructing Process-Centered SDEs. Software Process Modelling and Technology, editor A. Finkelstein and J. Kramer and B. Nuseibeh, Research Study Press distributed by J. Wiley and sons, London, 1994.
- [Mo89] L. Monteiro and A. Porto. Contextual logic programming. "Proceedings Sixth International Conference on Logic Programming", G. Levi and M. Martelli (Eds.), The MIT Press, 1989.
- [Pr90] 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.
- [Pe86] 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.
- [Sa86] S. Safra, E. Shapiro. Meta Interpreters for real. Inf. Proc. 86. H-J Kugler (Ed.), 1986, pp. 271-278.
- [Sc86] R. W. Scheifler, J. Gettys. The X window system. ACM Transaction on Graphics, Vol.5, No.2, April 1986, pp. 79 - 109.
- [Sp84] D. L. Spooner. Database Support for Interactive Computer Graphics. Proc. SIGMOD, 1984, pp. 90 - 99.
- [St85] L. Sterling. Expert System = Knowledge + Meta-Interpreter. Dept. of Applied Mathematics, The Weizmann Institute of Science, Internal Report CS-84-17, 1985.
- [We76] D. Weller, R. Williams. Graphics and Database Support for Problem Solving. ACM SIGGRAPH Computer Graphics, Vol.10, 1976, pp. 183 - 189.
- [Za90] C. Zaniolo. Deductive Databases: Theory Meets Practice. (Invited Paper) Proc. EDBT'90, Lecture Notes in Computer Science No. 416, Springer - Verlag, pp. 1-15.