I.E.I. - CNUCE- CGD

FINAL REPORT

Patrizia Asirelli and Gianni Mainetto

Contratto di ricerca IEI (CNR) / CASCADE GRAPHICS DEVELOPMENT S.P.A. (ROMA)

Table of Contents

1. *Overview of the Project and Report*

According to the Project aims (as it appears from the Allegato A of the Contract), the collaboration between the Istituto di Elaborazione dell'Informazione of C.N.R. and the Cascade Graphics Development as been realized by repeated meetings of the two respective representatives P.Asirelli and P. Castorina.

The first part of the year has been dedicated to know-how exchange. Assistance and materials has been provided concerning: the state of the art in Logic Programming; its theory and foundations; Logic Databases in general; a particular logic database management system realized by students, tutored by Dr. P. Asirelli for their Thesis at the Dipartimento di Informatica of the University of Pisa. Contribution to the thesis also arised from discussion between Asirelli and Castorina, expecially for the Transactions management part (see section 3).

The second part has been dedicated by Dr. Asirelli, Dr. Castorina and Dr. Mainetto to get insight into PHOGS, and propose possible integrations with the logic database management system. In particular, Dr. Mainetto has studied the PHOGS proposal in details to get an effecive proposal for extending MROLOG with PHOGS routines. Two reasons are behind the MPROLOG choice, one is that both DBLOG and EDBLOG have been implemented in MPROLOG. The other is that, since it was not clear what was the Prolog Language that Cascade wanted to use,we considered that MROLOG is general enough and has a kernel common to all other Prolog's, so that the proposal can be easily transferred into any other Prolog-like language.

This report will not get into implementation details which have been faced by Castorina while starting the implementation. Furthermore, we will report here just an overview of the overall ideas, where more care will be spent on the integration proposals.

We assumes the reader to be familiar with some Prolog Language and its interpreter. We also intend this report to be integrated with the following notes:

P. Asirelli, M. Martelli, "Integrity Constraints, Redundancy and Consistency in Logic Data Bases", CNUCE Int. Rep. C84-24, 1984.

P. Asirelli, P. Castorina, G. Mainetto, "Programmazione Logica, Basi di Dati Logiche e Grafica", *AICOGRAPHICS'85*, Milano, 4-8 Novembre, 1985.

P. Asirelli, P. Castorina, G. Mainetto, "Logic Databases and Graphics: A proposal for Integration", I.E.I. Int. Rep. B85-10, Settembre,1985.

P. Asirelli, M. De Santis, M. Martelli, "Integrity Constraints in Logic Data Bases", *Journal of Logic Programming*, Vol. 2, n. 3, Oct. 1985.

P. Asirelli, P. Castorina, G. Mainetto, "Integrazione di Ambienti grafici e Database Logici", Proc. of *Primo Convegno Nazionale sulla programmazione Logica*, Genova, 12-14 Marzo, 1986

P. Asirelli, P. Castorina, G. Dettori, "A Proposal For a Graphic-Oriented Logic Database System", to be presented at *IEEE 2nd Int. Conf. on Computers and Application*, Peking, China, June 24-26, 1987.

2. *Introduction*

2.1 Some basic notion on Data Bases

We will recall now some notion on database that will help clarify the various part of the system we are presenting.

A *Data Dase* is a set of data collected and stored in a computer according to some particular criterion.

A *Database Management System* (*DBMS*) consists of the software that allows the user to:
- use and/or update the data in the DATABASE,
- use and reason about the data in abstract terms more than on implementation details.

Furthermore, the DBMS must posses the following features:
- *Security* - that is, protection against uncontrolled access to the data;
- *Integrity* - that is, control over certain kind of "Consistency Constraints";
- *Syncronization* - that is, maintenance of the system consistency when the system is used by more than one user, simultaneously.
- *Crash protection* and *Recovery.*

A Database System can be seen from different point of view, each one corresponding to a different level of abstraction.

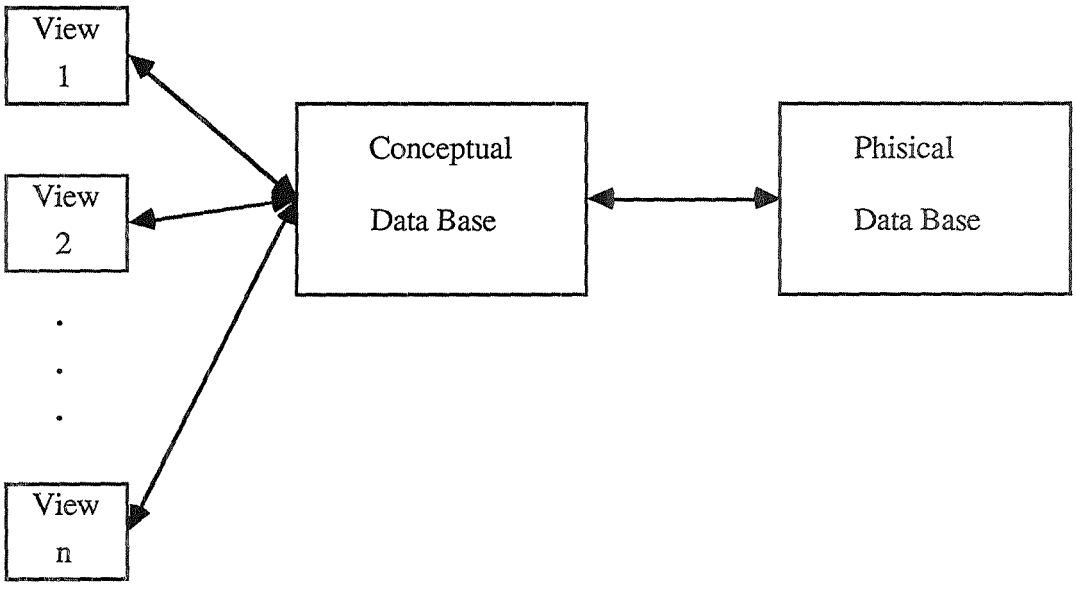

fig.1

The *Phisical Database* is the only database which really exists. It can be considered as a collection of Files and/or simple data structures.

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

Views are abstraction of parts of the Conceptual Database.

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

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

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

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

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

2.2 The syntax used

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

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

Each clause looks like:

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

where $A, B_1,\ldots, B_n$ are literals. A is the consequent, $B_1,\ldots, B_n$ are the premises and they look like $p(t_1,\ldots, t_m)$ where: p is a predicate symbol and $t_1,\ldots, t_m$ are *terms*.

The informal interpretation of a clause $A \leftarrow B_1,\ldots, B_n$ is that, A holds if $B_1,\ldots, B_n$ hold.

i - knowledge representation;
ii - knowledge acquisition;
iii - use of knowledge;

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

While knowledge in such a system is represented by means of Horn Clauses, knowledge acquisition 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.

2.3.1 Querying the LDB

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

On the other hand, logic programs are used via resolution of goals, where the initial goal is considered as the main program. It immediately follows that, when the database is represented by a logic program, a query is nothing else than a goal to be resolved against the program. The query evaluation process is resolution.

Integrity constraints are formulas which are properties of the logic program denoting the database and, in some cases resolution can still be used to verify them.

2.3.2 Basic Updating Operations

Updating operations in a LDB framework are related to knowledge acquisition. Operations are necessary to introduce/delete new/old 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 in an inconsistent state.

The introduction of a new integrity formula also cause verification of the actual database against the new formulas.

Updating operations also have to deal with redundancy problems. Such kind of problems are related to implementation and installation issues. They do not affect correctness of the system or its logical consistency.

2.3.3 Integrity Constraints Handling

Being the logic database we consider a logic program, integrity constraints (properties which the database must posses), can be considered as properties of logic programs, thus assimilating the problem of integrity constraint checking to that of logic programs properties proving.

Although logic programming offers a straightforward way of implementing deductive databases, some restrictions are needed to guarantee the termination of the query evaluation process and the evaluation of negative queries. Thus the class of logic programs has to be restricted to hierarchical

program definitions which do not allow recursive definitions [Clark 78, Shepherdson 84]. This restriction can be partially relaxed, at least with respect to negation and to certain kinds of queries [Barbuti 86].

In [Asirelli 85] an approach to integrity constraint handling for hierarchic databases is proposed in which a database is considered as consisting of a logic program plus a set of formulas, which must be proved to be true in the minimal model of the given program. Since a database will be updated, two approaches are proposed for integrity constraints checking. One approach (**The Modified Program Method**) considers a subset of the given logic formulas, called *IC - Integrity Contraints*, and uses them to modify the logic program automatically so that the given formulas are true in its minimal model (with respect to the model theoretic semantics).This means that all facts which do **not** satisfy *IC* are **not** provable/derivable from the modified logic program/DB (i.e. illegal queries cannot succed). The other approach (**The Consistency Proof Method**) considers a wider class of logic formulas (called *Controls*), and proves that they are true or false using a metalevel proof, on request from the user. The description of the algorithms is sketched in the next section, while a detailed description of them can be found in [Asirelli 85 and De Santis 85]. The integrity constraints formulas and the integrity checking algorithms can be extended to work on database which admit some recursion in the spirit of [Aquilano 86]. Structured database can be considered too. The extension to generally functional is described in [Mauro 85].

2.3.4 Redundancy

Redundancy probblems 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, than 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 consideration concern the amount of storage needed for the database.

Generally it is faster to derive information which are explicitely stated than to derive them 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 allows to save the store.

The above considerations must be taken into account when adding redundant information. If time has to be saved than redundant facts are accepted, while if space has to be saved they have to be rejected.
This all means that an LDBMS should provide for two modes of behaviour, letting the user to choose between them depending on its machine.

Details on the treatment and implementation of Redundancy Control are described in [Asirelli 84 and De Santis 85]

2.3.5 Transactions

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

A transaction definition language is generally defined by means of yet another language with its own interpreter that has to be integrated with the DBMS. Transactions allow a user to define its own operations at a more abstract level, in terms of other transactions or 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 tit has to be reset into a consistent state after system crashes or errors, 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, fiew efforts need to be made to build the interpreter using, once more, the basic resolution procedure used throughout the system.

3.The Logic Database Kernel and its Management System (DBLOG)

3.1 The Logic Database Kernel

A logic database management system EDBLOG [Mauro 85] has been defined which is an extension of DBLOG [De Santis 85] by introducing transactions definition and handling facilities. DBLOG considers the database system as consisting of three parts:

a) a logic program in which:

a.1) *the set of facts*, "unit" Horn clauses, are considered to be the Extensional component of the DB (EDB);

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

b) a set of integrity constraint formulas with:

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

$A_k \rightarrow B_1, \ldots, B_s$

which can be interpreted informally as: whenever A_k is true then B_1 and...and B_s must also be true;

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

i) $A_1,\ldots, A_m \rightarrow B_1,\ldots,B_n$

ii) $\rightarrow B_1,\ldots,B_n$

iii) $A_1,\ldots, A_m \rightarrow$

The informal interpretation for i) is that whenever A_1 and ... and A_m are true then B_1 and ... and B_n must also be true; analogously ii) means that B_1 and ... and B_n must be true and, finally, iii) means that A_1 and ... and A_m must be false.
Note that for formulas i)-iii), as well as for formula b.1), all the variables are intended to be universally quantified, apart from the local variables (i.e. variables occurring only on the right hand side) which are intended to be quantified existentially .

The basic components of the kernel can be depicted as follows:

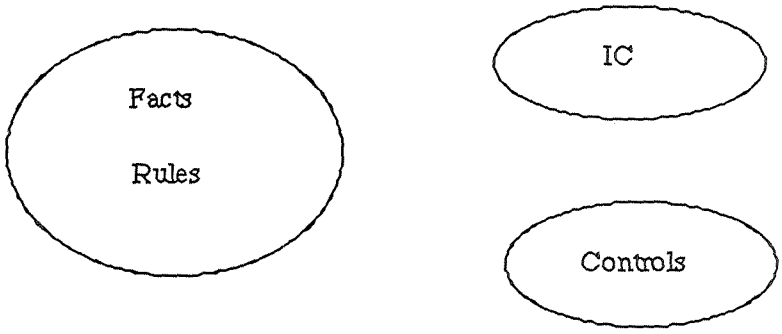

fig. 2

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

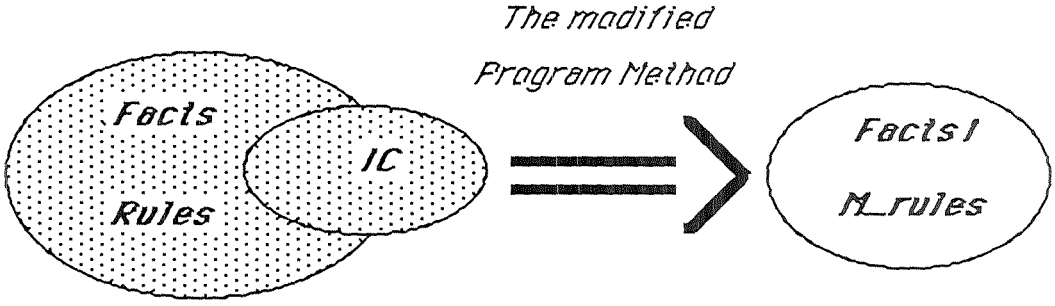

fig. 3

For example:

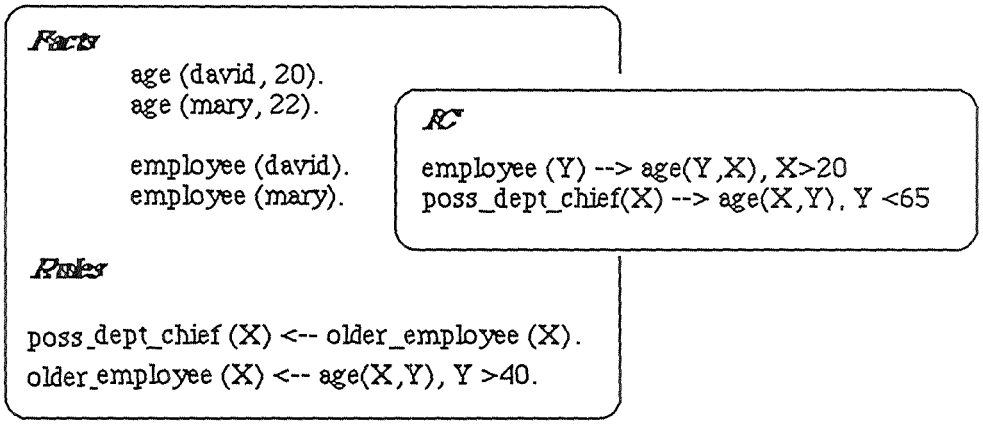

fig. 4

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

Implementation issues and the complete description of such operations can be found in [De Santis 85 and Mauro 85].

4. EDBLOG

EDBLOG is basically DVLOG extended with a theory of clauses Prolog-like to define transactions. Such clauses are either definite clauses or clauses with the following syntax`:

$$trans_i \leftarrow \mathbf{prec_1} \mid trans_1, \ldots, trans_n \mid \mathbf{post_1}$$

$$trans_i \leftarrow \mathbf{prec_2} \mid trans_1, \ldots, trans_n \mid \mathbf{post_2}$$

The language used to express transactions syntactically resembles Concurrent Prolog, with no annotated variables [Shapiro 83]. The informal interpretation is that to execute the operation *trans*, the precondition (**prec**$_1$ or **prec**$_2$) 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.

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. Thus, the *commitment* will be to the first clause whose precondition part succeeds: *or-nondeterminism* is not achievable. Or-nondeterministic behaviour can be obtained by defining transactions which do not have the commitment operation, i.e. standard Prolog clauses.

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 failure of postconditions or of some operations of the body, thus obtaining an *and-nondeterministic* behaviour of the clauses.

The system can be seen as an amalgamated theory [Bowen 82, Furukawa 84] consisting of the meta-theory (the theory which handles the evolution of the database), and the object theory (the logic database).

The set of elementary updating operations must be extended to introduce/delete transactions definition.system as a meta-theory with respect to the DB.

5. *Hints for the integration of EDBLOG with PHOGS.*

In the two final chapters of our relation we deal with problems related to integrating graphical capabilities with a database management system in a logic programming framework. The ultimate aim of the integration is to build a system which allows the programmer to develop graphical applications in a programming environment which offers a full set of graphical functions, while taking advantage of the facilities of logic and databases. This means to offer the ability to operate with objects having graphical and non-graphical properties and to generate other objects by deduction from those already existing, as well as the ability to prove properties of objects in the considered environment.

A satisfactory solution for this subject involves a lot of researches in different theoretical and applicative areas which until now have been considered separate or whose proposed solutions seem to be inadequate: modelling and graphics, deductive databases and DBMS, Prolog programming and graphics.

In the next two sections, an overview of the ideas that are going on in these fields will be provided. In the third section we will describe a feasible approach to integrating EDBLOG and PHOGS: the aim of the integration is to realize an environment to be used for testing new ideas and further developments. In the last chaper we will briefly describe the architecture of a system that should be investigated in order to give a final solution to the amount of problems involved in CAD/CAM applications.

5.1 *Modelling and Graphics*

The problem of modelling for computer graphics has been faced in the framework of relational DBMS.

Indeed, computer graphics manages complex objects that represent graphical data. A complex object can be thought of as a hierarchically organized collection of data describing an object. This idea works well for graphical data because most pictures are hierarchical in nature. To support such an idea, many computer graphics systems, both standard and non-standard, provide facilities for structuring complex objects (pictures) as hierarchies consisting of "segments" i.e. subpicture which can be manipulated and displayed indipendently from the rest of the picture.

When the concept of complex object is applied to a relational DBMS, a hierarchy of nested relations is obtained. These relations, conveniently arranged, are a practical tool sufficient to satisfy the requirements of computer graphics [Boeing 81, Weller 76]. The use of a relational DBMS as a place-holder into which the representations of segments are stored realizes the independency of the data model from the application, allows an easier integration of graphical and non-graphical data, permits the sharing of the graphical application data with related applications.

[Spooner 84] shows an approach to the integration of a relational DBMS as the data modelling component of a graphics application with an interactive graphics system. The interface between the DBMS and the graphics system is designed to be portable, that is adaptable to various standard graphics package like PHIGS and GKS.

Efforts are going on in the logic programming area to study the problem of defining a graphical interface to a database.

[Pereira 86] shows that the logic programming language Prolog can be used to hold the graphical representantion of an object and to describe how the object can be graphically displayed on a terminal or, viceversa, how an object can be identified via suitable graphical input operations.

The book-keeping of graphical representations is performed with the use of unitary clauses. Via this type of clauses both complex and primitive object representations are holded. The same set of program clauses (Prolog procedures) are used to implement view and identification operations: this is an example of the advantages that an implementation can take of the Prolog functions' invertible property.

This approach implies a different definition of the usual standard graphical abstract functions in order to satisfy the requirements of a non-procedural programming language such as Prolog. Further details about this last subject will be shown in the next section.

5.2 *Prolog Programming and Graphics*

From its beginning computer graphics has been connected to algorithmic languages. There are many applications for full graphical interaction with computers, mostly at a lower level than PROLOG. The graphical interaction is usually embedded in an algorithmic language by some graphics extensions. Therefore, since most algorithmic languages are procedural (the algorithm is a procedure), computer graphics is also procedurally oriented. In other words, in each implementation a set of graphics procedures (subroutines, ecc...) exists, representing the basic graphics functions.

Some attempts have been made to find a general set of graphics functions suitable for a wide range of applications and not connected to a particular algorithmic language. CORE, GKS, PHIGS and recently PHOGS are good examples for such a language-independent system. However, the structure of graphics package is strongly influenced by the procedural structure of the algorithmic language even if they are language-independent.

All these graphics packages consist of a set of functions for the manipulation of graphics data structures and for the management of graphics devices. The type of data structures and abstract functions provided by a package, and also the way into which the solution of a graphical problem can be implemented using such package are directly connected to the data structures and functions of algorithmic languages and to the von Neumann style of specifying an algorithm, i.e. via sequential steps.

For example, the most widespread graphics package GKS has a layered structure and provides one layer which is the unique part of the system dependent from the language, representing the interface between the system and the application language. This interface has been completely standardized for several algorithmic languages like Pascal and FORTRAN. The abstract GKS functions are represented in these two languages as subroutines or procedures, available in libraries. The GKS data types are easily mapped onto correspondent data types of the two languages (integer, real, array, etc...).

Prolog [Clocksin 81] on the other hand differs from other algorithmic languages in its basic concepts.

In Prolog the programmer only describes what problem has to be solved. He is not corcerned with how the specification is executed by a machine. The problem-specific knowledge (i.e. the logic) is separated from the control components which are left to the machine. This paradigm of separation between logic and control is realized in Prolog [Kowalski 79]: the Prolog programmer is free from any control specification.

Furthermore, many data types widely used in graphics are not available in Prolog implementations.

To define the Prolog-graphics interface the requirements of both the procedure-oriented graphics applications and the description-oriented host language Prolog have to be satisfied. There are two approaches to integrating graphics into Prolog:

a) the implementations of graphical functions especially designed for Prolog, taking into account the prolog features.

b) the connection of Prolog with an existing graphical package.

The first approach is actually used above all for small graphics packges which support simple Prolog applications. The visualization of the Prolog execution tree or the use of graphics for debugging Prolog programs are examples of such applications. This approach is also suitable for personal computer-iplementations, taking into account some performance and memory constraints. Graphics in Micro-Prolog [Julien 82] is a good example for this approach.

This first type of approach can be considered similar to the one briefly summarized in the previous section, but here the emphasys is on the prolog language rather than on the way to hold relations representing graphical objects in a database.

For more complex graphics applications a powerful graphics package is required. By taking the second approach, the problems of integrating a standardized graphics package in the description-oriented language Prolog have to be solved.

The second approach has been mainly investigated in the framework of integrating Prolog and the GKS standard graphics package.

[Syke85] presents a proposal for Prolog binding to GKS.

[Hubner 86] is another interesting proposal regarding the same subject. The authors extend the Prolog interpreter with two types of functionalities:

a) a set of built-in predicates which correspond one-to-one to GKS abstract functions (workstation control functions, output primitives, output attributes, some transformations, error handling).

b) a set of predicates expecially designed for the aim of managing and displaying on a terminal the segments (complex object representations or "pictures"). During the execution, the segments reside in the memory handled by the Prolog interpreter: they are a particular type of unitary predicates which can be loaded/stored from/onto the disk storage.

A special "segment interpreter" evaluates the segments and produces as result the set of graphical built-in predicates corresponding to their representations.

The concept of segment takes full advantage of prolog capabilities; indeed, one segment can have parameters and inside its definition Prolog control predicates can appear.

These ideas have been implemented on a Unix machine.

This paper provides many useful suggestions to be taken into account when a Unix programmer deals with the implementation problems of the integration.

6. *A proposal*

Given that the integration must take place between EDBLOG, actually implemented in MPROLOG [Mprolog 86] on a VM/CMS operating system using the simple database management support provided by every Prolog implementation, and PHOGS [Biagi 86], actually implemented in the C language on a 4.2 bsd UNIX operating system, we propose a type of integration that can be easily put into effect. This integration can provide a useful environment for exploring new ideas, testing future developments, gaining further experience on this subject.

The proposed integration supposes that the logic environment contains the application program and it directs the overall computation of the integrated system. In order to obtain an environment that is suited for graphical application programs, we must render the segment description visible in the logic part of the system. As previously noted, the description of segments constitutes a database, including information on hierarchical relationships among them. The main purpose of the integration is to increase the EDBLOG facilities to include segments management ones, while leaving to the PHOGS library the task of segment visualization and input handling. Application programs will be realized as a series of Prolog predicates which can take full advantage of the EDBLOG database management facilities and of the PHOGS high level device interface.

PHOGS is a library composed of a set of routines. These routines have their own interface with the application programs (parameters and their types), use their own data structures, and have a strategy for memory management. The most part of the implementation decisions will not be modified in the proposed integration. Particularly, the PHOGS PDS (Parent Data Structure) and HSS (Hierarchical Segment Storage) will not be changed and will coexist with the logic database of segments. PHOGS data structures and their logic counterparts will remain consistent during the computation of the applicative program.

The advantages of this integration are:
a) rapid development, ease of the implementation, PHOGS investment saving, primarily due to the fact that PHOGS can be used almost as it is, with only few changes;
b) possibility of dealing with segments as logical database objects on which facts and deductive rules can be defined in order to express graphical and non graphical properties;
c) solution to the problem of the segments archiving on secondary storage, which can be easily implemented in the logic part via EDBLOG facilities;
d) solution to the problem of managing the non-graphical data inserted in segments desciptions.

The main disadvantage of the proposed integration is the waste of memory at run time. Indeed, the contemporary presence of two representations of the same class of objects (segments) both in the logic and graphical parts of the integrated system produces this effect.

The integration takes place in the following three steps:
a) porting of EDBLOG on the UNIX system;
b) integration of Prolog and PHOGS via procedural extensions (built-in predicates);
c) rendering visible the segment representations to EDBLOG via the definition in Prolog of layer of graphical predicates that handle the logic database of segments before calling the PHOGS counterparts.

6.1 *Porting of EDBLOG*

EDBLOG is actually coded in MPROLOG over VM/CMS.

MPROLOG is an appealling Prolog implementation. Its main charateristics are the modular definition of the language, that allows separate compilations, and the presence of a set of tools that

constitute a flexible programming environment. Furthermore, in the last MPROLOG version a new tool, the compiler, is available. Compiler will increase the efficiency of the produced code. In the last version of this product [Logic 86], the programmer can develop applications using an interactive environment to test and debug his/her modules (PDSS), and then, when all modules perform their intended meanings, s/he can use a series of tools in order to produce an efficient code (Pretranslator, Compiler, Consolidator).

MPROLOG is also available on Unix systems, like on 4.2 bsd, the version of Unix running on Sun workstations. The various versions of MPROLOG are all highly compatible.

The acquisition of the 4.2 bsd version of MPROLOG is the cheapest solution that minimizes cost and time of EDBLOG porting. In this case only few changes to the actual EDBLOG code are needed due to the different way of naming file in VM and in UNIX.

In case of choosing another Prolog implementation running on 4.2 bsd Unix, the cost of porting would not be excessive due to the implementation decision of using, as far as possible, "standard" Prolog Dec-10 routines in VM EDBLOG. These routines are usually available in all Prolog implementations.

6.2 *PHOGS built-in predicates in EDBLOG*

Prolog is not a pure logic interpreter. The logic programming in Prolog is based on a procedural approach for the interpretation of logic. Some procedural extensions (built-in predicates) are also available in Prolog. Thus, we can add PHOGS functions at a Prolog procedural level using built-in predicates.

Many Prolog systems provide facilities to add built-in predicates without directly modifying the interpreter. One built-in predicate can be implemented in a high-level procedural language.

Consider the MPROLOG system on 4.2 bsd Unix over Sun machines [OS 86]: an MPROLOG program can define predicates that call "external" routines written in languages like C and Assembler. All the external routines coded in C constitute a set of modules loaded at run-time one at time when one of the routines of a module is called.

This MPROLOG capability is particularly well suited to the aim of extending MPROLOG with PHOGS routines at present coded in C. A smal number of updates must be implemented in the actual PHOGS code:

a) change every PHOGS routine to return success or failure when executed;
b) modify the routines that returns one value (the "functions") in this way: add one more item to the routine parameter list; this new parameter will serve as a place-holder where the MPROLOG caller can find the output value, i.e. the value actually associated to the routine name;
c) define a mapping between the actual type of input/output parameters of routines and the available types of parameters that can be exchanged between MPROLOG code and C routines. Modify accordingly PHOGS routine code. The last version of MPROLOG provides support to exchange data of three types: integer, real and string.

That is all for PHOGS routines from the C point of view.

From the MPROLOG point of view there is still a problem to deal with. This problem occurs in the MPROLOG predicate that calls the PHOGS routine: calls to external C routines fail on backtracking and their effects are <u>not</u> "undone".

One possible solution is simply to ignore the problem: indeed the programmer is acquanted with such a behaviour and, if s/he wants, s/he can use MPROLOG extra-logical predicates ("flow control" facilities) to properly manage backtracking.

In order to provide a clean semantics for MPROLOG-PHOGS routines, we must define for every routine that changes the state of the graphical system an undo function that restores the state on backtracking. The correct semantics of a PHOGS routine can be directly implemented in MPROLOG, using a combination of MPROLOG and PHOGS predicates.

One possible "compilative" approach is to try to define a new tool that will be the first and mandatory phase of the applicative program translation process and will produce the intended sequence of code as output. For example, let ..a.. , *set_corresponding_text_font(1),* ..b.. be the sequence of code containing the PHOGS routine we want to undo on backtracking. We could map this sequence to the following MPROLOG code:

```
..a..,
inquire _corresponding_text_font  (XXX),
(set_corresponding_text_font (1), ..b..;
 set_text_font (XXX), fail)
```

provided that *XXX* is an unbound variable not presente in ..a.. and ..b.. .
Unfortunately, many PHOGS routines have different semantics depending on the state in which they are executed and the run-time state cannot be inferred from the text of the program. One example of such situation is the set of PHOGS output primitives that have different effects when executed in INOP or in SGOP state: in the former case they are immediately displayed, in the latter they are inserted in the current segment. This characteristic of PHOGS renders ineffficient the compilation solution.
We propose to define a layer of MPROLOG predicates that perform the expected actions on backtracking. If the current state of the graphical part of the system is needed, it is estabilished via the PHOGS routine *inquire_system_state_value* . Here is an example:

```
text_2 (S)  ←
inquire_system_state_value (STATE),
(STATE = "INOP",
          ( _text_2 (S) ;
            _delete_interactive_primitive, fail);
 STATE = "SGOP",
          ( _text_2 (S) ;
            _delete_segment_primitive, fail)
).
```

where *_delete_segment_primitive* is a new PHOGS routine that deletes the last element inserted in one segment. It is worth noting that PHOGS routine names have a prefix underscore sign to prevent recursion. This set of predicates can constitute a module that includes the external declarations to be linked to every graphical application program.

A complete study is needed for every routine to control whether actual PHOGS routine are sufficient to correctly describe the undo functionality or some new routines must be defined.

In the next section we will discussin details the semantics of the MPROLOG-PHOGS routines that allow a user program to enter/exit into/from the states where segment representation are managed.

6.3 *Integration of PHOGS segments into EDBLOG*

In this section we describe the way of integrating the database of segments and their hierarchical relationships into EDBLOG. Such integration will be realized via the definition of a set of Prolog predicates, one for every PHOGS routine that handles segments. The tecnique is similar to the one described in the previous sectione regarding the undo function.

In PHOGS there are two states in which the user can manage segments representations: SGOP i.e. segment open and EDIT i.e. segment edit. We define an MPROLOG predicate for every routine that can occur in these two states and for those routines that enable the user program to enter/exit from these states. The purpose of MPROLOG predicates is to collect segment representations in a logical manner before calling the corresponding PHOGS routine. When the user program exits from one of the two states, the collected representations are inserted in the EDBLOG environment.

The MPROLOG definition of predicates is given in Appendix.

Segments are represented in EDBLOG as a set of "segment" facts. Every segment fact is a pair consisting of a segment name and the list of predicates that describe the segment. For example:

segment (dummy, *[poly_marker_2 (.....), set_color (...), execute (b)]*).
segment (square, *[poly_line_2 (1, 1, 1, 2, 2, 2, 2, 1, 1, 1)]*).

The hierarchical relationship among a "father" segment and its "son" subsegments, is also represented with a similar fact that describes for every segment the list of sons "executed" by the father.

subsegment (dummy, [b]).
subsegment (square, []).

In EDBLOG we can insert a couple of deductive rule that estabilish whether a segment is primitive or compound:

compound (Seg_Name) ← subsegment (Seg_Name, [_ | _]).

primitive (Seg_Name) ← subsegment (Seg_Name, [ ]).

About segment application data, we propose that they are completely managed by the MPROLOG predicate *insert_application_data* . *insert_application_data* has no PHOGS counterpart. This predicate has two parameters: a *tag* for the data and a *value*. When the edited segment is finally closed, application data will be inserted in EDBLOG as facts of the following form:

segment _name (tag, value).

For example, if the application program is editing the segment "a" and if it wants to add the following data to the current segment description:

insert_application_data (cost , 100)

this predicate will result, when editing will end, in the insertion of the following fact in EDBLOG:

a (cost , 100).

We have finally to deal with problems related to MPROLOG-PHOGS routines to enter/exit into/from SGOP and EDIT states.

From the logical point of view, the full set of predicates included between the entry into one of the two states (i.e. the set of operations that start with a *open_segment* or an *open_edit* operation) and the corresponding exit (i.e. the previous set ends with a *close_segment* or a *close_edit* operation), constitutes a "segment transaction" that is a single operation that updates the logical database of segments.

We want to model a segment transaction in a manner similar to the usual EDBLOG transaction, and in particular we expext that:

a) when a segment is inserted/modified in the logical part of the system, the database of segments is checked for consistency and the new item becomes part of the database if and only if IC and Controls are satisfied;

b) if the previous operation produces a failure, the effect of the segment transaction are completely undone also in the graphical part of the system.

In order to provide such a behaviour in EDBLOG, we define segment transactions as a set of clauses whose body begins with the predicate that enters in SGOP or EDIT state and ends with the following sequence: commit (!), predicate that exits from the corresponding state, commit (!).

The body of the clause representing a segment transaction is constituted from "primitive" backtrackable graphical predicates, calls to the usual user-defined EDBLOG transactions, primitive logical Prolog predicates, calls to Prolog user-defined predicates, but the programmer cannot insert operator insert Prolog operator like cut and slash to avoid problems related to such extra-logical Prolog mechanisms.
We embed a segment transaction definition in a particular predicate like *segment_tr* in order to check its synctatic defintion:

segment_tr (name1, Var1, ..., VarN) ← *open_segment (..), ... | close_segment* | .

segment_tr (name2, Var1, ..., VarM) ← *open_edit (..), ... | close_edit* | .

name1, name2, .. are ground atoms. The commit operator is mapped in cut. The definition of the four MPROLOG-PHOGS predicates is given in the Appendix: they are, as usual, MPROLOG predicates that call the corresonding PHOGS routine. *open_segment* and *open_edit* are backtrackable. *close_segment* and *close_edit* first call PHOGS routine; then issue a real EDBLOG transaction that tries to insert the segment representation and hierarchical relationship and application data into the database; if this attempt fails, the segment is also discarded from PHOGS database and the failure is propagated back that is the complete segment transaction fails due to presence of the commit operator.

7. *REFERENCES*

[Asirelli 84] Asirelli P. , Martelli M., "Integrity Constraints, Redundancy and Consistency in Logic Data Bases", CNUCE Int. Rep. C84-24, 1984.

[Asirelli 85] Asirelli P., De Santis M., Martelli M., "Integrity Constraints in Logic Data Bases", *Journal of Logic Programming*, Vol. 2, n. 3, Oct. 1985.

[Aquilano 86] Aquilano C., Barbuti R., Bocchetti P. and Martelli M., "Negation as Failure. Completeness of the Query Evaluation Process for Horn Clause Programs with Recursive Definitions", *J. of Autonated Reasoning*, n. 2, 1986, pp. 155-170.

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

[Boeing 81] "User guide: RIM 5.0 Prime PRIMOS", Boeing Commercial Airplane Company,Seattle, Whashington, 1981.

[Bowen 82] Bowen K.A. and Kowalski R.A., " Amalgamating Language and Metalanguage in Logic Programming", in *Logic Programming* , (Clark K.L. and Tarnlund S.-A. Eds), Academic Press, London, 1982.

[Cloksin 81] Clocksin, W.F., and C.S. Mellish, *Programming in Prolog*, Springer Verlag, New York, 1981.

[De Santis 85] De Santis M., "Logic Programming e Database: un ambiente di sviluppo adatto al trattamento dei vincoli di integrita'", Tesi di Laurea, Dip. Informatica, Universita' di Pisa, Jan. 1985.

[Furukawa 84] Furukawa K. et al., " Mandala: A Logic Based Knowledge Programming System", *Proc. of the Int. Conf. on Fifth Generation Computer Systems*, (ICOT Ed.), Tokio, 1984.

[Giannini 86] F. Giannini and E. Grifoni, " Programmazione Logica in Ambiente di Sviluppo Software: Data Base Logici come Data Base di Progetto", Tesi di Laurea, Dip. Informatica, Universita' di Pisa, Oct. 1986.

[Hubner 86] Hubner, W., and Z.I. Markov, "GKS Based Graphics Programming in Prolog" *Computer Graphics Forum*, Vol. 5, March, 1986, pp. 41-50.

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

[Kowalski 79] R. A. Kowalski, *Logic for Problem Solving*, Artificial Intelligence Series, (Ed. Nilsson, N. J.). North-Holland, 1979.

[Julien 82] Julien, S.M.P., "Graphical in Micro-Prolog"Res. Report DOC 8217, Imperial College, London, 1982.

[Lloyd 84] J. Loyd, *Foundations of Logic Programming*, Springer Verlag, New York, 1984.

[Logic 86] "Logic-Lab Reference - Release 2.1", SZKI, Budapest, Hungary, 1986.

[Mauro 85] F. Mauro, "Basi di Dati Logiche: un Approccio al Trattamento delle Transazioni", Tesi di Laurea, Dip. di Informatica, Universita' di Pisa, Nov. 1985.

[Mprolog 86] "MPROLOG Language Reference - Release 2.1", SZKI, Budapest, Hungary, 1986.

[OS 86] "OS - Specific details", SZKI, Budapest, Hungary, 1986.

[Pereira 86] Pereira, F.C.N., "Can Drawing Be Liberated from Von Neumann Style?", *Logic Programming and Its Applications*, van Caneghem, M., and D.H.D. Warren (Eds.), A.P.C., Norwood, New Jersey, 1986, pp. 175-187.

[Phogs 86] Biagi, B., and C. Montani,"PHOGS - Version 2.00 - Functional ovcerwiew and description",I.E.I, C.N.R., Pisa, 1986.

[Robinson 65] J. A. Robinson, "A Machine-Oriented Logic Based on the Resolution Principle", *J. Assoc. Comput. Mach.*, n. 12, 23-41, 1965.

[Shapiro 83] Shapiro, E. Y. and Takeuchi, A., Object-Oriented Programming in Concurrent Prolog, *New Generation Computing*, 1,1, 25-48, 1983.

[Spooner 84] Spooner, D. L., "Database Support for Interactive Computer Graphics", Proceedings SIGMOD, 1984, pp. 90-99.

[Sykes 85] Sykes, P., and R. Krishnamurti, "A Proposal for a Prolog Binding to GKS", Tech. Report EdCAAD, Univ. of Edinburgh,1985.

[Weller 76] Weller, D., and R. Williams, "Graphics and and Database Support for Problem Solving", ACM SIGGRAPH Computer Graphics,Vol. 10, Summer 1976, pp. 183-189.

8. *APPENDIX*

In this appendix we give the list of definitions of MPROLOG predicates that interface the application program with the PHOGS system, modified to be called from the MPROLOG environment.

The definitions regard the most significant PHOGS primitives that can be issued in EDIT and SGOP state. We are only interested in routines that change the state of the system in order to render them backtrackable.

This technique can be easily extended to the other four PHOGS states as shown in 6.2 .

SGOP

Control functions

```
open_segment (Name) :-
        assclause (editing_seg (Name, [])),
        assclause (editing_segparts (Name, [])),
        assclause (editing_segappl (Name, [])),
        _open_segment (Name);
        _close_segment (), _open_edit (Name),
        _delete_segment (Name), _close_edit (),
        fail .
```

This predicate definition creates in a temporary storage the facts that will be asserted in the logical database of segments when the segment will finally be closed. We use the predicate assclause to create the temporary fact because this predicate is backtrackable and its effect will be undone on failure. The second part of the definition discards the effect of the first part from PHOGS system if the segment transaction fails.

```
close_segment () :-
        _close_segment (),
        editing_seg (N, Desc),
        editing_segparts (N, Sub),
        editing_segappl (N, Appl),
        sgop_transaction (N, Desc, Sub, Appl );
        _open_edit (N), _delete_segment (N), _close_edit (), fail .

sgop_transaction (N, D , S, A) ←| , insert_f (segment (N,D)), insert_f (subsegment (N, S)),
      sgop_appl (N,A), | .

sgop_appl  (N,[] ) ← | .

sgop_appl  (N,[[ H | T ] | Rest]) ← | , insert_f (N(H,T), sgop_appl (N,Rest), | .
```

This predicate picks from the temporary storage the previously collected facts and inserts them in the logical database via the sgop_transaction. If the introduction creates a fail, the whole segment. is discarded from PHOGS system and the failure is propagated back to the calling segment_tr, that will also fail.

Output primitives

We define one predicate per output primitive. Here is an example:

```
text_2 (S) :-
        inquire_system_state_value (STATE),
        (STATE = "INOP",
                ( _text_2 (S);
                _delete_interactive_primitive, fail);
        STATE = "SGOP",
                ( _text_2 (S) , append_seg (editing_seg, [ text_2 (S) ] );
                _delete_segment_primitive, fail)
        ).
```

The predicate append_seg appends to the current editing_seg the list given as second parameter. It is backtrackable.
The other predicates are obvious.

Individual and generic attribute selection, modelling transformations, view operations.

The problems can be solved in the manner previously specified.

Segment content functions.

```
execute_segment (S) :-
        _execute_ (S) ,
        append_seg (editing_seg, [execute_segment (S) ] ),
        append_seg (editing_segpart, [ S ] ).

insert_application_data (Tag, Value) ←
        append_seg (editing_segappl, [ [Tag, Value] ] ).
```

Note that this predicate has no PHOGS counterpart.

```
copy_segment (Name) ←
        _copy_segment (Name),
        segment ( Name , Desc ) , append_seg (editing_seg , Desc ).
```

segment (Name , Desc) is a query to the logical database of segments.

```
copy_block (Name, From, To) ←
        _copy_block (Name, From, To),
        segment ( Name , Desc ) ,
        length (Desc, L), L >= To,
        get_elements (Desc, From, To - From + 1, Get_List),
        append_seg (editing_seg , Get_List ).
```

get_elements extracts a sublist from the list given as first parameter and unify the sublist with the last parameter. The sublist begins in the position passed as second parameter and has a length equal to the third parameter.

Other PHOGS functions.

They can be implemented using a combination of the techniques just shown.

EDIT

We are only interested in control functions. We have demonstrated that the proposed approach is feasible.

Control functions

```
open_edit (Name):-
        segment (Name, Desc),
        assclause (editing_seg (Name,Desc)),
        subsegment (Name, List),
        assclause (editing_segparts (Name, List)),
        bag_of (X, Name (X, _ ), L1),
        bag_of (Y, Name ( _ , Y), L2),
        merge (L1, L2, L3),
        assclause (editing_segappl (Name, L3)),
        _open_edit (Name);
        _close_edit (),
        fail .
```

We retrieve the segment description and its associated information in the logical database and we insert them in the temporary storage. merge is a predicate that given the two lists of the same length passed as first and second parameters creates one list of couples of elements corresponding to the third parameter.

```
close_segment ():-
        _close_edit (),
        editing_seg (N, Desc),
        editing_segparts (N, Sub),
        editing_segappl (N, Appl),
        edit_transaction (N, Desc, Sub, Appl );
        _open_edit (N), _delete_segment (N), _close_edit (), fail .

edit_transaction (N, D , S, A) ← | modify_f (segment (N,D), t ),
      modify_f (subsegment (N, S),t ), appl_transaction (N,A) | .

appl_transaction (N,[] )← | .

appl_transaction (N,[[ H | T ] | Rest]) ← | , insert_f (N(H,T), appl_transaction (N,Rest), | .
```