ALGRES: An Extended Relational Database System for the Specification and Prototyping of Complex Applications

F.Cacace[1], S.Ceri[4], S. Crespi-Reghizzi[1], G.Gottlob[3], G.Lamperti[2], L.Lavazza[2], L.Tanca[1], R.Zicari[1]

[1]Politecnico di Milano (Italy)
[2]TXT S.p.A. (Italy)
[3]Technical University of Wien (Austria)
[4]University of Modena (Italy)

Abstract

This paper illustrates by means of examples the functionalities offered by ALGRES: An advanced relational programming environment for the formal specification and rapid prototyping of data-intensive applications.

1. INTRODUCTION

A new trend in current database technology is to facilitate complex non standard application development and to provide reasoning capabilities on data stored in the database. New types of applications such as CAD/CAM, CASE, CIM, office automation systems, cartography, robot planning, call for new kind of database systems which are more powerful and flexible than the existing ones.

The introduction of these new application domain has resulted in a large effort in the scientific community to extend or even completely redesign database systems. These new generation database systems will support a variety of applications which are much more complex then the traditional business applications.

In particular, the new database systems need to offer several new concepts at the user level and more importantly, they need to be flexible enough to accomodate new requirements as they arise both from existing and new applications.

A non exhaustive list of the facilities that should be provided by a new database system can be summarized as follows:

i) **Data types and Operations.** Non standard applications require the management of more data types than the traditional ones. In fact, information such as maps, charts, images, pieces of text, structured documents, drawings, need to be represented as types in a database system. Present database systems do not support such new kinds of data. In some cases, (e.g. for text), data is stored in the database as a long string, with no further interpretation by the system. In other cases, these new types of information are stored in separate storage systems and are extracted when needed, and integrated with formatted records stored in the database. This results in an arduous and most of the time unreliable task. Problems arise in assuring data integrity and reasonable performance. Therefore the new generation database systems should provide new data types directly to the user, and more important they should allow the definition of new data types base on the built in ones. The introduction of new data types also implies the possibility of specifying new kind of operations on them. In fact, each new application area has in principle a specialized set of operations applicable to specific data types that the database system has to spport. Special operators are needed for manipulating long texts, or for comparing images, new operators such as a transitive closure are needed in processing recursive queries in knowledge based applications and so on.

ii) **Knowledge representation.** The new systems have to exploit the knowledge on the information stored in the database. This is needed in several knowledge based applications. The database system should then be able to offer different knowledge representation formalisms as they are needed from several applications or even from part of the same application. A multi paradigm knowledge representation seems to be a very promising approach.

iii) **Concurrency control and recovery mechanism** . Each new application area has also some specific requirements for concurrency control and recovery which are different from the traditional ones. For instance, handling long transactions manipulating large and complex structured objects requires and ad-hoc mechanism. Recovery as a traditional feature of databases has to be extended to handle complex objects with possible different versions.

iv) **Access methods and storage structures** . Each new application area

requires the development of appropriate access methods such as R-Trees used to access spatial data, KDB-trees and Grid-files. These methods allow fast access to an object in a multidimensional space and determine overlapping objects. Indeed, existing DBMS access methods are inadequate for many information types.

Several different approaches have been defined in the literature to enhance current database technology to include the facilities described above. A large number of research organizations, universities and industrial companies both in Europe, the USA and Japan are developing new database systems for handling new applications.

These efforts can be roughly classified as follows:

i) **Extensions to existing database systems.** This approach considers existing database systems, mainly relational ones, and provides several extensions to the basic theoretical model in order to enhance the power and flexibility of the system, but still retaining the original advantages of the relational model: Simplicity and a non procedural language for query processing.

The approach can be further divided into two classes when considering how these implementations are performed. In the first case the extensions are developed adding an extra layer on top of an existing system, in the second case, the database is augmented with new processing components, new algorithms, and new storage techniques from the "inside". The resulting differences is in the performance of such systems more than their functionalities.

Examples of such an approach are the systems based on nested relational models and the systems that allow user-defined abstract data types.

ii) **Semantic data models.** This approach tends to define a higher level data model which is closer to the needs of the applications than the relational model. In fact such models are not directly comparable with the relational model and with the ones defined at the previous point, but they have to be considered as high level conceptual models which might be implemented on top of the relational model. Although several semantic data models have been defined in the literature, few of them have been implemented.

iii) **Extensible database systems.** This approach tends to develop a kernel database system which can be easily extended to meet the demands of new applications. For such systems, the sharing of common functionalities is essential, and specific data structures and operations can be defined based on the common database.

Moreover, other fields of computer science have considerably influenced the development of the current database technology: namely AI and Programming Languages. In particular, several database architectures have been investigated and designed in order to integrate some of the paradigms usually used in AI technology and in Programming Languages. Running prototypes and commercial products have already been developed. These systems can be classified in two main classes:

i) **Object oriented database systems.** These systems make extensive use of work done in the object oriented ares, especially in the programming languages.

ii) **Deductive database systems.** These systems make extensive use of the work done in logic programming and AI.

In the rest of this section we will limit ourselves only to the nested relational and the deductive database approaches.

1.2. NESTED RELATIONS

One approach to the definition of a new data model tends to minimize the extensions to the relational model in order to retain much of its advantages. Within this context, relaxing the first normal form assumption for relations has been suggeste by several authors [MAKI77],[SCHE82],[ABIT84], [FIS83], [JSSC82], [SCSC86], [ROTH84]. Relaxing the first normal form corresponds to allow unnormalized relations normally called nested relations. A nested relation permits components of tuples to be relations instances themselves instead of atomic values. For nested relations, new types of perations have been defined with respect to the standard operations of relational algebra. Among them, two operations called Nest and Unnest have been extensively investigated by several authors [MAKI77], [SCHE82], [ABIT84], [FISH83], [ROTH84]. Essentially, the Nest and Unnest operators permit to pass from flat relations to nested hierarchical ones and vice versa. In this way, it is therefore possible to model all cases where data is by its definition a tree structure (e.g. office forms). Several theoretical studies have been done on the nested relational model. The basic approach is to extend the results of relational theory to the nested model.

Recently the nested relational model has been extended and modified to capture a few characteristics not covered by the original model. In particular, the following extensions have been proposed:

- New static attribute types (tuples, sets, multisets and lists) with their respective operators have been added [CERI88], [DADA86], [GUET87], [PIST86].

- Ordering among tuple components has been considered [GUET87].

- Duplicates are allowed for multisets and lists [PIST86], [CERI88], [GUET87].

- Keys on multivalued attributes and surrogates have been introduced [DADA86].

- New null values have been introduced [ROTH85], [GUET87].

- More powerful restructuring operations than Nest and Unnest have been defined [ABIT86], [GUET87].

- A Closure operator has been introduced. This operator applies on algebraic expressions and iterates the evaluation of the expression until the result reaches a fixpoint [CERI88], [LINN87].

- Many-sorted algebras rather than one-sorted algebras have been also proposed [GUET87].

- Extended definitions of the SQL language have been defined [PIST86], [ROTH85].

Notable models that include some of the above features are: The extended NF^2 model developed at IBM Heidelberg [DADA86], the NST model [GUET87] developed at IBM San Jose', the ALGRES model developed at Politecnico di Milano [CERI88]. All of them served as a formal base for the implementation of running prototypes. Results of experimentation of such systems with real case application are under investigation.

A major difference in the above systems is the architecture. The extended NF^2 system is a new database system that provides a user interface based on extended nested relations. The NST system has been implemented on top of an existing relational database. ALGRES instead is interfaced with a commercial database system, but is not implemented on top of it.

1.3. DEDUCTIVE DATABASE SYSTEMS

The deductive database systems are a rule-based paradigm which integrate in the database the knowledge shared by several applications. The knowledge included in the system is expressed as general rules which define deduced or derived facts from the facts stored in the database. While the facts stored in the database constitute the extensional database, the rules define the intensional database. The most familiar approach to integrate a rule base language is to use logic programming language such as Horn Clause language.

Since the notion of deductive databases is well understood [GALL78], [REIT78],[GALL84], systems designers have started to build practical deductive DBMS. In [CERI86] is shown how to translate logic programs for deductive database systems into relational algebra. This translation can be used to transform logical rules and queries into algebraic expressions, which can be computed more efficiently. Since a goal of the ALGRES language is to investigate the integration of the relational systems and the deductive database systems, it provides a **fixpoint** operator (see Section 2) which allows to evaluate recursive queries in a buttom-up fashion. An interface which performs the translation of DATALOG programs into ALGRES ones is already operational.

2. THE ALGRES SYSTEM

ALGRES is a vehicle for the integration of two research ares: The extension of the relational model to deal with complex objects, and the integration of databases with logic programming.

The ALGRES system consists of:

i) A data model which incorporates standard elementary types (character, string, integer, real, boolean) and type constructors such as RECORD, SET, MULTISET, and SEQUENCE. An ALGRES object has a schema which consists of a hierarchical structure of arbitrary depth built through the above type constructors.

ii) An algebraic language, called ALGRES-PREFIX, which provides:

- Classical relational operations such as selection, projection, cartesian product, union, difference (and derived operations, e.g. join) suitably extended to deal with multiple-levels objects and new types.

- Restructuring operators (nesting, unnesting, extension of a tuple), which modify the structure of a complex objects. The nesting operators allow the partitioning of relations with respect to the value of some attribute (like, for example, the *aggregate formation* operator defined in [Klug82]).

- Facilities for query-language applications. These are: Evaluation of tuple functions and aggregate functions; tuple updates; built-in functions to evaluate the cardinality of a collection, the number of attributes in a collection, etc; operators to handle ordered collections.

- Operations for type transformation, which enable the coercion of any type

constructor into a different one.

- A fixpoint operator which applies to an algebraic expression and iterates the evaluation of the expression until a certain predicate is satisfied. This operators is quite flexible and allows to emulate all the different "closure" operators proposed to evaluate recursive queries. The fixpoint operator can therefore be specialized to optimize the evaluation of particul classes of recursive queries, as proposed, for example in [Agra88]. Moreover, the fixpoint operator derives extra power from the presence of aggregate functions, nested relations, and ordered collections: These features are all present in ALGRES and open interesting possibilities of evaluation and optimization of logic programs and deductive databases programs.

The fixpoint operator has the following structure:

FIXPOINT [*Temp* := TRANSF (*Op*) ;
PRED (*Temp*, *Op*);
COMB (*Temp*, *Op*)] *Op*

where *Op* is the operand and *Temp* is a temporary object whose scope is limited to the **FIXPOINT** operation. TRANSF, PRED, COMB are three ALGRES expressions: TRANSF computes a new instance of *Temp* from *Op*; PRED is the exit condition from the loop; COMB creates a new instance of *Temp* from *Temp* and *Op*. The three inner statements of the **FIXPOINT** operators are iteratively executed until *Pred* is not satisfied. Note that, like every ALGRES statements, the **FIXPOINT** operator does not actually modify the original *Op* object; a copy of *Op* is used in the evaluation of the **FIXPOINT**, and it eventually becomes the final result.
The **FIXPOINT** operator then corresponds to the following PASCAL-like program:

```
end : = false;
Result : = Op ;
repeat
    Temp  : = TRANSF ( Op ) ;
    if PRED ( Temp, Op )
        then end : = true;
        else  Result : = COMB (Temp, Result);
until end
```

Although being much more general, the fixpoint operator can be used to simulate the **transitive closure** to evaluate recursive relational expressions: R = f(R).
The least fixpoint of this expression is a relation R* such that R* = f(R*) and R*⊆ S for any S satisfying S = f(S). The least fixpoint can be obtained using the **FIXPOINT** operator as follows:

FIXPOINT [*Temp* := f(*R0*) ;
NOT (*Temp* ⊆ *R0*);
UNION *Temp R0*] *R0*

where *R0* is an empty collection with the required schema. Chandra and Harel [ChHa82] have shown that the relational algebra (without the set difference) together with a least fixpoint evaluator becomes equivalent to the function-free Horn clause programs. The **FIXPOINT** operator has a wider range of applicability than the computation of the least fixpoint. It can also be used to simulate **for-next** iterative statements, due to the presence of a generic PRED function. Moreover, the presence of the COMB expression allows the evaluation of non-monotonic or non-"growing" transformation over algebraic espessions. As a concluding remark, it should be pointed out that the expressive power of the **FIXPOINT** operator

makes possible to optimize the buttom-up evaluation of logic clauses and also to implement evaluation strategies more complex than "pure" depth-first or breadth-first.

A goal of the language is to limit the loop complexity and to suppress linked list structures, which are the two major sources of programming costs.

The ALGRES systems constitute a relational programming environment, in fact ALGRES is integrated with a commercial relational database system, but it is not implemented on top of it. This is for efficiency reasons: Execution of our operations would be expanded into a long chain of database operations, causing very long response time.
We assume that the ALGRES system will be typically applied to complex data structure of medium size, rather than to very large databases: Examples are CAD and engineering systems, and some office automation systems. As a consequence of these assumptions, ALGRES programs operate on main memory data structures, which are initially loaded from an external database, and returned to mass memory after manipulation. The abstract machine supporting the extended data model and its operations, named RA (Relational Algebra), reflects these assumptions: Only a fraction of the database is supposed to be present in main memory at any time, and speed (rather than memory) is the limiting factor.

The core of the ALGRES environment (fig. 1) consists of the ALGRES to RA translator and RA interpreter. RA instruction set provides traditional algebraic operations on normalized relations as well as special features; it has been carefully designed so as to provide instructions that can be efficiently executed over the flat subobjects stored in the main memory algebraic machine. The translator and RA machine are implemented both on a SUN workstation and Microvax, with the Informix[1] Database management system providing permanent storage of objects.

The ALGRES system is intended to compare favorably with other existing productivity tools or high-level languages which have been successfully used for rapid prototyping such as SETL and Prolog.

Applications in ALGRES can be programmed using directly the ALGRES-PREFIX and the C interface (ALICE). However, to provide the programmer with a multi-paradigm, easy-to-use programming environment, we have designed ALGRESQL, an extension of the SQL language which enables a "structured, English-like" query and manipulation language. Programs in ALGRESQL are translated into ALGRES-PREFIX before execution. We are also considering the use of the language DATALOG embedded into ALGRESQL, for dealing with recursive queries. DATALOG is a clause-based logic programming language developed in the database community, syntactically similar to Prolog, designed for retrieving information from a flat relational database system. Finally, we interfaced both ALGRESQL and DATALOG with the "C" programming language in order to exploit existing libraries, and to access the facilities of ALGRES from conventional programs.

The ALGRES project is broad-spectrum, and to give a specialized description of each of its parts is outside the scope of this paper; the interested reader is referred to [CERI88], [CERI88a], [GoZi88].

[1] Informix is a trademark of Informix Co.

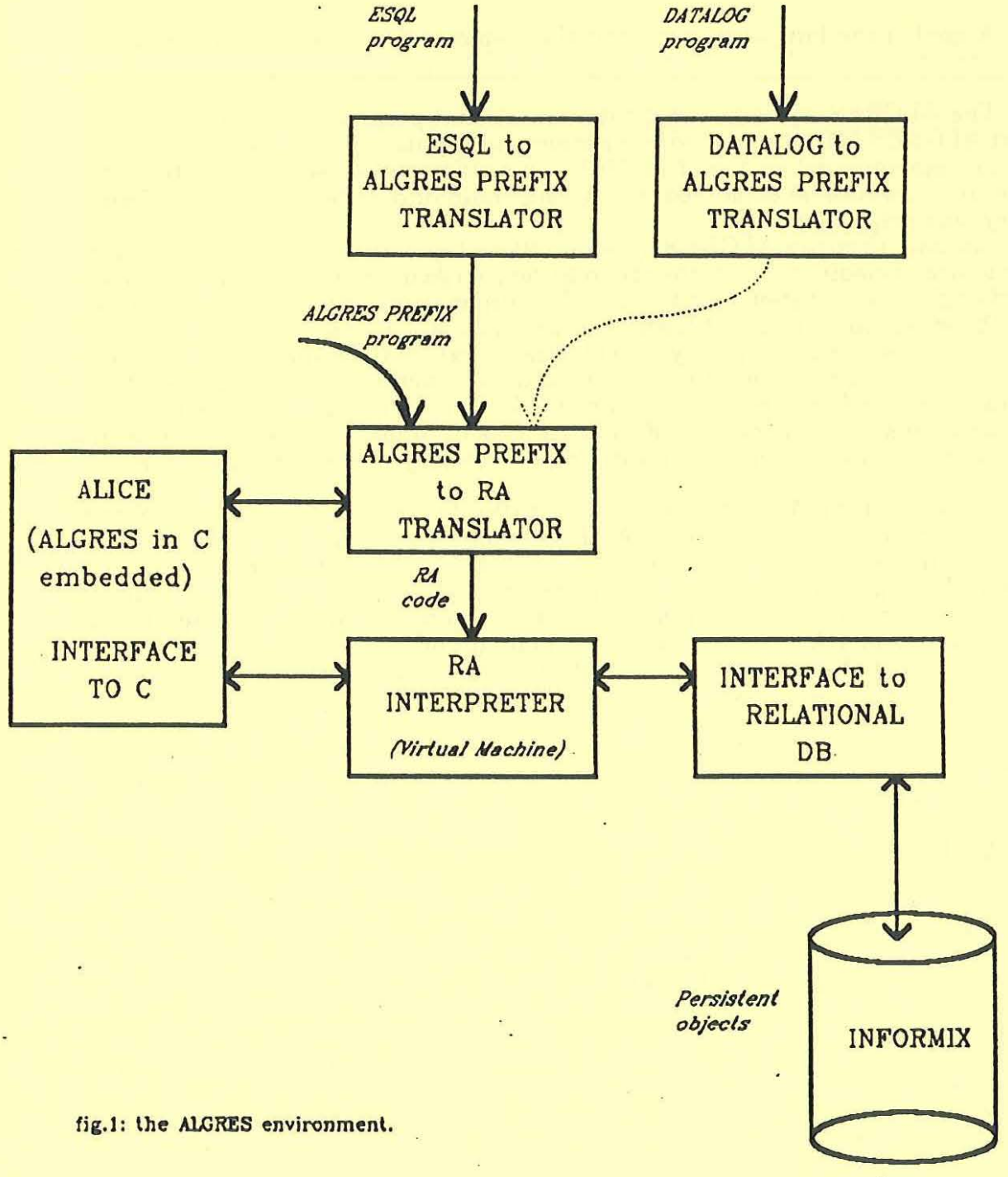

fig.1: the ALGRES environment.

3. A COMPARISON BETWEEN ALGRES AND RELATED APPROACHES

In this section we briefly analyze the main features of the ALGRES language, and compare them with current Relational Database Systems and Data Bases Programming Languages (DBPL).

3.1. ALGRES DESIGN GOALS

The ALGRES language includes several important extensions to the Relational Algebra, which make it a suitable tool for many different applications. In fact, ALGRES can be used as a **query language** for complex databases, since it allows to expresse concisely powerful queries on complex objects; or it offers its rich data modeling capabilities to become a **rapid prototyping** language for those applications (CAD-CAM databases, office automation) where modeling and restructuring of complex objects play an important role. ALGRES includes a flexible *closure* operator to express recursive queries: Depth-first and breadth-first strategies are both efficiently evaluable. It can therefore be used to support **deductive databases** and **logic programming language** in an efficient and well formalized way. Moreover the ALGRES operators provide a unified language-based solution for problems which involve **incremental computations**, advanced editing environments and speadsheet programs (see for example [YeSt88]).

A goal of the ALGRES language is to limit loop complexity and therefore it provides the possibility to build complex algebraic expressions from simple ones. Extended Relational Algebra operators can be applied on intermediate results and complex expressions can be nested one into another to perform long computations with a single statement and without using auxiliary variables.
Moreover, ALGRES programs can be structured by means of **procedures** and **macros**. The latter gives the possibility of re-using the same code on different complex objects (**polymorphic procedures**). We illustrate this with an example. Suppose we wish to define a new algebraic function *Self Join*, which makes the cartesian product of an object by itself and then selects the tuples that have two specified attributes with the same value. We can write:

SUBFUNC *Self_Join* (R{A1, A2})

BODY *Self_Join*: **JOIN** [A1 = A2] R R

where A1 and A2 are two generic attributes of the parameter, the relation R. Notice that the schema of R, A1, A2 are left unspecified and can be different on different calls of *Self_Join* (provided that A1 and A2 have the same schema).

3.2. LANGUAGE FEATURES

The ALGRES definition of complex objects is based on four type constructors: *Record, set, multiset, sequence.* The **record** and **set** type constructors allow the programmer to distinguish between the type of an **object** and that of a **collection** of the same object. Object-oriented languages and models (such as *Adaplex* [Smith83], *Taxis* [Mylo80] or *Galileo* [Alba83]) include declarations of **entities** with associated **extensions**. However it is not generally possible to declare *different classes over the same object type* as it is in ALGRES. Consider the classical example of describing the inventory of a manifacturing company (adapted from [Atk87]). The database represent the way certain parts are manufactured out of other parts. Since the manufactured parts may themselves be subparts in a further manufacturing process, the relationship between parts is hierarchical but it is a directed acyclic graph rather than a tree. The Adaplex data definition for *Parts* would be:

database *Manifacturing* **is**

type ...
type *Part*;
type ...

type *Part* **is entity**
Name : String(1..16);
Components : **set of** *Part*;
end entity;

Two entity values are considered distinct even if all their defined attributes have the same value. Notice that with this declaration we produce both the type and the extent. There is no clear distinction between the underlying type of the entity and the associated extension. The extent is bound to the entity, then **we cannot have two collections of the same type**. The usefulness of this limitation is uncertain. In ALGRES the declarations of the type and of the collection can be made distinct. The programmer can either define a separate type for the tuple or not, as the following equivalent declarations show:

DEF *Parts* **SET OF** *Part*: (*Name*: string,
Components: **MULTISET OF** (*C_Part*: string))

DEF *Parts* **SET OF** : (*Name*: string,
Components: **MULTISET OF** (*C_Part*: string))

The record type *Part* of the first declaration can be used elsewhere in the program, making possible the existence of many collections of *Part*, which is considered the same type. The same approach is used in PASCAL/R [Schm77], where one starts by constructing a record type and using that to construct a relation. But in PASCAL/R the consituent fields of records that form the basis of relations are limited to certain types and this should be regarded as a failure of type completeness: It would in fact be useful to have a **relation of** *integer*, that is, to have a generic set type without explicitely having to name the underlying record. Moreover, in the relational model of PASCAL/R complex objects are not allowed: To describe the hierarchy of parts we have to define two relations. To find the components of a part we must write a piece of code which computes the join of these two relations. In ALGRES the association between a part and its components is made by the data definition.

An advantage of the semantic data model approach over ALGRES is that the directed aciclyc graph of the example can be represented through a recursive definition, like that of Adaplex; in ALGRES we must explicitely represent it with a two-level tree. Moreover, languages like Taxis, Galileo or Adaplex use the notion of **inheritance** to define a hierarchy of types and represent variant records. In ALGRES variant records must be explicitely represented by different complex objects or by using *special* attributes.

Duplicates and ordering can be handled with the introduction of the multiset and sequence type constructors.
It is argued in the literature that many recursive queries can be expressed and evaluated more conveniently by using ordered sets with additional operators for them. (for example the problems of graph traversal, etc.).

4. EXAMPLES OF USE OF ALGRES

This section contains several examples of use of the ALGRES system. In particular, the first three examples show the facilities offered directly from the ALGRES system; the last example illustrate the use of ALICE, the integrated C environment with ALGRES.

All examples have been previously tested and executed. The code at the moment uses a concise, Assembler-like notation for algebraic operatorss. For the sake of readability we list the corresponding full names of the operators in Table 1.

TABLE 1.

PJ	PROJECT
SL	SELECT
DS	DELETE STRUCTURE
CS	CREATE STRUCTURE
EX	EXIST
NJ	NATURAL JOIN
MT	MULTISET TRANSFORM
TE	TUPLE EXTENSION
AF	AGGREGATE FUNCTION
JN	JOIN
CONTEQ	CONTAINED OR EQUAL
DF	DIFFERENCE
UN	UNION
FP	FIXPOINT

4.1. TEACHING SAMPLE PROGRAM

This example shows how to managea simple databases of courses offered in a university and related time-tables. The application consists of two objects definitions, *teachings* and *specific_teachings* , and six complex queries on them.

```
DEF teachings: { teaching_name: string,
                 sections: { section: char,
                             teacher: string,
                             time_table: { day: string, hour: integer,
                                           schoolroom: string }}}

teachings ← READ [ teachings_inst: VS [ teachings ] ]

DEF specific_teachings: { teaching_name: string, section: char }

specific_teachings← READ [ spec_inst: VS [ specific_teachings ] ]
```

The object *teachings* is a nested relation with a three nesting depth and

schema shown in fig. 4.1. The object and its nested subrelations are **sets**. Every tuple in *teachings* contains two attributes: *teaching_name* (single-valued attribute) and *sections* (set-valued attribute[1]): Every teaching can be divided into one or more sections. *Sections* is a set with three attributes: *section* (an identifier), *teacher*, *time_table*. In turn, *time_table* is a set with three attributes: *day*, *time*, *schoolroom*.

The second object, *specific_teachings*, shown in Fig. 4.2, is a classical relation with two (simple) attributes: *teaching_name* and *section*.

All queries in Appendix are coded in ALGRES. Every query is specified by an ALGRES statement.

Query 1.1 - Find and display the time-table of all the teachings contained in *specific_teachings*.

DISPLAY [ ] **PJ** [teaching_name, section, time_table]
NJ specific_teachings **DS** [sections] teachings

First, we unnest *teachings* of one level by deleting the internal relation *sections*. We obtain an intermediate object with the schema shown in Fig. 4.3.
We **JOIN** this object with *specific_teaching* on attributes *teaching_name* and *section*. We then **PROJECT** on the attributes we are interested: *teaching_name*, *section*, *time_table*.

Query 1.2 - Find and display the time-table for each schoolroom.

rooms_time ← **CS** [roomtime: { teaching_name, section, teacher,day, hour }]
DS [sections]
DS [time_table] teachings

DISPLAY [ ] rooms_time

This query takes as input the object *teachings* as shown in Fig. 4.1. We want to obtain the time-table of each schoolroom from the time-table of each teaching. This is obtained by unnesting the two internal relations *sections* and *time_table*: we then create a new nested relation *roomtime* from the attributes of the flattened relations with the operation **CREATE STRUCTURE** [*roomtime:* { *teaching_name, section, teacher, day, hour* }]. The new object *rooms_time* created by this operation contains two attributes: *schoolroom* and *roomtime* and has the schema shown in Fig. 4.4.

Query 1.3 - Find and display the time-table for each teacher.

teachers_time ← **CS** [teachtime: {teaching_name,section,
day,hour,schoolroom}]
DS [sections]
DS [time_table] teachings

DISPLAY teachers_time

We obtain the object *teachers_time* from *teachings* in a similar way as in the previous query. *Teachers_time* shows the time-table for each teacher and it is obtained by unnesting *time_table* and *sections* in *teachings* and nesting the new relation *teachtime* with attributes *teaaching_name, section, day, hour, schoolroom. Teachers_time* has the schema shown in Fig. 4.5.

[1] In the sequel, we denote single-valued attributes as simple, and set (lists, mulrisets)-valued attributes as complex. In the figures, complex attributes have double border width

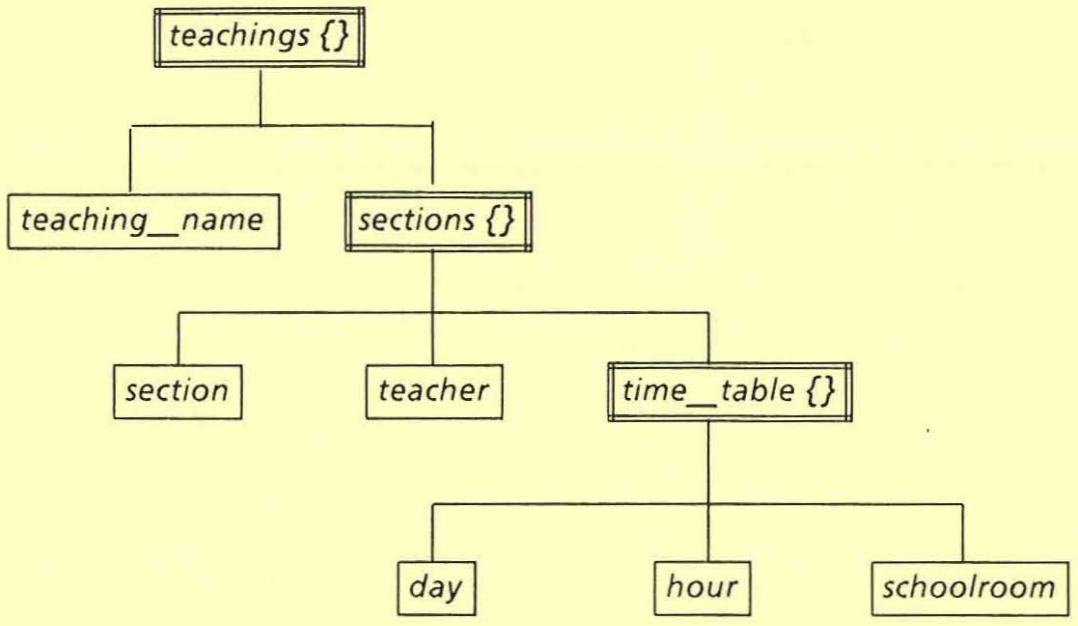

Fig 4.1- Schema of the object *teachings*

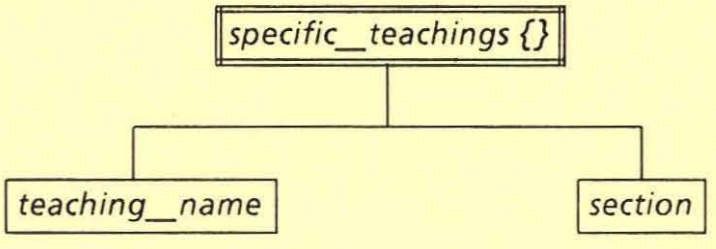

Fig 4.2- Schema of the object *specific_teachings*

Query 1.4 - Find and display all free schoolrooms on Thursdays at 9.00 a.m.

free_rooms ← **SL** [**NOT EX** [(day = "thur" **AND** (hour = 9)] roomtime]
rooms_time

In this query we use *rooms_time*, the object created by **Query 1.2**, to select the schoolrooms which are free at 9.00 a.m. of a generic Thursday.

The last two queries are:

Query 1.5 - Find out if there is any schoolroom which is used for different teachings at the same moment.

room_overload ← **SL** [**NOT** (**EQUAL** (**MT**[roomtime] **PJ** [day,hour] roomtime;
PJ [day,hour] **MT** [roomtime] roomtime))]
rooms_time

DISPLAY room_overload

Query 1.5 - Find out if there is any teacher which has to teach different teachings at the same moment.

teacher_overload ← **SL** [**NOT** (**EQUAL** (
MT [teachtime] **PJ** [day,hour] teachtime;
PJ [day,hour] **MT** [teachtime] teachtime))]

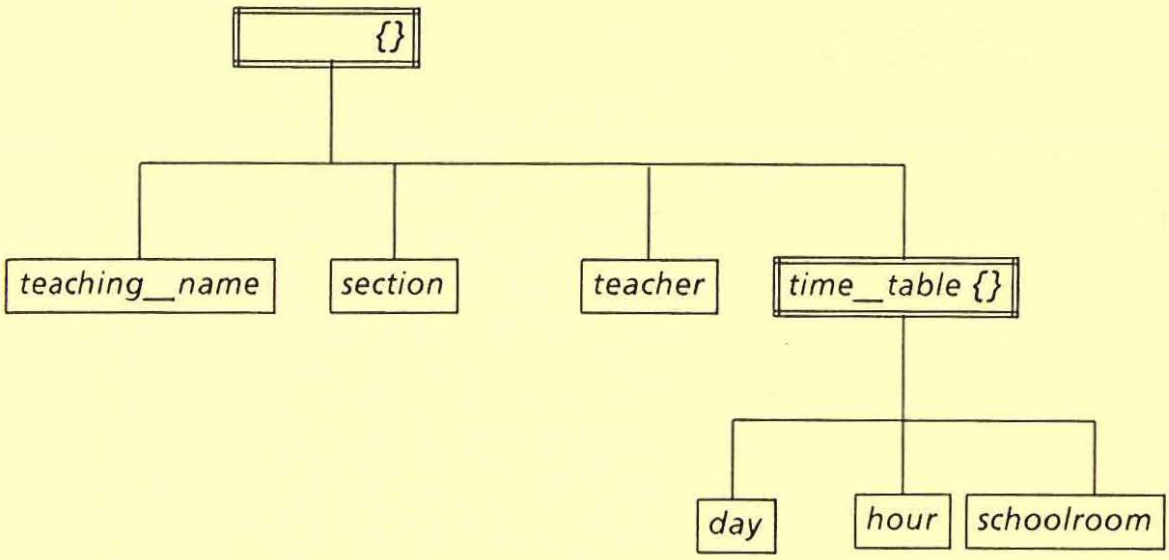

Fig. 4.3 - Schema of the object obtained by Unnest (*sections*) *teachings*

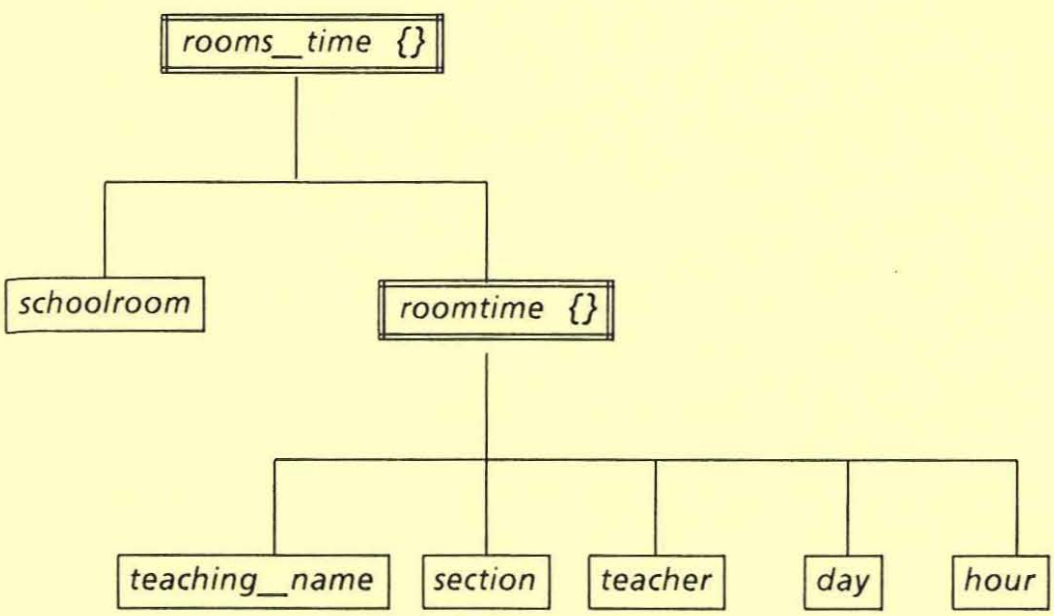

Fig. 4.4 - Schema of the object *rooms__time*

teachers__time

DISPLAY teacher__overload

4.2. FIXPOINT SAMPLE PROGRAMS

These following three examples use the ALGRES **FIXPOINT** (FP) operator. It

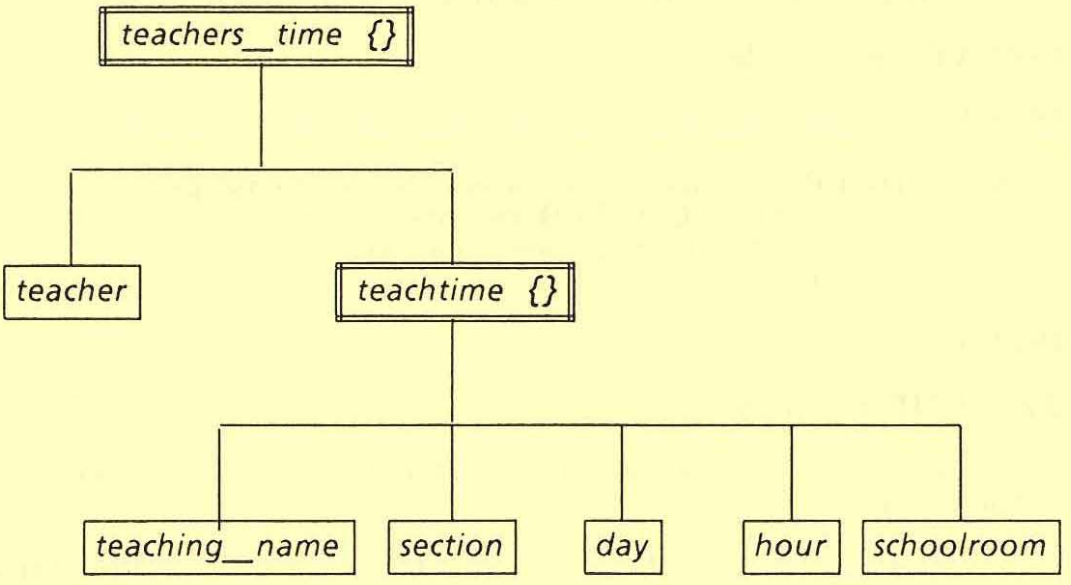

Fig. 4.5 - Schema of the object *teachers_time*

should be pointed out that the **FIXPOINT** is the only control structure of the ALGRES language and that it allows to implement recursion.

The following examples cannot be computed using a normal query language since they require the full computational power of a normal programming language.

4.2.1- ANCESTORS

The problem is to find all the ancestors of a set of persons (*persons*), given the object *relationship* containing the father-son relationship (Fig. 4.6).

The ancestors are obtained by recursively joining *relationship* and *persons* . In the **FIXPOINT** operator, then, the TRANFS operation is a **NATURAL JOIN** between *relationships* and *persons*; the PRED condition is that the new partial result *parents* is contained within the accumulated result *persons*; the COMB operation is the **UNION** of *persons* and *parents*.

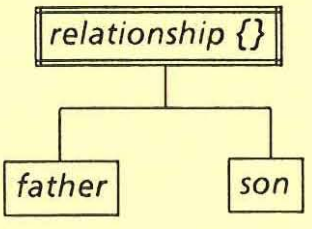

Fig. 4.6 - Schema of *relationship*

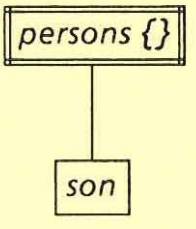

Fig. 4.7 - Schema of *persons*

DEF relationships : { parent: **string**, son: **string** }

DEF persons : { son: **string** }

relationships ← **READ** [rel_inst: **VS** [relationships]]

```
persons ← READ [ person_inst: VS [ persons ] ]

DISPLAY [ ] relationships

DISPLAY [ ] persons

ancestors ← DF FP [ parents := PJ [ parent ] NJ relationships persons ;
                    NOT ( CONTEQ (parents; persons ) );
                    UN parents persons ] persons
                  persons

DISPLAY [ ] ancestors
```

4.2.2- COMPONENTS

Now we deal with the classical problem of finding elementary components of a set of products.

We define the object *products* as a set of *product* (the product name) and *components* (an inner set of constituent parts *component*), with schema shown in Fig. 4.8. The object *input* (Fig. 4.9) contains the set of products that we want to examine. The use of the **FIXPOINT** operator follows the same pattern as our preceding examples: We have an auxiliary object *first_components* which contains the set of sub-components computed at each iteration. The only difference in the body of the **FIXPOINT** operator is that we perform an **AGGREGATE FUNCTION** of **UNION** over all the *components* sets which are present in the **NATURAL JOIN** between *input* and *products*, to obtain the single set *first_components*.

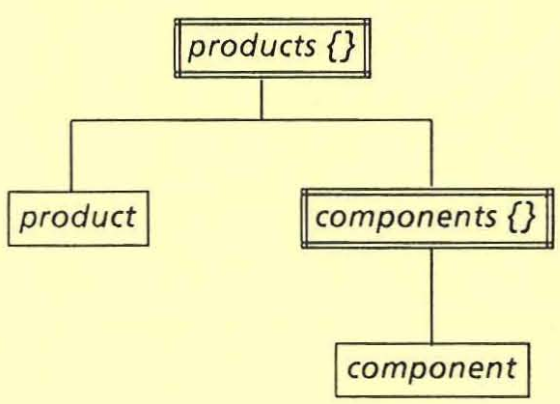

Fig. 4.8 -Schema of *products*

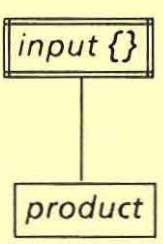

Fig. 4.9 - Schema of *input*

```
DEF products: { product: string, components: { component: string } }

DEF input: { product: string }

products ← READ [ product_inst : VS [ products ] ]

input ← READ [ input_inst : VS [ input ] ]

DISPLAY [ ] products

DISPLAY [ ] input
```

```
elementary_components ← DF FP [ first_comp := AF [ UNION / components ]
                                                  NJ input products ;
                                    NOT ( CONTEQ ( first_comp; input ) ) ;
                                    UN first_comp input ] input
                                input

DISPLAY [ ] elementary_components
```

4.3. TEACHERS FINDER

This is an example of an ALICE program. The program shows how ALICE can be used to interactively retrieve information from an ALGRES database. The example uses the same database defined for the TEACHINGS SAMPLE PROGRAMS. We want to look in the time-table to know, given a day and an hour, where a certain teacher is, .

The ALICE program defines his own C variables to store the values of ALGRES objects. It also contain two ALGRES instruction: the first one defines the object *teaching*, the second one loads from a file an instance of it. The ALICE program uses the special **GETOBJ** statement to transfer the values of ALGRES objects into its simple-valued C variables, tuple by tuple.

The **GETOBJ** statement is placed into the body of a C **while** statement, thus one can repeat the query more than once, changing its parameters each time. The program also adds some messages that are not allowed in the ALGRES environment; for example if one type a name that is not contained in the collection, the program answers that the specified person does not teach here.

In the example, the **GETOBJ** loads the whole *teachings* object, without performing projections on the interesting attribute. This feature is not mandatory, since we could have reduced the set of attributes examined in the data acquisition cycle of the **GETOBJ** statement.

```
#include "/usr/include/stdio.h"
main()
{
char buf [100];
char buffer [80];
chat t_name [40];
char sec;
char teach [40];
char teach_des [40];
char day [40];
char day_des [40];
int time_tab;
int hour_des;
char schroom [40];
int cont;
int found_teach, found_hour;
int badg, bado;
char resp = 'y';
char trace = 'n';
int i = 0;
int j = 0;
int k = 0;

ALGRES[];
```

```
DEF teachings:{
                    teaching_name:string,
                    sections:{
                                section: char,
                                teacher: string,
                                time_table:{
                                            day: string,
                                            hour: integer,
                                            schoolroom: string }}};

printf("\n");
printf("loading file teachingsdat...\n");
printf("\n");
teachings ← READ [teacist: VS [teachings] ← "teachings.dat"];
found_teach = 0;
found_hour = 0;
badg = 1;
bado = 1;
system("clear");
printf("Would you like to trace data acquiring ? (y/n)\n");
    gets(buf);
    sscanf(buf,"%c",&trace);
while (resp = = 'y')
    {
    i = 0;
    j = 0;
    k = 0;
    system("clear");
    printf("**********************************0);
    printf("***                              ***0);
    printf("***   TEACHERS FINDER            ***0);
    printf("***                              ***0);
    printf("**********************************0);
    printf("Desired teacher:.\n");
    gets(buf);
    sscanf(buf,"%s",teach_des);
    printf("day:.\n");
    gets(buf);
    sscanf(buf,"%s",day_des);
    printf("hour:.\n");
    gets(buf);
    sscanf(buf,"%d",&hour_des);
    if(strcmp(day_des,"sun") = = 0)||(strcmp(day_des,"sat) = = 0)) badg = 0;
    if(hour_des < 8)||(hour_des > 19)) bado = 0;
    if(badg = = 1)&&(bado = = 1))
        {
        GETOBJ teachings VIA
        {                                   /* teachings acquiring cycle */
        t_name DOC
                    if(trace = = 'y') printf("***TRACE:\t%s\n",t_name);
                END,
            {                               /* sections acquiring cycle */
            sec,
            teach DOC
                {
                if (trace = = 'y')
                    {
                    printf("***TRACE:\t\t%c\n",sec);
                    printf("***TRACE:\t\t%s\n",teach);
                    }
                if(strcmp(teach,teach_des) = = 0) found_teach = 1;
```

```
            }
            END,
        { day,                          /* time_table acquiring cycle */
        time_tab,
        schroom
          DOC
            {
            if(trace==='y')
                {
                printf("***TRACE:\t\t\t%s\n",day);
                printf("***TRACE:\t\t\t%d\n",time_tab);
                printf("***TRACE:\t\t\t%s\n",schoolroom);
                }
            if(!found_teach)
            {
                if(trace==='y')
                    {
                    printf("\n### TRACE: break: exit from time_table\n\n");
                    break;
                    }
            }
            if(strcmp(day,day_des)==0 && time_tab==hour_des)
                {
                printf("%s is in schoolroom %s\n\n",teach_des,schroom);
                found_hour=1;
                printf("\n### TRACE: break: exit from time_table\n\n");
                break;
                }
            }
            END
        DOC
        if(found_teach)
            {
            printf("\n### TRACE: break: exit from sections\n\n");
            break;
            }
        END
        }
    DOC
    if(found_teach)
        {
        printf("\n### TRACE: break: exit from teachings\n\n");
        break;
        }
    END
};

if(!found_teach) printf("%s doesn't teachs here\n\n",teach_des);
if(found_teach && !found_hour) printf("%s is in his office\n\n",teach_des);
}
else printf("I can't find %s\n\n",teach_des);
printf("Hit 'y' to continue, 'n' to stop\n");
gets(buf);
resp=*buf;
found_teach=0;
found_hour=0;
badg=1;
bado=1;
system("clear");
    }
```

}

4.4. ALGRES INTERPRETER SIMULATOR

In this example we use the ALICE capabilities to simulate an interactive interpreter for the ALGRES language.
We show how ALICE can be used not only to retrieve information from ALGRES database, but also to execute interactively new ALGRES statements. That is, queries on the ALGRES database can be changed at run-time without having to change the code.

The program starts by loading three instances of the following objects of our database:

students

institute_planning

exams_inscription_list

These objects are a simple database on which the user can perform some queries using the ALGRES language.
Some examples of queries you can ask are:

SELECT [*student* = "Benjamin"] *students*.

PROJECT [*student*] *students*.

This result is obtained by parametrizing an ALGRES **DISPLAY** statement with a C buffer (defined as an array of chars) containing the ALGRES statement to be executed.

```
#include "/usr/include/stdio.h"
main()
{
int a = 1;
char *result;
char buf[100];

ALGRES[];

DEF students:  { student: string,
                 inscription_year: integer,
                 individual_planning: {
                              year: integer,
                              choosen_teachings: {
                                          chosen_teaching: string } },
                 done_exams: {
                              exam_name: string,
                              grade: integer } };

DEF institute_planning: {
                         year: integer,
                         proposed_teachings: {
                                  proposed_teaching: string,
                                  must_be_done: bool,
```

```
                                        preceding_teachins: {
                                            preceding_teaching: string } };

DEF exam_inscription_list: {
                                    student: string,
                                    teaching: string,
                                    exam_date: string };

printf("\n");
printf("loading file istplan.dat ...\n");
printf("\n");
institute_planning ← READ [ reoist: VS [institute_planning] ←
"istplan.dat" ];
printf("\n");
printf("loading file inscription.dat ...\n");
printf("\n");
exam_inscription_list ← READ [ examist: VS [exam_inscription_list] ←
"inscriptions.dat" ];
printf("\n");
printf("loading file students.dat ...\n");
printf("0);
students ← READ [ studist: VS [students] ← "s.tudentsdat" ];
while(a)
    {
    system("clear");
    printf("**********************************\n);
    printf("***                              ***\n);
    printf("***    Now, enter your query ...   ***\n);
    printf("***                              ***\n);
    printf("**********************************\n);
    printf("\n");
    flush(stdout);
    gets(buf);
    }
ENDALGRES;
}
```

5. CONCLUSIONS

The concepts, techniques and tools necessary for the design and the implementation of the future database systems are expected to result from cross fertilization in several ares of Computer Science. One key area is Artificial Intelligence which provides knowledge bases techniques for reasoning, problem solving and question answering. An other area is Programming Languages and especially Object Oriented Languages which provides powerful languages to handle complex applications, that is, with numerous different data types. Finally, the Database field is concerned with the efficient management of large amounts of persistent, reliable and shared data. The issue of partially integrating these different fields has received wide attention from the scientific and technical community.

The ALGRES system is an attempt to unify two reasearch areas: The extension of the Relational Model to deal with complex objects and Logic Programming. As such it can be used as a tool for rapid prototyping and sperimentation of new data-intensive applications.
Several example have been given in the paper to show the capabilities of the system.

The use of the system with industrial applications will give us the necessary experience and point out the advantages and the real limits of this approach.

The ALGRES system is operational on SUN workstations (SUN-2, SUN-3) with Berkeley UNIX[1] 4.3 BSD Operating System, and on VAX/780 with ULTRIX[2] Operating System interfacing an INFORMIX Database System.

A distribution kit of the ALGRES system is available for non-profit use directly from the authors at a nominal fee.

For further information please write to:

Prof. Roberto Zicari
Dipartimento di Elettronica
Politecnico di Milano
Piazza Leonardo da Vinci, 32
20133 Milano, Italy

phone: +39-2-2399-3523
fax: +39-2-2399-3587

AKNOWLEDGMENTS

The ALGRES project is mainly supported by the EEC-Esprit Project 432 "Meteor", and also partially by the italian CNR, MPI, and Rank Xerox University Grant Program

REFERENCES

[ABIT84] S.Abiteboul, N.Bidoit, *"Non First Normal Form Relations to Represent Hierarchially Organized Data"*, ACM 1984, 191-200.
[ABIT86]S.Abiteboul, N.Bidoit, *"Non First Normal Form Relations: an Algebra Allowing Data Restructuring"*, JCSS, 1986.
[Agra88] R. Agrawal, *"Alpha: An Extension of Relational Algebra to Express a Class of Recursive Queries"*, in IEEE Transactions on Software Enfineering, Vol. 14, N. 7, July 1988.
[Alba83] A. Albano, L. Cardelli, R. Orsini *"Galileo: A Strongly Typed, Interaptive, Conceptual Language"* Tech. Rep. Internal Technical Document Services, AT&T Bell Laboratories, Murray Hill, N.J.
[Atk87] M.P. Atkinson, O.P. Bumeman *"Types and Persistence in Database Programming Language"*, ACM Computing Survey, Vol. 19, No. 2, June 1987.
[CERI86] S.Ceri, G.Gottlob, L.Lavazza: *"Translation and Optimization of Logic Queries: The Algebraic Approach"*, in Proc. 12th VLDB, Kyoto, 1986.
[CERI88] S.Ceri, S.Crespi Reghizzi, G.Gottlob, G.Lamperti, L.Lavazza, L.Tanca, R.Zicari *"The ALGRES Project"* in Proc. EDBT 88, Venice, 1988.
[CERI88a] S.Ceri, S.Crespi Reghizzi, G.Lamperti, L.Lavazza, R.Zicari *"ALGRES: An Advanced Database System for Complex Applications"* IEEE Software (to appear).
[ChHa82] A.K. Chandra, D. Harel *"Horn clauses and the fixpoint query hierarchy"* in Proc. 1st Symp. Principles of Database Systems, 1982, pp 158-163.
[DADA86] P.Dadam, K.Kuespert, F.Anderson, H.Blanken et al., *"A DBMS Prototype to Support Extended NF2 Relations: An Integrated View on Flat Table*

[1] UNIX is a trademark of AT&T.
[2] ULTRIX is a trademark of DEC Corporation.

and Hierarchies", IBM Heidelberg Scientific Center, ACM-SIGMOD Conference, 1986.c

[FIS83] P.Fisher, S.Thomas, *"Operators for Non-First-Normal-Form Relations"*, Proc. 7th COMPSAC, Chicago, November 1983.

[GALL78] H.Gallaire, J.Minker (eds.) *"Logic and Databases"* PlenumPress, 1978.

[GALL84] H.Gallaire, J.Minker, J.-M. Nicolas *"Logic and Databases: A Deductive Approach"*, ACM Computing Surveys, 16:2, June 1984.

[Gord79] M. Gordon, A. Milner, C. Wadsworth , Lecture Notes in Computer Science, vol. 78, Springer-Verlag, New York, 1979.

[GoZi88] G.Gottlob, R.Zicari, *"Closed World Databases Opened through Null Values"* in Proc. 14th VLDB, San Francisco, 1988.

[GUET87] R.Gueting, R.Zicari, D.M.Choy, *"An Algebra for Structured Office Documents"* IBM Almaden, Report RJ 5559 (56648), March 1987.

[JSSC82] B. Jaeschke, H.-J. Schek, *"Remarks on the Algebra on Non-First-Normal-Form Relations"*, Proc. 1st PODS March, 1982.

[KCB88] W. Kim, H. Chou, J. Banerjee *"Operations and Implementation of Complex Objects"*, in IEEE Transactions on Software Enfineering, Vol. 14, N. 7, July 1988.

[Klug82] A. Klug *"Equivalence of relational algebra and relational calculus query languages having aggregate functions"* Journal ACM, Vol. 29, pp. 699-717, July 1982.

[LINN87] V.Linnemann, *"Non First Normal Form Relations and Recursive Queries: an SQL based Approach"*, Proc. Workshop on Theory and Application of Nested relations and Complex Objects, Darmstadt, 1987.

[MAKI77] A. Makinouchi, *"A consideration on normal form of not-necessarily normalized relations in the relational model"*, Proc. 3rd VLDB, October 1977.

[Mylo80] J. Mylopoulos, H.K.T. Wong, *"Some Features of the Taxis Data Model"* 6th International Converence on Very Large Data Bases, Montreal, 1980.

[PIST86] P.Pistor, F.Andersen, *"Designing a Generalized NF² Model with an SQL-Type Language Interface"*, Proc. 12th VLDB, Kyoto, Japan, 8/86, pp. 278-285.

[REIT78] R.Reiter *"On Closed World Databases"*, in Logic and Databases, H.Gallaire and J. Minker (eds.), Plenum Press, 1978.

[ROTH84] *M.A.Roth, H.F.Korth, A.Silberschatz, "Theory of Non First Normal Forma Relational Databases"*,University of Texas at Austin, Techn. Rep. TR-84-36.

[ROTH85] M.A.Roth, H.F.Korth, A.Silberschatz, *"Extended Algebra and Calculus for non-1NF Relational Database"*, Techn. Rep. TR-84-36, Revised Version, University of Texas at Austin, 1985.

[SCHE82] H.-J. Schek, P.Pistor, *"Data structures for an Integrated Data Base Management and Information Retrieval System"*, Proc. 8th VLDB, Mexico, 9/82.

[SCSC86] H.-J. Schek, M.Scholl, *"An Algebra for the Relational Model with Relation-Valued Attributes"* Information Systems, 11:2, 1986.

[Schm77] J.W. Schmidt, *"Some high level language constructs for data of type relation"* ACM Trans. Database Systems 2,3 September 1977, 247-261.

[Smith83] J.M.Smith, S.Fox, T.Landers, *"ADAPLEX: Rationale and Reference Manual"* Second ed. Computer Corporation of America, Cambridge, Mass., 1983.

[YeSt88] D. Yellin, R.Strom *"INC: A Language for Incremental Computations"* in Proc. SIGPLAN 88 Conference on Programming Language Design and Implementation, Atlanta, Georgia, 1988.