An Algebraic Approach to the Manipulation of Complex Objects

Umberto Nanni[†] Silvio Salza[‡] Mario Terranova[‡]

[†]Dipartimento di Matematica Pura e Applicata, University of L'Aquila, L'Aquila, Italy
[‡]Istituto di Analisi dei Sistemi ed Informatica del CNR, Rome, Italy

Abstract

In this paper we present an algebra for complex objects, which has been developed within LOGIDATA+, a national project funded by the Italian National Research Council, as an internal language in a prototype system for the management of extended relational databases, with complex object types. The object algebra is a set-oriented manipulation language that plays in the object oriented DBMS the same role as the relational algebra in a relational system. That is providing efficient access to mass storage structures and simplifying query optimization. The algebra refers to a data model that includes structured data types and object identity, thus allowing both classes of objects and value-based relations. The main original feature of the algebra is the ability to cope with object identity, and, at the same time, to preserve the capability of handling value based complex structures without identity. This has required extension of the semantics of operators with respect to the nested relation model, and to introduce additional operators for type conversion and oid invention. The paper also briefly discusses the implementation of the primitives of the object algebra in the prototype by means of the operators of the relational algebra.

1. Introduction

In the last decade the relational model as proposed by Codd [8] has been widely adopted for standard applications, because of its simple and uniform structure and of the larger deal of independence between the logical and physical level.

However the relational approach has proved to be insufficient for several non-conventional applications, as CAD, CASE, office automation, multimedia databases and knowledge bases. In all these cases the flat structure of the model makes the representation of complex and structured data difficult.

Several proposals have been made to extend the relational model, both overcoming the restrictions imposed by First Normal Form [10, 9, 16, 15], and introducing the concept of *object identity* [12, 13, 11, 4, 17]. For such extended models several authors have presented query and manipulation languages based on calculus [6], on algebra [2, 9, 1] and logic [3].

From the system architecture point of view, the algebraic approach consists, as we will see later in more detail, in defining a set-oriented manipulation language that will play in the object oriented DBMS a role equivalent to that of the relational algebra in a relational DBMS. This allows efficient access and manipulation of mass storage structures and query tree optimization.

This approach makes especially sense when the object oriented system is aimed at traditional database applications, where the typical transaction involves large sets of objects. For this class of applications indeed the relational systems perform already quite well, and the motivation for using the object oriented model lies mostly in improving the design phase, and, in general, in having a more direct representation of the real world in the schema. Therefore in such a context the relational model can still be used as an internal level of representation, by mapping the object schema into a relational schema.

According to these ideas a prototype system has been developed within LOGIDATA+, a national project funded by the Italian National Research Council, that has the goal of integrating logic programming and extended relational databases, with complex data types. The architecture of the prototype is sketched in Figure 1; further details on the implementation are given in a later section.

The system is based on a commercial relational DBMS that provides the basic support for the permanent storage of data and the relational operations, as well as for concurrency control and recovery, and adopts LOA (LOGIDATA+ Object Algebra), an object oriented algebraic language for the internal repre-

0073-1129-1/92 $3.00 © 1992 IEEE

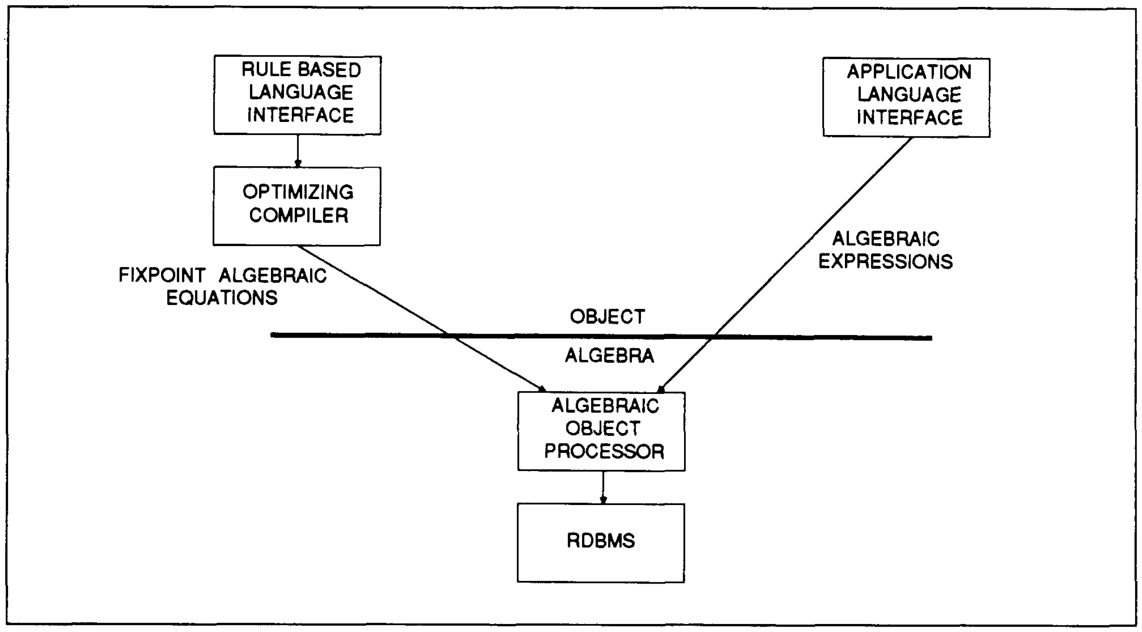
RULE BASED LANGUAGE INTERFACE
APPLICATION LANGUAGE INTERFACE
OPTIMIZING COMPILER
FIXPOINT ALGEBRAIC EQUATIONS
ALGEBRAIC EXPRESSIONS
OBJECT
ALGEBRA
ALGEBRAIC OBJECT PROCESSOR
RDBMS

Figure 1: The LOGIDATA+ prototype architecture

sentation of the transactions.

At the user level, queries are expressed in a rule-based declarative language, and then compiled and optimized, according to standard rewriting techniques, to produce a set of fixpoint equations. These are evaluated by the Algebraic Object Processor, that maintains the mapping between the object schema and the relational schema, and translates the object algebra expressions into relational algebra. An additional interface is also provided by allowing the embedding of algebraic queries in an application language. These are directly handled by the Algebraic Object Processor.

In this paper we focus on the definition of the algebraic object language LOA and we discuss some issues concerning the implementation of the Algebraic Object Processor.

2. The Object Algebra

LOA provides set-oriented operators to manipulate collections of complex objects and structured values, which are defined according to the LOGIDATA+ model [5], whose main features are the following:

- it allows the definition of types, functions, and two kinds of data collections: the *class* (based on object identity), and the *relation* (value based);
- data are basically structured with the tuple, set, and sequence constructors;
- an *isa* hierarchy with multiple inheritance can be defined in the set of classes.

Further details on the data model are given in Section 3 with the formal definition of types.

As far as the algebraic operators are concerned, we had to extend previous proposals. These basically dealt with the Nested Relation model [9, 16, 2] having only the relation and tuple constructors, and thus producing a recursive schema with a tree structure. Abiteboul and Beeri [1] have considered a set constructor and have added to the algebra a powerset operator to reach the expressive power of the domain independent calculus.

In our case, besides considering different kinds of constructors, the main original contributions are related to the need to cope with object identity, and, at the same time, to preserve the capability of handling value based complex structures without identity. These features are not considered in the previous proposals which referred to strictly value based models.

Object identifiers are treated as special atomic values that cannot be directly accessed, and have to be preserved in the relational operations. This had to be taken into account in the definition of the semantics of the operators, and demanded for additional operators to convert objects into structured values and vice versa.

The extension of the operators of the relational algebra is attained by introducing in the conditions a set of *navigational* operators to move through complex data structures, e.g. to take components from structures, to extract elements from sequences, and to transform data types.

Nest and *Unnest* operators are defined as in [10, 15]. These allow respectively the grouping of several tuples over an attribute or group of attributes, and the distribution of the elements of a set attribute over a set of tuples. Similar operators are also provided for the tuple constructor. We may point out that, due to the introduction of the navigational operators in the conditions, the role of the Nest and Unnest is mainly restricted to the restructuring of data.

Further primitives are defined to transform objects or their components into tuples and vice versa. The latter operation generates new objects and requires object invention.

The paper is organized as follows. In the next Section we give the formal definition of the data model. In Section 4 we introduce the language used to express complex conditions inside the operators of the extended relational algebra. These operators are discussed in Section 5. In Section 6 we introduce the structure and conversion operators. In Section 7 we discuss the mapping between object algebra and relational algebra and the architecture of the Algebraic Object Processor.

3. The Data Model

Given a finite set of *domains* $\mathbf{D}_1, \ldots, \mathbf{D}_D$ with *domain names* $\mathcal{D}_1, \ldots, \mathcal{D}_D$, a countable domain of *object identifiers* Ω, and a countable set of *attribute names* $\mathcal{A}_1, \mathcal{A}_2, \ldots$, we refer to a database schema composed by the following elements:

- A finite set of *types names*, or shortly *types*, $\theta_1, \ldots, \theta_\Theta$, and the corresponding *type definitions*.

- A finite set of *classes* $\mathbf{C}_1, \ldots, \mathbf{C}_C$ with names $\mathcal{C}_1, \ldots, \mathcal{C}_C$.

- A finite set of *relations* $\mathbf{R}_1, \ldots, \mathbf{R}_R$ with names $\mathcal{R}_1, \ldots, \mathcal{R}_R$.

- An ISA hierarchy between the classes.

The domain Ω contains the object identifiers that are associated to the objects, each one having an identifier which is unique in all the database. Classes are collections of objects of the same type, and relations are collections of structured values.

Type definitions allow to build structured types from the basic types associated to the domains, and, for *object types*, to add the identity. A *value set*, i.e. the set of all possible values, is associated to each type. More formally:

- A *type* θ is either a *value type* τ or an *object type* ω.

- A domain name $\mathcal{D}_i$ is a *value type* and the corresponding value set is $\mathbf{D}_i$.

- If $\theta_1, \ldots, \theta_n$ are types with value sets $\mathbf{V}_1, \ldots, \mathbf{V}_n$, then $\tau = (\mathcal{A}_1 : \theta_1, \ldots, \mathcal{A}_n : \theta_n)$ defines a *tuple type* τ, with value set $\mathbf{V}_\tau = \mathbf{V}_1 \times \ldots \times \mathbf{V}_n$. Round brackets denote the *tuple constructor.*

- If θ is a type and $\mathbf{V}$ is the corresponding value set then $\tau = \{\theta\}$ defines a *set type* τ with value set $\mathbf{V}_\tau = \text{PART}(\mathbf{V})$, i.e. the powerset of V. Curly brackets denote the *set constructor.*

- If θ is a type and $\mathbf{V}$ is the corresponding value set then $\tau = \langle\theta\rangle$ defines a *sequence type* τ with value set $\mathbf{V}_\tau = \text{SEQ}(\mathbf{V})$, i.e. the set of all the sequences over V. Angle brackets denote the *sequence constructor.*

- If τ is a *tuple* value type with value set $\mathbf{V}$, and $\mathcal{C}$ is a class name, then $\omega = [\mathcal{C}, \tau]$ is an *object type*, and the corresponding value set is $\mathbf{V}_\omega = \Omega \times \mathbf{V}$.

Each relation is defined on a value type and each class is defined on an object type. More precisely, as we shall see below, there is a one to one correspondence between object types and classes.

We may now introduce the notion of *refinement* as a partial order relationship in the set of types, according to the following definition.

A type θ (either an object type or a value type) is a *refinement* of a type θ' (in symbols $\theta \preceq \theta'$) if and only if one of the following condition holds:

- $\theta = \theta'$;

- $\theta = \omega = [\mathcal{C}, \tau]$, $\theta' = \omega' = [\mathcal{C}', \tau']$, with $\tau \preceq \tau'$;

- $\tau = (\mathcal{A}_1 : \theta_1, \ldots, \mathcal{A}_{k+p} : \theta_{k+p})$, $\tau' = (\mathcal{A}_1 : \theta'_1, \ldots, \mathcal{A}_k : \theta'_k)$, with $\theta_i \preceq \theta'_i$, for $1 \leq i \leq k$;

- $\tau = \{\theta\}$, $\tau' = \{\theta'\}$, with $\theta \preceq \theta'$;

- $\tau = \langle\theta\rangle$, $\tau' = \langle\theta'\rangle$, with $\theta \preceq \theta'$;

Note that the ISA relationship in the schema requires that if $\mathcal{C}$ ISA $\mathcal{C}'$ then the type ω of $\mathcal{C}$ must be a refinement of the type ω' of $\mathcal{C}'$. Object types that have a common supertype in the ISA hierarchy are in a special relationship and are said to be *compatible* types.

In Figures 2 and 3 we present a sample schema, that will be referred to in the other examples, and the related type definitions.

4. Conditions on structured types

To extend the operators of relational algebra we need to extend the definition of the conditions to cope with structured data types. First we introduce the notion of *derived component*, which is an identifier that we use in the conditions to denote an element in a class or in a relation, or a structured value that is a part of it. Formally, given a relation $\mathcal{R}$ or a class $\mathcal{C}$ of type θ:

- If $\mathcal{A}$ is an attribute of type θ then $\mathcal{A}$ denotes a derived component from $\mathcal{A}$ of type θ.
- If $\mathcal{A}$ is an attribute of type θ, with corresponding value type $\tau = (\mathcal{A}_1 : \theta_1, \ldots, \mathcal{A}_k : \theta_k)$, and E_i is a derived component from $\mathcal{A}_i$ of type θ_E, then $\mathcal{A}.E_i$ is a derived component from attribute $\mathcal{A}$ of type θ_E.
- If $\mathcal{A}$ is an attribute of type $\{\theta_B\}$, with corresponding value type $\tau_B = (\mathcal{A}_1 : \theta_1, \ldots, \mathcal{A}_k : \theta_k)$, and E_i is a derived component from $\mathcal{A}_i$ of type θ_E, then $\mathcal{A}.E_i$ is a derived component from attribute $\mathcal{A}$ of type $\{\theta_E\}$.
- If $\mathcal{A}$ is an attribute of type $\langle\theta_B\rangle$, with corresponding value type $\tau_B = (\mathcal{A}_1 : \theta_1, \ldots, \mathcal{A}_k : \theta_k)$, and E_i is a derived component from $\mathcal{A}_i$ of type θ_E, then $\mathcal{A}.E_i$ is a derived component from attribute $\mathcal{A}$ of type $\langle\theta_E\rangle$.
- If E denotes a derived component of type θ_E, with an associated value type τ_E, then:
  - if $\tau_E = \{\{\theta_B\}\}$ then FLAT(E) denotes a derived component of type $\{\theta_B\}$;
  - if $\tau_E = \langle\langle\theta_B\rangle\rangle$ then FLAT(E) denotes a derived component of type $\langle\theta_B\rangle$;
  - if $\tau_E = \langle\theta_B\rangle$ and then POS(E, n) denotes a derived component of type θ_B (the n-th element in the sequence), SET(E) denotes a component of type $\{\theta_B\}$;
- If $\mathcal{R}$ is a relation of type $\tau = (\mathcal{A}_1 : \theta_1, \ldots, \mathcal{A}_k : \theta_k)$, and and E_i is a derived component from $\mathcal{A}_i$ of type θ_E, then $\mathcal{R}.E_i$ is a derived component from the relation $\mathcal{R}$ of type θ_E, and $\mathcal{R}$ denotes a derived component of type τ, i.e. a tuple in the relation.
- If $\mathcal{C}$ is a class of type θ with associated value type $\tau = (\mathcal{A}_1 : \theta_1, \ldots, \mathcal{A}_k : \theta_k)$, and E_i is a derived component from $\mathcal{A}_i$ of type θ_E, then $\mathcal{C}.E_i$ is a derived component from the class $\mathcal{C}$ of type θ_E, and $\mathcal{C}$ denotes a derived component of type θ, i.e. an object in the class.

With reference to the schema in the Figures 2 and 3, the following are examples of components:

- **Student**: denotes an object of type $\omega_{\texttt{stud}}$ in the class **Student**.
- **Student.curriculum.exams**: denotes the set of couples (**course, degree**) of an object of the class **Student**.
- **Family.children.name.givennames**: denotes the sequence of sequences of givennames of the children of a given family. To get the set of names one should write: **Family**.FLAT(SET(**children.name**. SET(**givennames**)))

Components are used to build *conditions*, i.e. boolean predicates based on the comparisons between components and constants of a compatible type. Formally:

- If E_1 and E_2 are components of value type τ_1 and τ_2, with value sets $\mathbf{V}_1 \equiv \mathbf{V}_2 \equiv \mathbf{V}$, then $E_1 \text{OP} E_2$ and $E_1 \text{OP} v$ are conditions, where OP is a comparison operator, and $v \in \mathbf{V}$.
- If E_1 and E_2 are components of object type ω_1 and ω_2, then $E_1 = E_2$ or $E_1 \neq E_2$ are conditions.
- If E_1 is a component of type θ and E_2 is a component of type $\{\theta\}$ then $E_1 \in E_2$ and $E_1 \notin E_2$ are conditions.
- Every boolean expression whose terms are conditions is a condition.

Note that only equality and inequality comparisons are allowed between components of object type, and that comparison may take place between objects of different types. In all these cases the comparison is based on the object identity.

5. Extending the Operators of Relational Algebra

The traditional operators of the relational algebra, SELECT, PROJECT and JOIN, are extended in two ways.

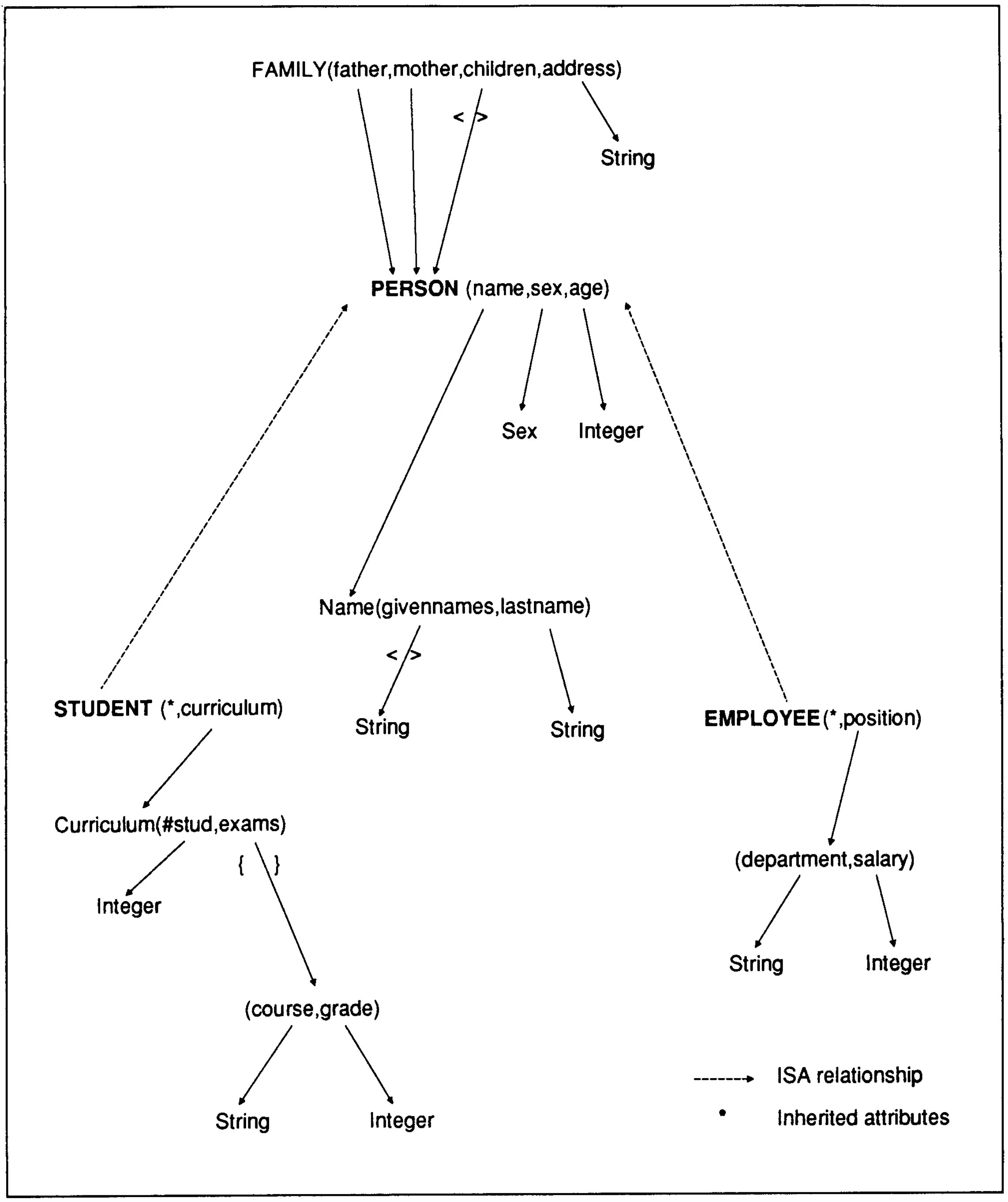

Figure 2: A sample schema

Domain names:

`String`; `Integer`; `Sex`.

Value types:

$\tau_{\mathtt{name}} = (\mathtt{givennames} : \langle \mathtt{String} \rangle,\ \mathtt{lastname} : \mathtt{String});$

$\tau_{\mathtt{pers}} = (\mathtt{name} : \tau_{\mathtt{name}},\ \mathtt{sex} : \mathtt{Sex},\ \mathtt{age} : \mathtt{Integer});$

$\tau_{\mathtt{stud}} = (\mathtt{name} : \tau_{\mathtt{name}},\ \mathtt{sex} : \mathtt{Sex},\ \mathtt{age} : \mathtt{Integer},\ \mathtt{curriculum} : \tau_{\mathtt{curr}});$

$\tau_{\mathtt{curr}} = (\#\mathtt{stud} : \mathtt{Integer},\ \mathtt{exams} : \{(\mathtt{course} : \mathtt{String},\ \mathtt{grade} : \mathtt{Integer})\});$

$\tau_{\mathtt{empl}} = (\mathtt{name} : \tau_{\mathtt{name}},\ \mathtt{sex} : \mathtt{Sex},\ \mathtt{age} : \mathtt{Integer},\ \mathtt{position} : \tau_{\mathtt{pos}});$

$\tau_{\mathtt{pos}} = (\mathtt{department} : \mathtt{String},\ \mathtt{salary} : \mathtt{Integer}).$

$\tau_{\mathtt{fam}} = (\mathtt{father} : \omega_{\mathtt{pers}},\ \mathtt{mother} : \omega_{\mathtt{pers}},\ \mathtt{children} : \langle \omega_{\mathtt{pers}} \rangle,\ \mathtt{addr} : \mathtt{String}).$

Object types:

$\omega_{\mathtt{pers}} = [\mathtt{Person}, \tau_{\mathtt{pers}}];$

$\omega_{\mathtt{stud}} = [\mathtt{Student}, \tau_{\mathtt{stud}}];$

$\omega_{\mathtt{empl}} = [\mathtt{Employee}, \tau_{\mathtt{empl}}].$

Relations:

$\mathtt{Family} : \tau_{\mathtt{fam}}.$

Classes:

`Person`; `Student`; `Employee`.

ISA hierarchy:

`Student` ISA `Person`;

`Employee` ISA `Person`.

Figure 3: Type definitions for the sample schema

First we allow more powerful conditions, using the framework introduced in the previous section. The other extension concerns the definition of the type of the result according to the type of the operands and the structure of the conditions.

The SELECT extracts from a relation or a class the subset of elements satisfying a given condition. Formally:

- If $\mathcal{R}$ is a relation of type τ and f a condition on the type τ, then $\mathcal{R}' \Leftarrow \text{SELECT}(\mathcal{R}; f)$ is a relation of type τ.
- If $\mathcal{C}$ is a class of type $\omega = [\mathcal{C}, \tau]$ and f a condition on the objects of $\mathcal{C}$, then $\mathcal{C}' \Leftarrow \text{SELECT}(\mathcal{C}; f)$ is a class of type $\omega' = [\mathcal{C}', \tau]$.

The PROJECT extends the relational operation in the sense that, instead of a subset of attributes one has to specify the supertype of the operand type that contains the required information. Note that all the nested levels of the data structure are kept in the result. Formally:

- If $\mathcal{R}$ is a relation of type τ and $\tau \preceq \tau'$ then $\mathcal{R}' \Leftarrow \text{PROJECT}(\mathcal{R}; \tau')$ is a relation of type τ'.
- If $\mathcal{C}$ is a class of type $\omega = [\mathcal{C}, \tau]$ and $\tau \preceq \tau'$, then $\mathcal{C}' \Leftarrow \text{PROJECT}(\mathcal{C}; \tau')$ is a class of type $\omega' = [\mathcal{C}', \tau']$.

The projection includes the duplicate elimination. This can be effective only when the operand is a relation.

The JOIN is defined in such a way that, regardless of the type of the operands, that can be classes or relations in any combination, the result is a binary relation, i.e. further level is added to the structure of the type.

Formally, if $\mathcal{X}_1$ and $\mathcal{X}_2$ are classes or relations of types θ_1 and θ_2, and f is a condition involving components from types θ_1 and θ_2, then $\mathcal{R}' \Leftarrow \text{JOIN}(\mathcal{X}_1, \mathcal{X}_2; f)$ is a relation of type $\tau' = (\mathcal{A}_1 : \theta_1, \mathcal{A}_2 : \theta_2)$.

With reference to the schema in Fig. 1 the following are sample queries:

–

Homonyms $\Leftarrow$ SELECT(**Family**; **Family**.**father**.**name**.SET(**givennames**) $\cap$ **Family**.FLAT(SET(**children**.**name**.SET(**givennames**))) $\neq \emptyset$.

Homonyms contains all the families in which at least one child shares a name with the father.

– **F_names** $\Leftarrow$ PROJECT(**Family**; (**father** : (**name** : (**givennames**)))).

Fnames contains the givennames of the fathers in **Family**. Note that, as the type of the result is a supertype of the type of the operand, it maintains the complete nested structure around the projected attributes. To flatten the structure, one should use the structure operators introduced in the next section.

– **Artists** $\Leftarrow$ JOIN(**Family**, **Student**; (**Family**.SET(**children**) $\ni$ **Student**) $\wedge$ (**Student**.**curriculum**.**exams**.**course** $\ni$ '**finearts**')).

Artists is a binary relation of type: (**Family** : $\tau_{\mathtt{fam}}$, **Student** : $\omega_{\mathtt{stud}}$) and contains all the families with a child who passed the exam of fine arts.

Additional operators are provided for set operations. These are straightforward extensions of the corresponding relational operators UNION, INTERSECT, DIFFERENCE.

6. Structure and Conversion Operators

The need for these operators arises from the richer structure of the types in the object oriented data model. They are required in the algebra to attain the full capability of data restructuring. This includes the transformation of the components in a structure, from values to objects, and vice versa. Some of these operators have been introduced in algebras for the nested relation model [15, 7].

The structure operators apply only to relations, and produce a relation as a result. They allow to modify the structure by means of actions such as gathering several attributes in a tuple and multiple values of an attribute in a set.

More specifically NEST is used to collect in a set the group of distinct values of an attribute of a tuple that share the same value of the rest of the tuple. Formally, if $\mathcal{R}$ is a relation of type $\tau = (\mathcal{A}_1 : \theta_1, \ldots, \mathcal{A}_k : \theta_k)$ then $\mathcal{R}' \Leftarrow \text{NEST}(\mathcal{R}; \mathcal{A}_k)$ is a relation of type $\tau' = (\mathcal{A}_1 : \theta_1, \ldots, \mathcal{A}_k : \{\theta_k\})$.

UNNEST is the reverse operation, and generates a separate tuple in the result relation for every distinct value in an attribute of type set. Formally, if $\mathcal{R}$ is a relation of type $\tau = (\mathcal{A}_1 : \theta_1, \ldots, \mathcal{A}_k : \{\theta_k\})$ then $\mathcal{R}' \Leftarrow \text{UNNEST}(\mathcal{R}; \mathcal{A}_k)$ is a relation of type $\tau' = (\mathcal{A}_1 : \theta_1, \ldots, \mathcal{A}_k : \theta_k)$. With the same definition the UNNEST can be applied also to sequences.

For example the following steps compute a binary relation that associates to each father the set of his wives:

Couples $\Leftarrow$ PROJECT(**Family**; (**father**, **mother**))
Polygamy $\Leftarrow$ NEST(**Couples**; **mother**)

On the other end the following steps compute from **Family** the relation **Parent** that contains the couples (parent, child):

U_Family ⇐ UNNEST(**Family**; **children**)
Parent ⇐ UNION(PROJECT(**U_Family**; (**father** : **Person**, **children** : **Person**)), PROJECT(**U_Family**; (**mother** : **Person**, **children** : **Person**)))

In addition to NEST and UNNEST two more operators are provided, CLUSTER that groups a subset of attributes in a tuple to generate a subtuple in a single attribute, and MELT which does the reverse operation, i.e. flattens a tuple structure with a nested tuple attribute.

Conversion operators basically allow to convert values into objects and vice versa. Depending if the whole structure or part of it is to be converted, different operators are required:

- CLASS: transforms the elements of a relation into objects, thus defining a new class and the corresponding object type.
- DECLASS: transforms the objects in a class in structured values, by depriving them of the object identity, and thus eliminating duplicate values from their result.
- OBJECT: converts a value component inside a tuple structure into a component of object type, thus defining a new class.
- VALUE: converts an object component inside a tuple structure into the corresponding value.

Note that the execution of the conversion operators may require the invention of object identifiers. For further details and examples one should refer to [14].

7. The Algebraic Object Processor

The architecture of the Algebraic Object Processor is represented in Figure 4.

As we pointed out in the introduction, the object schema is mapped into a relational schema. This function is performed by the Schema Manager and requires two different kinds of actions. The first is to provide a mechanism to maintain the object identity, which is obtained by introducing a new type of key attribute directly managed by the system. Moreover the schema is normalized to eliminate the set and sequence constructors.

The Schema Manager needs also to make the decisions concerning the physical organization of data, which in relational systems are usually taken during the design phase. This mainly consists in setting up the access paths, and in particular in building indices to speed up the join operations which originate from the normalization process. For this purpose the system may also utilize the option of clustering indices which are available in the implementation environment (Oracle).

According to the schema mapping the object algebra equations are translated into relational queries. This requires also an optimization phase, which has to take into account both the information on the physical organization of the database and the behaviour of the optimizer of the relational DBMS.

The handling of recursive queries requires additional actions. As stated in the introduction, the compiler generates a system of fixpoint equations of the Object Algebra. These are in turn translated into fixpoint equations of relational algebra and then evaluated by the Fixpoint Evaluator.

This computation demands for a repeated interaction with the RDBMS, and is indeed not very efficient if the relational operations are performed entirely in mass memory. Thus, as shown in Figure 4, an additional module has been added to maintain in main memory during the fixpoint evaluation the temporary relations corresponding to the mutually recursive predicates.

In the Main Memory Database the relations are stored in B*-tree structures. Moreover to improve the execution of the semi-naive evaluation a special primitive has been introduced which integrates the three actions of inserting a new tuple, eliminating duplicates and checking if the relation has changed in the last iteration.

The prototype is currently being implemented under UNIX System V on a Sun Sparcstation and uses Oracle as the underlying RDBMS. Preliminary results pointed out the dramatic improvement given by the main memory database in the evaluation of recursive queries, and the crucial role of the optimization of the physical organization of data.

8. Conclusions

In this paper we have presented an algebra for the manipulation of complex objects. The algebra refers to a data model which includes structured data types and two different kinds of data collections: classes with object identity, and value based relations. To extend the relational primitives more powerful conditions are provided, based on a set of navigational operators that allow to move through complex data structures. Additional new operators are defined to manipulate the structure of data, and to attain the full capability of

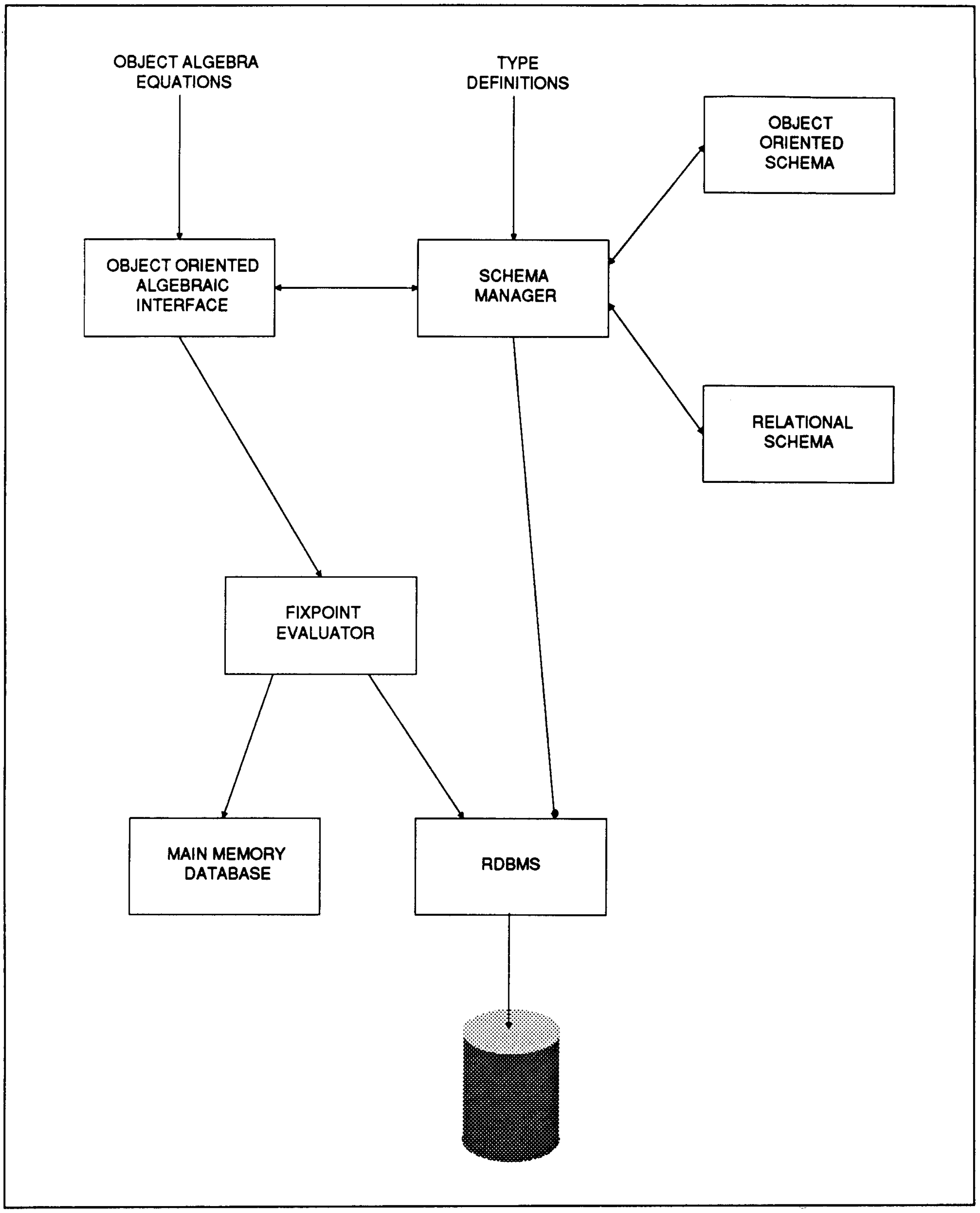

Figure 4: Architecture of the Algebraic Object Processor

data restructuring.

The algebra has been developed within the project LOGIDATA+ as an internal language in the prototype object oriented system. From the system performance point of view, the algebraic set-oriented approach proves to be effective, especially when moving to the object oriented model traditional database applications, where transactions typically operate on large sets of objects.

In the prototype system the object oriented schema is mapped into a relational schema, and the database is actually managed by a relational DBMS, that provides the basic support for the permanent storage of data as well as for concurrency control and recovery. The object algebra expressions can then be mapped into relational algebra expressions, thus relying on the efficiency of the RDBMS for the access to mass storage structures and the efficient execution of set-oriented operations.

Although the algebra has been originally conceived as an internal language for the prototype, it proves also to be quite effective in expressing complex queries. It may then form the basis for the definition of a high level user oriented query and manipulation language.

References

[1] S. Abiteboul and C. Beeri. *On the Power of Languages for the Manipulation of Complex Objects.* Technical Report 846, INRIA, 1988.

[2] S. Abiteboul and N. Bidoit. Nonfirst normal form relations: an algebra allowing data restructuring. *Journal of Comp. and System Sc.*, 33(1):361–393, 1986.

[3] S. Abiteboul and S. Grumbach. Col: a logic-based language for complex objects. In *EDBT'88 (Int. Conf. on Extending Database Technology), Venezia, Lecture Notes in Computer Science 303*, pages 271–293, Springer-Verlag, 1988.

[4] S. Abiteboul and P. Kanellakis. Object identity as a query language primitive. In *ACM SIGMOD International Conf. on Management of Data*, pages 159–173, 1989.

[5] P. Atzeni and L. Tanca. *The LOGIDATA+ Rule Language.* Rapporto LOGIDATA+, Politecnico di Milano e IASI-CNR, Roma, 1990. Presented at the Workshop "Information Systems '90", Kiev.

[6] F. Bancilhon and S. Khoshafian. A calculus for complex objects. *Journal of Comp. and System Sc.*, 38(3):326–340, 1989.

[7] S. Ceri, S. Crespi-Reghizzi, G. Lamperti, L.A. Lavazza, and R. Zicari. ALGRES: an advanced database system for complex applications. *IEEE Software*, 7(4), July 1990.

[8] E.F. Codd. A relational model for large shared data banks. *Communications of the ACM*, 13(6):377–387, 1970.

[9] P.C. Fischer and S.J. Thomas. Operators for non-first-normal-form relations. In *Proc. of IEEE Computer Software Applications*, pages 464–475, 1983.

[10] G. Jaeschke and H.-J. Schek. Remarks on the algebra for non first normal form relations. In *ACM SIGACT SIGMOD Symp. on Principles of Database Systems*, pages 124–138, 1982.

[11] S. Khoshafian and G. Copeland. Object identity. In *ACM Symp. on Object Oriented Programming Systems, Languages and Applications*, 1986.

[12] G.M. Kuper. *The Logical Data Model: A New Approach to Database Logic.* PhD thesis, Stanford University, 1985.

[13] G.M. Kuper and M.Y. Vardi. A new approach to database logic. In *Third ACM SIGACT SIGMOD Symp. on Principles of Database Systems*, 1984.

[14] U. Nanni, S. Salza, and M. Terranova. *LOA: the LOGIDATA+ Object Algebra.* Technical Report 5/23, Progetto Finalizzato Sistemi Informatici e Calcolo Parallelo, 1990.

[15] M.A. Roth, H.F. Korth, and A. Silberschatz. Extended algebra and calculus for $\neg$1NF relational databases. *ACM Trans. on Database Syst.*, 13(4):389–417, December 1988.

[16] H.-J. Schek and M.H. Scholl. The relational model with relation-valued attributes. *Information Systems*, 1986.

[17] M.H. Scholl and H.-J. Schek. A relational object model. In *Third International Conference on Data Base Theory, Paris, Lecture Notes in Computer Science 470*, pages 89–105, 1990.