An implementation model for reasoning with complex objects

Q. Chen, Georges Gardarin

INRIA

UNITÉ DE RECHERCHE
INRIA-ROCQUENCOURT

Institut National
de Recherche
en Informatique
et en Automatique

Domaine de Voluceau
Rocquencourt
B.P.105
78153 Le Chesnay Cedex
France
Tél.:(1) 39 63 55 11

Rapports de Recherche

N° 793

AN IMPLEMENTATION MODEL FOR REASONING WITH COMPLEX OBJECTS

Qiming CHEN
Georges GARDARIN

FEVRIER 1988

UN MODELE D'IMPLANTATION DE REGLES SUR OBJETS COMPLEXES

Qiming CHEN
SM Research Institute
16 Bei Tai Ping Lu
BEIJING 100039
China

Georges GARDARIN
INRIA & Univ. Paris VI
SABRE Project
BP.105, 78153 LE CHESNAY-Cédex
France

Résumé :

Dans cet article, nous proposons tout d'abord une extension naturelle de DATALOG pour traiter des objets complexes comme des prédicats imbriqués. Puis, nous introduisons une extension du modèle relationnel avec des références pour représenter les objets complexes. Un modèle d'implantation d'un langage de règles tel DATALOG avec objets complexes sur un SGBD relationnel gérant des références est alors proposé.

Finalement, considérant l'implantation d'un point de vue sémantique, nous discutons les problèmes de sûreté d'une question sur des règles à prédicats imbriqués.

Mots-clés : Bases de données relationnelles, programmation logique, objets complexes, prédicats imbriqués.

AN IMPLEMENTATION MODEL FOR REASONING WITH COMPLEX OBJECTS

Qiming CHEN
SM Research Institute
16 Bei Tai Ping Lu
Beijing 100039
CHINA

Georges GARDARIN
INRIA & UNIV. PARIS VI
SABRE Project
BP 105, 78153 LE CHESNAY
FRANCE

Abstract :

In this paper, we first propose a natural syntactical extension of DATALOG for dealing with complex objects represented as nested predicates. Then, we introduce the token object model which is a simple extension of the relational model with tokens to represent complex objects and support referential information sharing. An implementation model of a NESTED_DATALOG program is defined by mapping it to the token object model which remains a straightforward extension of classical logical databases. Through this work, we can accommodate two basic requirements : The availability of a rule language for reasoning with complex objects, and the mechanism for mapping a complex object rule program to a relational DBMS. Finally, the definition of the semantics of NESTED_DATALOG through a mapping to the token object model allows us to discuss the safety issue in NESTED_DATALOG.

Key-words : Relational databases, logic programming, complex object, prédicats imbriqués.

1. INTRODUCTION

Database (DB) and Logic Programming (LP) technologies are moving fast towards a common destination. This holds true both from their common ancestor of mathematical logic and from the complementary benefits they can provide. From a database point of view, the major efforts made so far consist in the logical foundation given to relational DB [Gallaire84, Reiter84] and the extensions proposed to relational DB to support rules. These extensions yield DATALOG, a pure Horn clause rule language with a fixpoint semantics. DATALOG has been extended for handling functions, negation and set terms [Zaniolo 85] [Zaniolo 86] [Beeri 86] [Kuper 86] [Kuper 87] [Abiteboul87] [Gardarin87a].

The handling of complex facts in logic programming is naturally integrated by using functions. It is possible to query these facts using predicates containing functional arguments. This facility makes logic programming more powerful and flexible than pure DATALOG based deductive databases, although typing and large predicate extensions are not well supported. Extending deductive database technology to handle complex objects has become a key issue for introducing it to engineering applications. However, to directly support database reasoning involving complex objects, the following problems still remain :

(a) The support of aggregation hierarchies with mutually nested tuples and sets, namely, the description of the references among objects in a logic framework.

(b) The consistency and complexity of the rule language semantics against the possible introduction of nested predicates, multilevel nested sets and so on.

(c) The support of information sharing, particularly, the link between a complex object rule language and an existing object management system adopting sharing by references.

(d) The availability of inferences at any given level of an aggregation hierarchy.

These problems have been investigated previously. Speaking roughly, they were tackled along the following two lines :

(1) Handling complex objects as nested functional terms [Zaniolo 85] [Zaniolo 86] [Beeri 86] [Abiteboul87], which firmly stays within the FOL world but has obvious limitations in predicating arbitrary objects and in supporting data abstraction and referential sharing. Most of these attempts allow the database to store complex facts in non normalized relation and introduce an extended relational algebra to deal with non flat relations.

(2) Defining a new theory for complex objects with an associated calculus and rule language. According to the approach described in [Bancilhon86], complex objects can be built from atomic objects by applying to them recursively a tuple and a set construct. Together with the subobject relationship $\leq$, which is essentially a containment relationship, the set of objects forms a partial order lattice. This structure is used to define rule languages for reasoning about complex objects.

However, the implementation model of such languages has not been properly developed.

We concentrate in this paper on the extension of DATALOG for dealing with complex objects. We first propose a natural extension of DATALOG for dealing with nested predicates, called **NESTED_DATALOG**. With NESTED_DATALOG, any predicate, nested or not, can have a database extension. To define the semantics of NESTED_DATALOG, we introduce **the token object model**; it is a simple extension of the relational model with tokens to represent complex objects and support information sharing. The token object model may be perceived as the implementation model for NESTED_DATALOG. Indeed, it can be understood as a simple extension of the relational model with tokens. Tokens are introduced to represent complex objects and to provide links among composing sub-objects. The multilevel configuration hierarchies of complex objects are represented in a level-independent fashion through the use of token values, as unnested sets or tuples of tokens. The transition of inferences at different levels of the object configuration hierarchy is made by passing tokens through a kind of special predicates called **token-predicates**, which represent the link between the token and the token value of a non-atomic object, thus associating the abstract description of an object at a higher level and the more detailed description of it at a lower level. The definition of the semantics of NESTED_DATALOG through a mapping to the token object model allows us to discuss the safety issue in NESTED_DATALOG, which is not an obvious problem.

Compared with previous approaches, the proposed framework for reasoning with complex objects has the following distinct characteristics :

(1) It allows the user to query any predicate, even those nested in other predicates. Thus, it allows the user to zoom in any particular level, while suppressing the details of other levels.

(2) Since the tokens provides links among different parts of a complex object, thus expands the aggregation hierarchy to a level-independent 'flat' scheme, it may be implemented on top of classical relational systems. Such an implementation has already been considered by others; for example, building an object-oriented interface over a relational backend [Kiernan87] or compiling logic programming languages with nested terms into conventional relational algebra operations [Al-Amoudi87] require similar concepts and methodologies to that presented here.

(3) It simplifies the semantics of LP with sets by introducing strong typing, which allows to map the multilevel nested sets and tuples to only one level. Indeed, the token model does not refer any nested predicate or set.

The rest of this paper is organized as follows : NESTED_DATALOG is presented in section 2 as a natural extension of logic programming over complex objects. Then, the token object model is introduced in Section 3, as the implementation model of NESTED_DATALOG facts. Section 4 is to offer the development of the mapping of rules from NESTED_DATALOG to the token object model enhanced with DATALOG. Section 5 presents the further discussions on the semantics and safety of NESTED_DATALOG rules.

2. THE SYNTAX OF NESTED_DATALOG

To introduce our rule language, let us see how complex objects would normally be dealt with as nested functional terms. As usual, $\{O_1,O_2...O_n\}$ represents the set of elements $O_1,O_2,...O_n$ and (O_1,O_2,O_n) is a tuple constructed with elements $O_1,O_2,...O_n$. The example given below includes a complex fact `registration` with nested functional terms, which shows a natural syntax to represent complex facts with the tuple and set constructor. As mentioned in [Tsur 86], predicates are top-level constructs which cannot be parameters of other terms or predicates; thus only `registration` is entitled to be a predicate symbol in this example. Courses, course, prof, students, student are function symbols.

registration (cs, 1987, courses (
{course (231, db, prof (smith, male, 36),
students ({student (john, 17), student (jan, 18), student (hull, 20)})),
course (171, os, prof (smith, male, 36),
students ({student (lee, 18), student (jan, 18), student (hull, 20)})),
course (281, ai, prof (smith, male, 36),
students ({student (john, 17), student (lee, 18), student (hull, 20)})) })).

Suppose we have queries such as
(a) Find the set of student names Y professor x (name) teaches on course c (course number).
(b) Find student lee's age.
To express and answer these queries in an LP system supporting functions, one must start from the top predicate `registration`, then go all the way down to the points where the required information can be found. Even for a very simple query such as (b) it could not be processed directly since `student(lee,18)` has been treated as a **functional term**, thus cannot be taken as a **predicate** in the same logical universe. The rule language we are now going to define will explicitely consider nested predicates and support direct queries and rules.

We shall extend the above natural syntax to a rule language for complex objects, called NESTED_DATALOG. We now define the syntax of NESTED_DATALOG. As COL [Abiteboul87], NESTED_DATALOG is a strongly typed language : We start with sets of atomic objects which are the basic types. As usual, we use the following symbols :

(1) **Constants** : a, b, c, ... filler '_';
(2) **Variables** : x,y,z, ...;
(3) **Type predicates** : p, q, r ...;
(4) **Comparison predicates** : =, <,≤,>,≥, ...;

(5) **Logical connectors** : $\wedge$, $\vee$, $\neg$, $\Rightarrow$.

A **simple term** is defined as being either a variable or a constant. For example, 1 and x are terms.

The novelties of the language are to support nested predicates (which indeed exists already in PROLOG or COL, but in that languages, internal predicates are considered as functions) and to be strongly typed. Types and Complex terms may be defined as follows.

Definition 2.1 : Type

A type is defined recursively as :

(a) Any type predicate is a type.

(b) If $p_1, p_2, \ldots p_n$ are types then $p(p_1, p_2, \ldots p_n)$ is a type (called a tuple type).

(c) If p' is a type then p ({p'}) is a type (called a set type).

Definition 2.2 : Complex Term

A complex term is defined recursively as :

(a) If $t_1, t_2, \ldots, t_n$ are simple or complex terms and p is a n-place type predicate, then $p(t_1, t_2, \ldots, t_n)$ is a complex term (called a tuple term).

(b) If $t_1, t_2, \ldots, t_n$ are simple or complex terms and p is a unary type predicate, then $p(\{t_1, t_2, \ldots, t_n\})$ is a complex term (called a set term).

A term t tagged explicitly with a type symbol p, as p(t), is called a strongly typed term. The above definition requires complex terms to be strongly typed. For example, a typed term p({q({x}) }) is a legal NESTED_DATALOG term, but neither {{x}} nor p({{x}}) are legal with NESTED_DATALOG since not strongly typed.

With the notion of complex term, one can construct a set of syntactically well formed formulas (in short, formulas) as follows :

Definition 2.3 : Formula

(1) If p is a type predicate or a comparison predicate and $t_1, t_2, \ldots, t_n$ are complex terms, then $p(t_1, t_2, \ldots, t_n)$ is an atomic formula.

(2) Atomic formulas are well formed formulas.

(3) If F1 and F2 are well formed formulas, so are F1,F2 and $\neg$F1.

Definition 2.4 : Rule

A **rule** is defined as **head** ← **body**, where the body is a formula and the head is an atomic formula.

Definition 2.5 : Program

A **program** is a finite set of rules.

Definition 2.6 : Query

A **query** is a rule without a head, denoted as **?body**.

For example, against the complex object :

registration (cs, 1987, courses (
{course (231, db, prof (smith, male, 36),
students ({student (john, 17), student (jan, 18), student (hull, 20)})),
course (171, os, prof (smith, male, 36),
students ({student (lee, 18), student (jan, 18), student (hull, 20)})),
course (281, ai, prof (smith, male, 36),
students ({student (john, 17), student (lee, 18), student (hull, 20)})) })).

one may define the rule

young_instructor (x, names{z}) ← registration (x, _, courses ({course (n, _, prof (z, _, g), _)})),
n > 200, g < 40.

where the the young_instructor predicate as defined above gives the set of names of all the young professors (age < 40) instructing courses such that (course_number > 200) in the department x.

3. MAPPING COMPLEX FACTS TO FLAT FACTS

As mentioned above, NESTED_DATALOG is implemented on top of an enhanced relational model, called the **token object model**. In this paper, for reasons of conciseness, we also consider that the semantics of NESTED_DATALOG is defined through its implementation, although it could be defined in another way (e.g., by considering nested predicates as functions and by completing the rule program with unesting rules). Thus, it is desirable to formally defined the mapping of NESTED_DATALOG programs to the implementation model for their FOL-based evaluation. In this

section, we describe the representation of facts with the token object model. We also present the mapping from NESTED_DATALOG facts to token object model facts.

3.1 Token Object Facts

In general a complex object is constructed progressively from more primitive lower level objects, thus it has a configuration hierarchy. The representation of a complex object may refer other objects or be referenced by still complex objects. To identify complex objects properly and uniquely is important both for maintaining the system consistency and for supporting the information referential sharing. Using surrogate [Meier83] or token [Woelk86] to represent abstractly an object with complex structure is beneficial from both implementation point of view and conceptualization point of view. In short, a token is an atomic object of a predefined type which uniquely identifies a complex object and stands for the whole configuration structure of the object. For simplicity and convenience, an atomic object needs not be associated with an additional token: It is identified by its own value.

The token object model consists in representing all complex object with **flat relations** (i.e., FOL predicates) and with **tokens**. Moreover, we support two special domains :

(i) One level **sets of tokens.**

(ii) One level **tuples of tokens.**

These two domains are the only structures allowed to represent the links between components of complex objects. Similarly to the notions given in [Koshafian86] we can define formally the following two mappings.

Definition 3.1: Token Mapping

The Token Mapping τ maps an object **O** to its token τ**O** which is an atomic object identifying object **O**.

Thus, mapping τ simply gives the token of an object which belongs to an atomic type, the token type.

Definition 3.2: Token Value Mapping

The Token Value Mapping ρ maps an object **O** to its token value ρ**O** non-recursively defined as :

(a) For an atomic object O, $\rho O = \tau O = O$;

(b) For a tuple object $\rho(O_1, ..., O_n) = (\tau O_1, ..., \tau O_n)$

(c) For a set object $\rho\{O_1, ..., O_n\} = \{\tau O_1, ..., \tau O_n\}$

(d) $\rho(\rho O) = \rho O$

Mapping ρ gives a structure of tokens which is a complex type. Indeed, it decomposes an object into its elements and returns a structure which is composed of the element tokens.

Some interesting properties of these two mappings can be summarized as

$\tau(\tau O) = \tau O$ $\quad\quad$ $\rho(\rho O) = \rho O$

$\tau(\rho O) = \tau O$ $\quad\quad$ $\rho(\tau O) = \tau O$

3.2 Mapping Complex Facts to Token Object Facts

3.2.1 An informal presentation

How to map complex objects to the token object model, that is to a logical relational database framework [Reiter84] with tokens, is the key issue to define a simple and clean implementation of reasoning about complex objects. To illustrate the approach, we shall present examples. In the examples, a symbol with a star such as **s*** stands for a **token**. In general, the mapping of complex objects to tokens and relations is characterized by the following rules :

(1) Each object O is represented by two separate elements : Its token τO and its token value ρO.

The token τO is used to refer to O in complex objects including O. Thus, the fact that object A is a component of object B is represented by referring the token of A in the representation of B, that is, by including the token of A as a parameter of term B. This philosophy is applied level by level along the configuration hierarchy of any complex object. Thus, in a normal token model object, there exist only single-level, unnested set terms and unnested tuple terms, namely, all the elements of any set are atomic.

The token value ρO is used to define the object content at a lower level than the token. For complex objects embedded in O such as set and tuples, the tokens included in the token value ρO refer to their representation one level below.

Token values are predicated by the object type. That is, we use the type name as predicate symbol on the terms sharing the type, such as with tuples :

```
prof (smith, p1*, 36)
prof (hoare, p2*, 49)
```

or with sets:

```
courses ({c1*, c2*, c3*})
```

Indeed, the standard form of the latter is :

`courses (c*)` where `c*` is the token for `{c1*, c2*, c3*}`.

Some predicates may have only one parameter, which can be an atomic term such as

```
man(john).
```

In application to the above rule, an atomic object is represented by a simple term, such as `smith`, as there is no difference between its token and its token value. Thus, we can consider that an atomic object is also represented by its token and its token value. As we do not allow duplicates, it is easy to see that an atomic object need not be additionally identified: It is identified by itself.

A tuple object is represented by a token if it is an element of another object and by a tuple of tokens or simple terms for its token value.

For instance, student in :

```
prof(smith, male, 36, student (john,17))
```

will be represented by :

```
prof(smith,male,36,s*) and student (john,17)
```

where s* is the token for `(john,17)`.

A set object is represented by a set term for its token value, such as :

```
courses({c1*, c2*, c3*}),
```

or a token (atomic term) if it is an element of another object, such as `ss3*` in

```
course(281,ai,p*,ss3*).
```

(2) Token-predicates are introduced to represent the relationship between the token and the token value of a non-atomic object.

Thus, token predicates associate the abstract description of an object at a higher level and the more detailed description of it at one level below. Token predicates are named with the object name preceded by a "*", such as `*prof` .The first parameter of this predicate must be a token term. All the token terms are atomic terms. Below are some examples.

```
*prof (p*, (smith, male, 36)).
*courses (c*, {c1*, c2*, c3*}).
```

note that `c*` is the token for the set `{c1*, c2*, c3*}` with three other tokens as its elements.

In summary, the predicated token value and the token predicate associated to the same object form a pair which define the value and the identifier of a complex object (It does not mean that the value should be physically duplicated: The predicated token value should be defined as a view of the token predicate). Examples are :

```
prof (smith, male, 36).        *prof (p*, (smith, male, 36)).
courses ({c1*, c2*, c3*}).     *courses (c*, {c1*, c2*, c3*}).
```

The token value predicates can be involved in any reasoning at the appropriate level; the token-predicates provide the transitions between levels.

3.2.2 A formal definition

The mapping of facts from NESTED_DATALOG to the token object model is essentially a replacement of the complex objects by the corresponding token objects. It may be performed in a bottom-up manner, starting from the innermost predicates. In general, each complex object x is replaced by a token τx, while the token for an atomic object is just itself. The token value for an atomic object x is also itself, as $\rho x = x$. Let us recall that by definition, for $x = p(x_1, ..., x_n)$ $\rho x = p(\tau x_1, ..., \tau x_n)$, for $x = \{x_1, ..., x_n\}$ $\rho x = \{\tau x_1, ..., \tau x_n\}$. Based on this concept the above mapping is defined as following.

Let x be a complex term of NESTED_DATALOG and **u** be a set of DATALOG facts, the mapping

$$\omega_F : x \rightarrow u$$

is called a **fact rewriting** transformation. Fact rewriting may be defined as following.

Definition 3.3 : Fact rewriting

Set Ω_F of fact rewriting transformations defined recursively as :

(a) $x \rightarrow x \in \Omega_F$ where x is an atomic object;

(b) if $x = pT$ with $T = (x_1, ..., x_n)$

then $x \rightarrow \{ p(\tau x_1, ..., \tau x_n). \ *p(\tau T, (\tau x_1, ..., \tau x_n). \} \in \Omega_F$

(c) if $x = p(S)$, $S = \{x_1, ..., x_n\}$

then $x \rightarrow \{ p(\tau S). \ *p(\tau S, \{\tau x_1, ..., \tau x_n\}). \} \in \Omega_F$

The rewrite result is the union of all the sets of clauses created by applying the set Ω_F of fact rewriting transformations to the structure of the complex term in the innermost order (bottom-up). The semantics of set union implies the removing of duplication. The order of applying the above mapping rules to the nested object structures is important since token objects have level-independent representation. A top-down NESTED_DATALOG to the token object model conversion will drop the lower level information. A fact rewriting is valid if all the tokens introduced are distinct.

3.2.3. An example

The stepwise conversion of facts (innermost first) can be shown as following.

(stepwise results of fact rewriting (innermost first))
Complex fact

registration (cs, 1987, courses (
{course (231, db, prof (smith, male, 36),
students ({student (john, 17), student (jan, 18), student (hull, 20)})),
course (171, os, prof (smith, male, 36),
students ({student (lee, 18), student (jan, 18), student (hull, 20)})),
course (281, ai, prof (smith, male, 36),
students ({student (john, 17), student (lee, 18), student (hull, 20)})) })).

step 1

registration (cs, 1987, courses (
{course (231, db, p*, students ({s_1*, s_2*, s_3*})),
course (171, os, p*, students ({s_4*, s_2*, s_3*})),
course (281, ai, p*, students ({s_1*, s_4*, s_3*})) })).

with the following DATALOG clauses generated:

prof (smith, male, 36).
prof (p, (smith, male, 36)).
student (john, 17).
student (s_1, (john, 17)).
student (jan, 18).
student (s_2, (jan, 18)).
student (hull, 20).
student (s_3, (hull, 20)).
student (lee, 18).
student (s_4, (lee, 18)).

step 2

registration (cs, 1987, courses (
{course (231, db, p*, ss_1*),
course (171, os, p*, ss_2*),
course (281, ai, p*, ss_3*) })).

with the following additional DATALOG clauses generated:

students (ss_1*).
students (ss_1, {s_1*, s_2*, s_3*}).
students (ss_2*).
students (ss_2, {s_4*, s_2*, s_3*}).
students (ss_3*).
students (ss_3, {s_1*, s_4*, s_3*}).

step 3

registration (cs, 1987, courses ({ c_1*, c_2*, c_3* })).

with the following additional DATALOG clauses generated:

course (231, db, p*, ss_1*).
course (c_1, (231, db, p*, ss_1*)).
course (171, kb, p*, ss_2*).
course (c_2, (171, kb, p*, ss_2*)).
course (281, ai, p*, ss_3*).
course (c_3, (281, ai, p*, ss_3*)).

step 4, 5

registration (cs, 1987, c*).

with the following additional DATALOG clauses generated:

courses (c*). *courses (c*, {c_1*, c_2*, c_3*}).

registration (cs, 1987, c*). *registration(r*, (cs, 1987, c*)).

4. MAPPING COMPLEX RULES TO FLAT RULES

4.1 Rule Language for the Token Object Model

The proposed rule language for the Token Object Model is an extension of DATALOG with negation and sets. Since we must handle one level sets and negations, we use a version of DATALOG able to handle correctly these features. Thus, we assume stratified DATALOG programs where sets and negated predicates are computed before using them [Apt87]. The notations given in [Beeri86] for representing set-oriented manipulation, such as <x> for grouping, and **member (x, X)** for set membership are also adopted. We also extend the grouping notation to allow tuple grouping, such as <(x,y)>. We assume here that the semantics of such a language is well understood [Gardarin 87b].

The transition of inferences at different levels of the object configuration hierarchy is made by passing tokens through the token-predicates. This mechanism supports both top-down reasoning and bottom-up reasoning along the object configuration hierarchy. In an actual logic program, the direction of reasoning along the object hierarchy may not be fixed, but mixed with up and down evaluation.

In the case of top-down reasoning, suppose that a complex object A is a component of an even more complex object B. A is abstractly represented as a token term which is a parameter of the term for B. When detailed information of A is needed, the program passes from the token level of A to the token value level of A in terms of the token-predicates, and receives another term with more information. The same evaluation principle applies to the components of A as well, and step by step down along the object hierarchy. For instance, starting with the predicate

```
course (281, ai, p*, ss3*).
```

the token p* of the professor teaching course 281 is obtained, then through

```
*prof (p*, (smith, male, 36)).
```

the program can pass to a lower level and receive more detailed information about the professor.

In the case of bottom-up reasoning, one can start with certain known information about an object, use this information to find the corresponding token of the object, and then pass to the upper

level where the token works as a parameter of other higher-level terms. For instance, in the above example the query "which course professor smith teaches ?" can be processed by starting with

```
*prof (p*, (smith, male, 36)).
```

to receive the token p*, then pass to

```
course (281, ai, p*, ss3*).
```

and get the course number 281.

It is worth noting that by modeling objects in the way mentioned above, and by introducing the special token-predicates, the token passing evaluation can be carried out via the usual DATALOG inference mechanism. A rule program is actually stratified in the sense of "computing a token before using it", and as usual, "computing a set and a negated predicate before using it". Associated with each DATALOG program, a **Token Passing Graph** can be given showing the stratification of the evaluation.

To illustrate this approach, let us see how it works on the example given before. In the expressions given below, a capital symbol stands for a set; a symbol with a star such as s* stands for a token.

Facts (ground formulas)

registration (cs, 1987, c*).
courses (c*).
course (231, db, p*, ss_1*).
course (171, kb, p*, ss_2*).
course (281, ai, p*, ss_3*).
prof (smith, male, 36).
students (ss_1*) .
students (ss_2*) .
students (ss_3*) .
student (john, 17).
student (jan, 18).
student (hull, 20).
student (lee, 18).

registration (r, (cs, 1987, c*)).
courses (c, {c_1*, c_2*, c_3*}).
course (c_1, (231, db, p*, ss_1*)).
course (c_2, (171, kb, p*, ss_2*)).
course (c_3, (281, ai, p*, ss_3*)).
prof (p, (smith, male, 36)).
students (ss_1, {s_1*, s_2*, s_3*}).
students (ss_2, {s_4*, s_2*, s_3*}).
students (ss_3, {s_1*, s_4*, s_3*}).
student (s_1, (john, 17)).
student (s_2, (jan, 18)).
student (s_3, (hull, 20)).
student (s_4, (lee, 18)).

Rules

(a) Find the set of student names Y professor x (name) teaches on course c (course number).

teach (x, c, <y>) :- *prof (t_1, (x, _, _)), course (c, _, t_1, t_2), *students (t_2, S),
member (s, S), *student (s, (y, _)).

?- teach (smith, 231, Y) succeeds with unifier {Y/{john, jan, hull}}.

(b) Find student lee's age.

?- student (lee, x). succeeds with unifier {x/18}.

(c) Get the name set Z of all the young professors (age < 40) instructing graduate level courses (course_number > 200) in the department x.

young_instructor (x, <z>) :- registration (x, _, t_1), *courses (t_1, C), member (t_2, C),
*course (t_2, (n, _, t3, _)), *prof (t_3, (z, _, g)), n > 200, g < 40.

?- young_instructor (cs, Z) succeeds with unifier {Z/{smith}}.

From the above example, we can see that the proposed token passing approach can be employed as a simple technique for building deductive databases on complex objects. It firmly stays within the logical database framework (indeed, DATALOG with stratification). It does not refer any nested predicate. It provides links among different parts of a complex object and expands the aggregation hierarchy to a level-independent 'flat' scheme. It accommodates in the LP system the philosophy of sharing support found in conventional data management. It has the capability of handling mutually nested sets and tuples. It simplifies the semantics of LP with sets by removing the nesting of sets. More importantly, these features can easily be integrated to existing relational DBMSs supporting DATALOG. Since tokens are handled as atomic terms without special treatment, the proposed system can also support classical applications without using tokens.

4.2. The Mapping of Rules

In this section, we only consider simple head. A simple head contains at most one level nested predicates with set arguments. Moreover, nested predicate in a simple head should not appear in the rule body. Indeed, up to section 5, we only allow one level nesting for sets in head. Further nesting in head (e.g., nested tuples and nested recursion) will be considered in section 5 of this paper.

Let v be a NESTED_DATALOG rule head. A mapping

$$\omega_H : v \rightarrow u$$

which maps v to a corresponding DATALOG elements in the token object model is called a **head rewriting** transformation.

Definition 4.1 : Head Rewriting

Set Ω_H of head rewrite transformations from NESTED_DATALOG to DATALOG defined recursively as :

(a) $v \rightarrow v \in \Omega_H$ where v is a constant, a symbol or a filler;

(b) if v is a variable, then $p(\{v\}) \rightarrow <v> \in \Omega_H$

(c) if (v1,v2,...vn) is a tuple, then $p(\{(v1,v2,...vn)\}) \rightarrow <(v1,v2,...vn)> \in \Omega_H$

(d) if $v_1 \rightarrow u_1$, ..., $v_n \rightarrow u_n \in \Omega_H$,
then $p(v_1, ..., v_n) \rightarrow p(u_1, ..., u_n) \in \Omega_H$

Similarly, let v be a NESTED_DATALOG atomic formula appearing in a rule body, u a DATALOG expression in the token object model, the mapping

$$\omega_B : v \rightarrow u$$

is called a **body rewriting** transformation.

Definition 4.2 : Body Rewriting

Set Ω_B of body rewriting transformations from NESTED_DATALOG to DATALOG defined recursively as :

(a) $v \rightarrow v \in \Omega_B$ where v is a constant, a symbol or a filler;

(b) if $v_1 \rightarrow u_1$, ..., $v_n \rightarrow u_n \in \Omega_B$,
then $p(v_1, ..., v_n) \rightarrow *p(t, (u_1, ..., u_n)) \in \Omega_B$

(c) if $v \rightarrow u \in \Omega_B$, then $p(\{v\}) \rightarrow *p(t, \{u\}) \in \Omega_B$

where t's are token variables.

A body rewriting is valid if all the token variables introduced are distinct.

Body rewriting is not sufficient to get a DATALOG body: It only introduces tokens and token values in a single expression. We further need to expand this multi-level expression to a conjunction of DATALOG predicates; also, we must introduce the member predicate to handle sets. The expansion applies in a top-down, "outermost first" and "depth-first" fashion, step by step unquoting the expression from the outer level to the inner level, while generating lists of single-level predicates from the left to the right. The final list of predicates is yielded by the bottom-up, innermost-first

concatenation of these lists. We shall give a definition by an algorithm for the expansion.

Definition 4.3 : Expansion

Let **u** be the hierarchically structured expression resulting from the body rewriting of a NESTED_DATALOG body formula, **C** be the conjunction of a list of DATALOG predicates separated by commas, the mapping :

$$\varepsilon : u \rightarrow C$$

defined by the BODY_EXPANSION algorithm is called an **expansion.**

Below we give the BODY_EXPANSION algorithm. To be clear we shall use <First | Rest> to represent the first and the rest elements of a list, and use capital character to represent variables for sets; the type symbols with or without * precedence are not distinguished unless especially marked, thus a type symbol can mean p or *p.

[Algorithm BODY_EXPANSION]

Let **u** be the hierarchically structured expression resulted from the body rewriting of a NESTED_DATALOG body variable, the body rewrite expansion of **u** includes the following steps :

(1) Replace the outermost ***p (t, (u))** by **p (u),** and ***p (t, {u})** by **p (*p (t, {u}))**.

Example : *p(t, (q(x,y), z)) is replaced by p(q(x,y), z) .

(2) Replace the outermost $p(u_1, ..., u_n)$ by $p(s_1, ..., s_n), q_1, ..., q_n$ where for each u_i, if

(a) u_i is an atomic object, a variable or a filler, then $s_i = u_i$ and $q_i = \varnothing$;

(b) $u_i = p_i (t_i, (u'_i))$, where (u'_i) represents a tuple, then $s_i = t_i$, and $q_i = p_i (t_i, (u'_i))$;

(c) $u_i = p_i (t_i, \{u'_i\})$, then $s_i = t_i$, and $q_i = p_i (t_i, S_i)$, member (u'_i, S_i) , where S_i is a variable for set.

This step transforms a hierarchical expression to the conjunction of a list of expressions, while the first one is a "well formed" predicate.

Example 1 : *p (x, *q(t_1, (a,b)), *r(t_2, {y})) is replaced by

*p (x, t_1, t_2), *q(t_1, (a,b)), *r(t_2, Y), member (y, Y) .

Example 2 : member (y, Y) is replaced by itself.

Example 3 : member (*p(t, (x, a)), Y) is replaced by member (t, Y), *p(t, (x, a)) .

We denote the conjunction of a list of expressions, transformed from the hierarchical expression shown above, as <First | Rest>, the First is a "well formed" predicate and to be appended to the right end of the predicate list yielded; then each expression of the Rest is to be applied by the same mapping described above. This process is to be propagated in a "depth-first" fashion along with each expression, until no new expressions generated.

(3) The final resulted list of predicates is the level by level, bottom-up (innermost-first) unquoting and concatenation of all the previously resulted predicate lists.

Thus if we denote the mapping described in (1) as ε_1, the mapping described in (2.a, 2.b, 2.c) as λ, and the whole mapping described in (2) as ε_2, the mapping described in (3) as ε_3, by using the FP functional form notation given in [Backus78], it is clear that

$$\varepsilon = \varepsilon_3 \cdot \varepsilon_2 \cdot \varepsilon_1$$

$$\varepsilon_2 = [\textbf{First}, \alpha(\varepsilon_2) \cdot \textbf{Rest}] \cdot \lambda$$

where First and Rest are viewed as functions, meaning get the first and rest elements of a list; function forms '.' means composition, 'α' means apply-to-all, '[ ]' means list construction (e.g.,[f, g] : x = <f:x, g:x>).

A body rewrite expansion is valid if all the set variables introduced are distinct. We have now all necessary elements to define rule rewriting.

Definition 4.4: Rule Rewriting

A **Rule Rewriting** from a NESTED_DATALOG rule

R_C : $head_C$ ← $body\text{-}variable_C$, condition expressions

to a DATALOG rule

R_1 : $head_1$:– $body_1$

consists of the follows :

(a) a rewriting from the $head_C$ to the $head_1$;

(b) a transformation of symbol "←" to ":-";

(c) a valid rewriting from $body\text{-}variable_C$ to the intermediate expression u, followed by a valid rewrite expansion from u to the list of predicates in the $body_1$;

(d) add condition expressions to the $body_1$.

A rule rewrite is valid if all the above transformations are valid.

[Example]

NESTED_DATALOG rule

young_instructor (x, names({z})) ← registration (x, _, courses ({course (n, _, prof (z, _, g), _)})),
n > 200, g < 40.

head rewriting

(head)

young_instructor (x, names{z})

(result of head rewrite)

young_instructor (x, <z>)

body rewriting

(body)

registration (x, _, courses ({course (n, _, prof (z, _, g), _) }))

(result of body rewriting)

*registration (t_0, (x, _, *courses (t_1, {*course (t_2, (n, _, *prof (t_3, (z, _, g)), _)) }))))

stepwise results of body expansion

registration (x, _, *courses (t_1, {*course (t_2, (n, _, *prof (t_3, (z, _, g)), _)) }))

registration (x, _, t_1), *courses (t_1, {*course (t_2, (n, _, *prof (t_3, (z, _, g)), _)) }))

registration (x, _, t_1), *courses (t_1, C),
member (*course (t_2, (n, _, *prof (t_3, (z, _, g)), _)), C)

registration (x, _, t_1), *courses (t_1, C),
member (t_2, C), *course (t_2, (n, _, *prof (t_3, (z, _, g)), _))

registration (x, _, t_1), *courses (t_1, C),
member (t_2, C), *course (t_2, (n, _, t3, _)), *prof (t_3, (z, _, g))

DATALOG rule with tokens

young_instructor (x, <z>) :- registration (x, _, t_1), *courses (t_1, C), member (t_2, C),
*course (t_2, (n, _, t3, _)), *prof (t_3, (z, _, g)), n > 200, g < 40.

5. FURTHER DISCUSSIONS ON THE SEMANTICS OF NESTED_DATALOG

5.1 Strong Typing of NESTED_DATALOG

NESTED_DATALOG is a strongly typed language. The mapping from a NESTED_DATALOG fact or rule to the set of DATALOG clauses with tokens essentially consists in the conversion of objects to tokens and token values, and the conversion of types to predicates, therefore transforming all the nested terms into predicates, referred to as **fully predicated**. Indeed, to have a NESTED_DATALOG fact or rule to be computable based on the token object model requires the converted set of DATALOG clauses to be fully predicated, and to have the resulted set of DATALOG clauses to be fully predicated requires that the original NESTED_DATALOG fact or rule to be **strongly typed.**

For example, a typed NESTED_DATALOG term

p({ q({x}) })

corresponding to the two level set type p({ q({q'}) }) can be mapped through the body rewrite and expansion rules to the conjunction of a list of DATALOG clauses with tokens

$*p(t_1, \{ *q(t_2, \{x\}) \})$	by body rewrite
$p(t_1), *p(t_1, P), member(t_2,P), *q(t_2,Q), member(x, Q)$	by body rewrite expansion

However, neither {{x}} nor p({{x}}) can be mapped to a set of fully predicated DATALOG clauses with tokens, through the mapping rules given above, since they are **not strongly typed** thus **cannot have fully predicated** images in the DATALOG universe. This is why they are not legal NESTED_DATALOG terms and cannot be evaluated on the DATALOG framework with tokens.

5.2 Tolerant Reconstructions

It is obvious that the body formula of a NESTED_DATALOG rule must belong to a type of objects actually defined in the database, or a virtual type constructed from actual types. To guarantee the head formula to have interpretations associated with the conditions specified in the body, it must belong to a type which is properly reconstructed from the type of the body formula.

The type reconstruction has been an interesting issue for a time. However, reconstructions can be made on different purposes. For information retrieval, we need the results of reconstructions to have a tolerant structure for holding the information obtained from the original structure. For

example, the rewrite from type p ({p'}) to type p' may not be a structure tolerant reconstruction since the first type has higher level set nesting than the second type. Below we define a restricted type reconstruction called **Tolerant Reconstructions** which guarantees the rewrite results be the assembles of the parts from the original type. The renaming problem is not to be particularly mentioned; however, the definition given below may be extended with more renaming (i.e., a predicate name can be replaced by a new predicate name).

Definition 5.1 : Tolerant reconstruction of types

Let types be represented as (a) p; (b) p (p_1,..., p_n); and (c) p ({p'}) where recursively p' and each p_i are types, the set $\Pi(p)$ of **Tolerant Reconstructions** of type p is defined recursively as

(a) $p \in \Pi(p)$.

(b) for p (p_1,..., p_n), $\forall i \in \{1,..., n\}$ $p_i \in \Pi(p)$.

(c) for p (p_1,..., p_n), $\forall i \in \{1,..., n\}$ $p_i \in \Pi(q) \Rightarrow p \in \Pi(q)$.

(d) for p ({p'}) and q ({q'}), $p' \in \Pi(q') \Rightarrow p \in \Pi(q)$.

(e) $p_1 \in \Pi(p_2) \wedge p_2 \in \Pi(p_3) \Rightarrow p_1 \in \Pi(p_3)$.

From the above definition we can see that the reconstruction of a type **p** may either be smaller or bigger than **p**, but all the parts of the reconstructed type are obtained from **p**. For example, given the simple set type p({q(q_1,..., q_n)}) (set of tuples), the following tolerant transformations will rewrite it to a tuple type (tuple of sets) with an appropriate structure to hold the information from type p :

since for i = 1,...,n , $q_i \in \Pi(q)$ (case **b** of the above definition),
then $\{q_i\} \in \Pi(p)$ (case **d** of the above definition),
then $(\{q_1\}, ..., \{q_n\}) \in \Pi(p)$ (case **c** of the above definition).

5.3 Determinism of NESTED_DATALOG Programs

In this section, we would like to study the determinism of the semantics of a NESTED_DATALOG program. This is a major issue which has already been well addressed within the DATALOG framework. Let us recall that a DATALOG program with negated predicates and sets is said to be **stratifiable** if no recursion crosses negation or sets. In that case, the program has a unique minimal model for all instances of the EDB which may be computed strata per strata [Apt87]. With the semantics given to NESTED_DATALOG, it is simple to state the following proposition :

Definition 5.2 : NESTED_DATALOG Stratification

A NESTED_DATALOG program is stratifiable iff its mapping to DATALOG is

stratifiable.

To guarantee that the mapping of a NESTED_DATALOG program to be safe, one must guarantee that predicates are nested according to a given order, that is, the stratification according to nesting is required. It is then possible to demonstrate that a sufficient condition for a NESTED_DATALOG program to be safe is that the following properties holds :

(a) Each rule is strongly typed.

(b) For each rule, the type of the head is a tolerant reconstruction of the types of the body.

(c) The program can be stratified according to sets and negations.

5.4 Variable Scope of NESTED_DATALOG and Extended Reasoning

In pure DATALOG, there is only one sort of variables, the atomic variables. In certain DATALOG-like language with sets, such as in LPS [Kuper 87], there are two sorts of variables, the atomic variables and the set variables. In the DATALOG with tokens described in this paper, three sorts of variables are introduced, they are the atomic variables, the set variables, and the tuple variables. However, in the NESTED_DATALOG, in principle there is no need to restrict the scope of variables, they can be atomic, or mutually nested tuples and sets of any order.

In a NESTED_DATALOG query, if a variable is an atomic one, the answer to the query under the token passing approach will return the actual value as the substitution to the variable. However, if a variable is a non-atomic one, the normal query processing terminates with a token for the variable returned. For example, let us consider the NESTED_DATALOG fact

r (c, q({p(a, 1), p(b, 2)}))

which is transformed to the following clauses of DATALOG with tokens :

r(c, t_0*).	*r(t*, (c, t_0*)).
q(t_0*).	*q(t_0*, {t_1*, t_2*}).
p(a, 1).	*p(t_1*, (a, 1)).
p(b, 2).	*p(t_2*, (b, 2)).

The following NESTED_DATALOG query with an atomic variable x

r'(x) ← r (c, q({p(x, 1)}))

can be transformed into the query of DATALOG with tokens

r'(x) :- r(c, t), *q(t, S), member(s, S), *p(s, (x, 1)).

and has the answer x = a.

However, the following NESTED_DATALOG query with a set variable x

r'({x}) ← r (c, q({x}))

is transformed into the query of DATALOG with tokens

r'(<x>) :- r(c, t), *q(t, X), member (x, X).

and has the answer X = {t_1*, t_2*}, which is just a set of tokens. If tokens are made transparent to users such an answer may not be considered as the appropriate termination of the query processing.

To solve this problem we have developed the notion called **Extended Reasoning**, that is, if the query processing turns out to an answer with tokens involved, those tokens will be replaced step by step by their associated token values, until no token occurrence in the derived answer. Thus the above query will have the final answer of

X = {p(a, 1), p(b, 2)}

The extended reasoning is handled automatically by the system and need not be expressed explicitly in the user's program. It includes the following steps (we suppose that tokens form a special type of constants which can be recognized by the system) :

(1) For each token t* occurring in the answer, search the token predicates *g(t*, u), where g is any predicate symbol. Then replace t* by u.

(2) If u is still a token or a construct including tokens, do (1) again for each token appearing in u; otherwise, return g(u) as the replacement of t* in the final answer.

5.5. The Handling of Complex Head Formula

To restrict the head formula of a NESTED_DATALOG to a simple formula with only one level of set nesting, as mentioned before, is necessary since under the token passing approach, a complex head formula with nested structure would be mapped to more than one formula with tokens. As this is not allowed by the definition of Horn clause, the result would not be computable in the framework of DATALOG with tokens. In this section, we show that the above restriction does not limit the powerfulness of the proposed mapping. The general idea can be described as following.

Suppose a NESTED_DATALOG rule containing the set of variables {x_1, x_2, ..., x_k} of any sorts in the head and the set of variables {x_1', x_2', ..., x_n'} in the body, is represented as

head (x_1, x_2, ..., x_k) ← **body** (x_1', x_2', ..., x_n').

where {x_1', x_2', ..., x_n'} ⊃ {x_1, x_2, ..., x_k} . If this rule succeeds wrt the EDB, with substitution θ for the set of variables x_1, x_2, ..., x_k, then the **head** is considered as the construction with the substitution θ for its variables. That means we can introduce a simple atomic formula called **abstract head formula** denoted as

$H(x_1, x_2, ..., x_k)$

and receive the variable substitutions through evaluating

$H(x_1, x_2, ..., x_k) \leftarrow$ **body** $(x_1', x_2', ..., x_n')$

and then introduce a nesting rule :

head $(x_1, x_2, ..., x_k) \leftarrow H(x_1, x_2, ..., x_k)$

to reconstruct the rule head.

Definition 5.3 : Nesting rule

Given a flat atomic formula $H(x_1, x_2, ..., x_k)$ and a nested rule head **head** $(x_1, x_2, ..., x_k)$, a nesting rule constructs the complex facts **head** $\theta(x_1, x_2, ..., x_k)$ for every possible substitution θ such that $\theta(x_1, x_2, ..., x_k)$ holds in H.

Let us recall that given the rule

head $(x_1, x_2, ..., x_k) \leftarrow$ **body** $(x_1', x_2', ..., x_n')$,

such that $\{x_1', x_2', ..., x_n'\} \supset \{x_1, x_2, ..., x_k\}$ and the head is the tolerant reconstruction of the body, the set of substitutions for the variables $x_1, x_2, ..., x_k$ in the **head** contains and only contains the substitutions of the same variables in the **body** wrt the EDB. Based on this rule semantics, we can introduce a mapping for complex heads. For the rule shown above, let **B'** be the list of DATALOG clauses with tokens mapped and expanded from **body**, and **H** (x1, x2, ..., xk) the abstract head formula introduced, for any substitution of $x_1, x_2, ..., x_k$ wrt the EDB,

if **H** (x1, x2, ..., xk) $\leftarrow$ **B'** holds

then **head** (x1, x2, ..., xk) $\leftarrow$ **H**(x1, x2, ..., xk) holds.

A proof skeleton may be given as follows :

1. Let θ be any substitution for $x_1, x_2, ..., x_k$ wrt the EDB,

   H $(x_1, x_2, ..., x_k) \leftarrow$ **B'**

   holds means θ**B'** $\in$ EDB and θH $\in$ IDB.

2. θ is a substitution for the variables in **body** wrt the EDB.

3. θ is also a substitution for the variables in **head** wrt the IDB, and

   θ**(head)** $\leftarrow$ θ**(body)**

Some rules may be without bodies, such that

course (231, db, prof (x, _, _), _) .

which are evaluated directly against the facts of the EDB with the semantics defined below. To map such rules to the target model, we must first rewrite them as follows : Rule **r** $(x_1, x_2, ..., x_k)$ is converted to the rule

$r\ (x_1, x_2, ..., x_k) \leftarrow r\ (x_1, x_2, ..., x_k)$

which can enter the general framework defined above.

Finally we demonstrate the handling of complex head by the following example.

Rule with complex head

cos (x, prof (pn, pa), stds ({std(sn, sa)})) ←

course (x, _, prof (pn, _, pa), students ({student (sn, sa) }))

Query

? cos (x, prof (pn, pa), stds ({std(sn, sa)}))

Step 1 : Introduce simple parameterized head formula

p (x, pn, pa, {(sn, sa)}) ← course (x, _, prof (pn, _, pa), students ({student (sn, sa) }))

Step 2 : Mapping to DATALOG with Tokens

...

Step 3 : Evaluating

In this step the above query has an answer substitution

{231/231, pn/smith, pa/36, {(sn, sa)}/{(john, 17), (jan, 18), (hull, 20)}}

Step 4 : constructing the head with the above substitution

course (231, db, prof (smith, male, 36),

students ({student (john, 17), student (jan, 18), student (hull, 20)}))

6. CONCLUSION

In this work, we try to accommodate two basic requirements : A logic framework on which the consistent reasoning of complex objects is available, and a natural style of complex object rule language which represents hierarchical structures naturally. Thus, from theoretical point of view, we propose a natural non-1NF rule framework that can be mapped to the classical 1-NF rule framework. We formally and precisely defined the necessary mappings. From a practical point of view, that demonstrates the possibility to support engineering applications on top of relational systems (the performance issue is another question) .

In this paper, we started with describing a natural rule language called NESTED_DATALOG for dealing with complex objects and supporting the deductive retrieval and reconstruction of nested predicates. With introducing the Token Object Model, we show that it is possible to map NESTED_DATALOG to DATALOG on top of a relational DBMS supporting the token concept. By starting with NESTED_DATALOG and studying how hierarchically structured facts and rules can be converted to a set of "flat" clauses for the FOL-based reasoning, interesting properties of NESTED_DATALOG programs can be inferred. We present an example of such properties with stratification. Other interesting properties, such as safety, could be studied. To go further along this line will be a significant future work.

REFERENCES

[Abiteboul 87] S. Abiteboul, S. Grumbach, "Une Approche Logique de la Manipulation d'objets Complexes", 3e Journées BD3, Port Camargue, Mai 1987.

[Al-Amoudi87] Al-Amoudi S., Harper D., "On Compiling Logic Programming Languages into Conventional Relational Algebra Operations", Internal Report, University of Glasgow, UK, 1987.

[Apt86] Apt K.R., Blair H., Walker A. ,"Towards a Theory of Declarative Knowledge", IBM Research Report RC 11681, April 1986, in Proc. Workshop on Foundations of Deductive Databases and Logic Programming, Washington D.C., pp. 546,629, 1986.

[Backus 78] J. Backus, "Can Programming Be Liberated from the von-Neumann Style? A Functional Style and Its Algebra of Programs", CACM, vol.21, No.8, 1978.

[Bancilhon 86] F. Bancilhon and S. Khoshafian, "Objects Calculus", Proc. PODS, 1986.

[Beeri 86] C. Beeri et.al, "Sets and Negation in a Logic Database Language LDL1", MCC Tec. Rep, 1986.

[Chen 85] Q. Chen, "Extending the Implementation Scheme of Functional Programming System FP for Supporting the Formal Software Development Methodology", Proc. of 8th International Conference on Software Engineering, UK, 1985.

[Chen 86] Q. Chen, "A Rule-based Object/Task Modeling Approach", Proc. of ACM-SIGMOD 86, USA.

[Chen 87] Q. Chen, "An Extended Object-Oriented Database approach", Approach", Proc. of COMPSAC 87,

Japan, 1987.

[Gardarin 87a] G. Gardarin, "Magic Functions: A Technique to Optimize Extended Datalog Recursive Programs", proc. VLDB 13, 1987.

[Gardarin 87b] G. Gardarin, E. Simon, "Les Systèmes de Bases de Données Déductives", TSI, Dunod Ed., A paraître.

[Gallaire84] Gallaire H., Minker J., Nicolas J.M. : "Logic and databases : a deductive approach", ACM Computing Surveys, Vol. 16, N° 2, June 1984.

[Khoshafian 86] S. Khoshafian and G. Copeland, "Object Identity", Proc. OOPSLA 86, 1986.

[Kiernan87] Kiernan G., Morize I., "Building an Object-oriented Interface Over an Extended Relational Backend", Progress Report, ISIDE ESPRIT Project, N.221, Oct. 1987.

[Kuper 87] G. Kuper "Logic Programming with Sets", Proc. ACM PODS Conference, 1987.

[Meier 83] D. Meier and R. Lorie, "A Surrogate Concept for Engineering Databases", Proc. VLDB 9, 1983.

[Reiter84] Reiter R. : "Towards a Logical Reconstruction of Relational Database Theory", in On Conceptual Modelling, Book, Pp 191-234, Springer-Verlag Ed., 1984.

[Tsur 86] S. Tsur and C. Zaniolo, "LDL: ALogic Based Data Language", Proc. VLDB 12, 1986.

[Woelk 86] D. Woelk, W. Kim and W. Luther, "An Object-Oriented Approach for Multimedia Databases", Proc. ACM-SIGMOD 86, 1986.

[Zaniolo 85] C. Zaniolo, "The Representation and Deductive Retrieval of Complex Objects", Proc. VLDB 11, 1985.

Imprimé en France
par
l'Institut National de Recherche en Informatique et en Automatique