Integrating Object-Oriented Data Modeling with a Rule-Based Programming Paradigm

F. Cacace, S. Ceri, S. Crespi-Reghizzi, L. Tanca, R. Zicari
Dipartimento di Elettronica, Politecnico di Milano
Piazza L. Da Vinci, 32 20133 Milano, Italy

Abstract

LOGRES is a new project for the development of extended database systems which is based on the integration of the object-oriented data modelling paradigm and of the rule-based approach for the specification of queries and updates

The data model supports generalization hierarchies and object sharing, the rule-based language extends *Datalog* to support generalized type constructors (sets, multisets, and sequences), rule-based integrity constraints are automatically produced by analyzing schema definitions Modularization is a fundamental feature, as modules encapsulate queries and updates, when modules are applied to a LOGRES database, their side effects can be controlled

The LOGRES project is a follow-up of the ALGRES project, and takes advantage of the ALGRES programming environment for the development of a fast prototype

1 Introduction

The two most promising approaches towards the development of new generation database systems include object orientation and rule-based query languages [LRV88,Naqv89],[dMS88,ABDDMZ89]

Though these two formalisms have been regarded as in opposition, indeed they can be integrated within a single approach, which selects several compatible features and merges them into a unique setting

The feasibility of such an approach has been shown by many authors, mostly at a theoretical level [Beer88,Abit89,KiWu 89]

This paper presents an overview of LOGRES , a novel approach to the integration of object-orientation and rule-based programming paradigms for databases The project LOGRES is currently under development at the Politecnico di Milano as the follow-up of ALGRES [Ceri88,CCLLZ89], a main-memory based programming environment supporting an Extended Relational Algebra The main features of LOGRES are summarized as follows

- The LOGRES data model is based on the object-oriented approach, it supports classes of objects, with generalization hierarchies and object sharing These features take advantage of the introduction, at the instance level, of object-identifiers (oids) However, the LOGRES data model supports also conventional and extended relations, similar to those supported by ALGRES [Ull88,CGT89]
- The static structure of a LOGRES database is based on the use of *type equations*, which satisfy several structural properties, in particular, the consistency of legal database states is dictated by a collection of *integrity constraints*, which are automatically built from type equations Integrity constraints are expressed using the standard rule-based programming language
- The user language of LOGRES is rule-based, it extends *Datalog* to support sets, multisets, sequences, and controlled forms of negation Through the rule-based paradigm it is possible both to perform queries and updates [Abit88a] Further additions to *Datalog* enable oid unification, though oids are not visible to users
- The LOGRES system supports *modules*, a module is a collection of type equations and rules In particular, the *evolution* of a LOGRES database is ob-

This work is supported by the Esprit Project P2424, Stretch, and by the Project Logidata+ of the National Research Council of Italy

tained through sequences of applications of *update modules* to existing LOGRES database states

- When specifying the application of a module to a LOGRES database, the user specifies the intended *mode of application*, which dictates the side-effects of the module application on the LOGRES database, modes of application indicate as well the particular semantics that should be given to rules (e g inflationary vs non-inflationary), with this mechanism, LOGRES modules and databases are *parametric with respect to the semantics of the rules they support*

This collection of novel features makes LOGRES an ideal setting for testing various innovative ideas, ALGRES provides a fast prototyping environment In particular, the very liberal structure of the *closure* operation in ALGRES makes it possible to change the semantics of rules very easily We plan to prototype LOGRES upon ALGRES, though rather inefficiently, by introducing the notion of oids *above* ALGRES

This paper presents the various features of the LOGRES project the data model (Section 2), the rule-based language (Section 3) and modularization (Section 4) Each section is structured by including an overview and a comparison with recent approaches presented in the literature, which highlights the novel features of LOGRES, formal definitions of the data model and the rule-based language are in Appendix

2 Data Model

In this section, we give a description of the LOGRES data model The fundamental choice of the data model is to have both *relations* (called *associations* in LOGRES) and *classes* therefore, LOGRES integrates value-oriented models for complex objects, and pure object-oriented databases Our approach is similar to that of the language IQL of [Abit89] additional LOGRES features include multiset and sequence type constructors, the handling of multiple inheritance, the generation of active referential integrity constraints from type equations, the use of data functions A more completed comparison with IQL and other approaches is made in Section 2 2

2.1 Overview

A *LOGRES database* includes *classes* and *associations*, their schema is defined by *LOGRES type equations* Classes are sets of *objects*, each object has a unique *object identifier (oid)*, oid's are managed by the system, and not visible to users Associations are sets of *tuples*, usual NF^2 relations are examples of associations Objects can be shared, at the instance level, object sharing is provided by oid's Associations include classes, while classes cannot include associations, at the instance level, this is obtained by including oid's within tuples of associations Tuples are collections of attributes and oid's, legal values for attributes are defined by *domain equations* These equations may include classical built-in elementary types, such as *integer, string* Classes or associations may be extensions of *predicates* in the LOGRES query language, while domains may not, in essence, domains can be used as type constructors, but they are not first-class citizens of the LOGRES world

The distinction between classes and associations is needed, for a number of reasons in first place, classes introduce object sharing and inheritance in the LOGRES data model and associations preserve conventional and NF^2 relations into it In addition, not all data items deserve the status of an object, in particular, results of queries are typically associations Finally, since elements of a class are always distinguished by their oid's, we do not have the possibility of removing undesired duplicates, because *a class never contains duplicates* Thus, we need associations for those computations where elimination of duplicates is needed (e g fixpoint computations)

In a LOGRES database, each domain, association, or class is defined through its own *type equation* A type equation has the form $LHS = RHS$ where the LHS is the name of the type being defined and the RHS is a list of type constructors and other types, each of which is a basic type or appears as the LHS of one equation Names of types in the RHS must be unique, a *labelling mechanism* allows for including two or more occurrences of the same type in the RHS of one equation, each labeled by a different name Whenever needed, labels can be used to distinguish among two or more occurrences of the same type in the RHS of type equations

Type equations express the notion of *domain refinement* the domain of the type on the LHS is a *subset* of the domain of the type constructed on the RHS Thus, every type equation defines the *schema* and the *domain* of the LHS type, namely the structure of its elements and the values that are compatible with the schema, further, type equations also dictate the *active domain*, or *instance*, of the LHS type, namely the set of elements of that type present in a given state of the database The active domain is the range of the implicit quantifiers present into a rule This assumption is very useful when we have negated literals in the RHS of a rule we assume that variables which are only present in negated literals be restricted to their current active domain

If the class T_2 is referenced in the RHS of the type equation of the class T_1, then at the instance level we impose that the oid of the object of type T_2 must be set to that of an existing object of the class T_2 (which is ensured by the system through an *active referential integrity constraint*), or be the *nil* value, which is a legal value for oid's of any type In contrast, we do not accept *nil* oid's within associations each association instance must refer to *existing objects*

Types can be built by using the following type constructors

a TUPLE (or record) constructor (),
b SET constructor {},
c MULTISET (set with duplicates) constructor []
d SEQUENCE (ordered set) constructor <>

A tuple constructor allows introducing internal labels into type definitions, for instance if we want the internal record $(T2, T3)$ to be labeled N, we have $T = (T1, N\ (T2, T3))$

Example 2.1 The following type equations define the schema of a football database Note that Score is a complex domain built as a tuple of two integer values, each Player can have a set of roles, a Team includes a sequence of base-players and a set of substitutes Player and Team are modeled by classes, while Game is modeled by an association

Domains section
NAME = STRING
ROLE = INTEGER
DATE = STRING
SCORE = (INTEGER, INTEGER)

Classes section
PLAYER = (NAME, ROLES { ROLE })
TEAM = (TEAM-NAME,
BASE-PLAYERS < PLAYER >,
SUBSTITUTES { PLAYER })

Association section
GAME = (H-TEAM TEAM , G-TEAM TEAM, DATE, SCORE)

□

Set valued functions (cfr [AbGr88]) are included in LOGRES, as a shorthand notation for associations A function is declared as follows

$$F \qquad T_1 \longrightarrow T_2$$

where T_1 and T_2 are previously defined types, and T_2 is a set type, i e $T_2 = \{T_3\}$ for some type T_3 *Nullary* functions (functions with 0 arguments) can be used for *naming* the extension of a type

Example 2.2 As an example, we declare the function CHILDREN, based upon the association PARENT, as follows (we anticipate from the next section the rule defining the function's body)

Functions section
CHILDREN
PERSON ⟶ { (PERSON , BDATE) }
member(T,CHILDREN(X)) ←
PARENT (FATHER X,CHILD Y, BDATE Z),
T=(PERSON Y, BDATE Z)

An example of nullary function is the following

Functions section
JUNIOR ⟶ { PERSON }
member(X,junior) ← person(X, age A), A ≤ 18

□

Generalization hierarchies (isa relationships) are established between classes with the usual meaning A generalization is expressed in the type definitions by adding the statement T_1 *isa* T_2, which means that *each object of the class* T_1 *also belongs to the class* T_2, hence, the class T_1 is a subset of the class T_2 For instance, the following type equations provide a legal definition of a generalization hierarchy, in which, by virtue of the classic inheritance property, we may regard BDATE and ADDRESS as properties of STUDENT

PERSON = (NAME, BDATE, ADDRESS)
STUDENT = (PERSON, SCHOOL)
STUDENT isa PERSON

Labels may be needed to distinguish among several instances of the same type in the RHS of a type equation, for instance

EMPL = (emp PERSON, manager PERSON)
EMPL emp ISA PERSON

At the instance level, we model generalization hierarchies by inserting the oid's of sub-classes within the oid's of the super-class Thus, if John is a student, he has a unique oid which is used both within the PERSON and the STUDENT classes

Multiple inheritance is also allowed, namely, the possibility of declaring a class as a subset of two or more classes However, we only allow multiple inheritance among classes which share a common ancestor, as we do not postulate the existence of a universal class As a consequence of this constraint, the universe of all oid's is partitioned into disjoint subsets, such that each subset is the root of a (possibly empty) generalization hierarchy A renaming policy is provided in order to solve conflicts in multiple inheritance

Object sharing is used whenever the type equation of a class contains another class into its RHS An object may be contained into one or more objects, as illustrated by the following example

PROFESSOR = (PERSON, COURSE, SCHOOL)
SCHOOL = (NAME, ADDRESS, KIND, DEAN PROFESSOR)

Associations cannot contain other associations Thus, the RHS of type equations of associations may only include classes or domains

2.2 Comparison with related approaches

NF^2 models have introduced *structural complexity* within value-based models, In contrast, object-oriented data models have introduced the concepts of *identity*, *object sharing*, *inheritance* Both these ingredients have been selected for the LOGRES data model Several works treat these models and their declarative semantics formally, the LOGRES data model follows and extends the ideas of [KV84,Mai86,KiWu 89,Abit89]

Differences between the first three approaches and the LOGRES data model are evident LOGRES has strong typing and allows statical type-checking, as in the relational model, there is a clear separation of the notion of instance and schema Instead, in [Mai86,KiWu 89] there is no instance-schema separation, query languages in the above proposals can be consedered as untyped extensions of Prolog LOGRES is also different with respect to "flat" object-oriented data models such as [KV84], because it includes associations and complex type constructors

LOGRES shares the same basic assumptions as IQL [Abit89], but it differs in a number of features LOGRES uses data functions, which have been introduced with two main purposes performing nesting and unnesting operations (see example 3 2 below), and allowing a more direct application of *inheritance polymorphism* (cfr [Card88]) In contrast, IQL uses object identities for manipulating set types, and this leads, in our opinion, to less readable and declarative programs In LOGRES, union types are not present and inheritance is handled directly, in IQL inheritance is handled indirectly, through union types Finally, LOGRES extends the use of type constructors to multisets and sequences

3 Rule-Based Language

The rule part of LOGRES is a typed extension of Datalog, intended to be *complete* both for queries and updates Features of the rule-based language include a deterministic semantics, negation in the body and in the head of rules, data functions, complex (typed) variables and creation of objects through the invention of new oid's Two different semantics can be assigned to LOGRES programs, thus obtaining two different (inflationary and noninflationary) languages, in this paper, we describe only the inflationary semantics The expressive power of these languages is studied in [Abit88a] (where also the nondeterministic case is considered) and their properties are investigated in [AbSi89]

3.1 Overview

The structure of a LOGRES rule is the following

$$L \leftarrow L_1, \ , L_n$$

where $L, L_1, \ , L_n$ represent positive and negated literals, each literal may include variables of three kinds

a Ordinary, typed variables, they are labeled by their type name

b Oid variables, defined only in the context of a predicate corresponding to a class These variables are labeled by the language keyword *self* Values of these variables are not visible to the user

c Tuple variables, corresponding to collections of ordinary and oid variables In particular, tuple variables defined for a class include the oid of the class, though this part is not visible to the user These variables are *not* labeled

Literals may also include constants, of appropriate domains, constants are labeled by their type name **Unification** is legal between constants and variables of compatible types, two types are compatible if one is obtained as a *refinement* (see Appendix A) of the other one Suppose a tuple variable is used within a predicate and also the oid and other ordinary variables are named Then, when the oid and ordinary variables get bound, the same bindings are propagated to the corresponding arguments of the tuple variable Not all the arguments of a predicate need to be present in a rule, however, no literal without arguments is allowed, if it refers to a non-0 argument predicate We will discuss *safety* requirements below

Example 3.1 Consider the following set of type equations (where domain declarations are omitted)

Classes section
PERSON = (NAME, ADDRESS)
SCHOOL = (NAME, ADDRESS, KIND,
DEAN (PROFESSOR))
STUDENT = (PERSON, STUDSCHOOL
SCHOOL)
PROFESSOR = (PERSON, COURSE,
PROFSCHOOL SCHOOL)
STUDENT isa PERSON
PROFESSOR isa PERSON

Association section
ADVISES = (PROFESSOR, STUDENT)

The following are examples of legal LOGRES predicate occurrences

1 person(name Smith, address X)
2 person(self X)
3 person(X)
4 person(name X,Y,self Z)
5 school(director(self X))
6 advises(professor X)
7 professor(X)

The following variables and constants unify
1 Variable X in line 4 and constant Smith in line 1
2 Variable Y in line 4, variable X in line 3, variable X in line 6, and variable X in line 7,
3 Variable Z in line 4, variable X in line 2, and variable X in line 5 □

The above example has shown that all arguments of predicates can be referenced by appropriate use of labels and parenthesization

In an association T_1 which has a class T_2 into its type equation, we can reference the class T_2 either through an oid or by a tuple variable, thus, the following cases are equivalent

Association section
PAIR = (P-NAME NAME, S-NAME NAME)

$$pair(X, X) \leftarrow professor(X1, name \ X), \ student(Y1, name \ X),$$

$$
\begin{array}{lll}
 & & advises(X1, Y1) \\
pair(X, X) & \leftarrow & professor(self\ \ X1, name\ \ X), \\
 & & student(self\ \ Y1, name\ \ X), \\
 & & advises(X1, Y1)
\end{array}
$$

Notice that variables X1 and Y1 are tuple variables in the former case, oid variables in the latter

A particular interpretation of oid variables must be given with *generalization hierarchies* Let C1 and C2 be classes with compatible types and consider the rule $C1(Y) \leftarrow C2(X)$ Here two cases are possible

a Neither $C1$ *isa* $C2$ nor $C2$ *isa* $C1$ hold, then, the rule is considered legal, for each object of class $C2$, a new object of class $C1$ is created, with the same values as those of $C2$ and with a new oid

b Either $C1$ *isa* $C2$ or $C2$ *isa* $C1$ hold, then, since two objects in a generalization hierarchy must have the same oid, Y is unified with X

Consider now the rule $C1(X) \leftarrow C2(X)$ This rule is regarded as incorrect, unless either $C1$ *isa* $C2$ or $C2$ *isa* $C1$ Indeed, two objects cannot have the same oid if they do not belong to the same generalization hierarchy

Recursive rules are written in the usual way Recursion and data functions can be used to create nested relations, as in the following example

Example 3.2 We want to build an association between a person and the set of his descendants

Associations section
PARENT=(PAR PERSON, CHIL PERSON)
ANCESTOR=(ANC PERSON,DES {PERSON})

Functions section
DESC PERSON $\longrightarrow$ {PERSON}

$$
\begin{array}{lll}
member(X, desc(Y)) & \leftarrow & parent(par\ \ Y, chil\ \ X) \\
member(X, desc(Y)) & \leftarrow & parent(par\ \ Y, chil\ \ Z), \\
 & & member(X, T), \\
 & & T = desc(Z)
\end{array}
$$

Rules section

$$
\begin{array}{lll}
ancestor(anc\ \ X, des\ \ Y) & \leftarrow & parent(par\ \ X), \\
 & & Y = desc(X)
\end{array}
$$

□

The safety requirements in LOGRES are stated as follows

1) The oid variable of the LHS predicate may be unbound, this causes the generation of an *invented* oid

2) All other LHS predicate arguments must also be present on the RHS

As usual, unsafe rules can be detected at compilation time

LOGRES includes a comprehensive list of built-in predicates to handle complex terms, (like, for example, *Member*, *Union*, *Count*, etc see [CCCTZ89] for a complete list) Though built-in predicates do not add expressive power to the language and could actually be simulated by using ordinary functions and rules, they greatly improve the readability and conciseness of LOGRES programs

Built-in predicates are untyped every variable which occurs in a built-in predicate must also occur elsewhere in the rule, in an ordinary predicate to become typed However, types of variables or constants used as arguments of built-in predicates should be consistent (for instance, the union of two sets S_1 and S_2 is legal only if the two sets are compatible), type checking may be done at compilation time

Finally, LOGRES provides conventional built-in predicates for equality and arithmetical operations and comparisons

Example 3.3 The following programs computes the *powerset* of the elements of the relation R through the built-in predicates *Append* and *Union*

Associations section
R=(D)
POWER=(SET {D})

Rules section

$$
\begin{array}{lll}
Power(X) & \leftarrow & X = \{\} \\
Power(X) & \leftarrow & R(Y), Append(\{\}, Y, X) \\
Power(X) & \leftarrow & Power(Y), Power(Z), \\
 & & Union(X, Y, Z)
\end{array}
$$

□

As previously mentioned, it is possible to generate *invented* oid's through the use of rules with unbound variables on the LHS As observed in [Mai86], universal quantification over such variables does not make sense, since we want to create a single tuple each time the rule is fired, therefore, it was suggested that such variables should be existentially quantified Thus, in a LHS like $p(self\ \ Y, T1\ \ X1, \ , Tn\ \ Xn)$ where Y is unbound, the correct quantification should be $(\forall X1\ \ Xn)(\exists Y)$

Many authors ([Mai86,Beer88]) argue that the choice of the quantification in rules with invented oids cannot be chosen solely on the basis of the synctactic structure of the rule Consider the *interesting pair* (IP) example ([Mai86,KiWu 89])

$$
\begin{array}{lll}
ip(self\ \ X, emp\ \ E, & & \\
\quad mgr\ \ M) & \leftarrow & emp(E, name\ \ N, works\ \ D), \\
 & & dept(D, deptmgr\ \ M), \\
 & & emp(M, name\ \ N)
\end{array}
$$

The intended meaning of this rule is that a pair employee-manager is interesting if the name of an em-

ployee is the same as the name of the manager of his department It is argued in [KiWu89] that in this case the correct quantification is $(\forall E, \forall M, \exists X, \forall N)$ The quantification $(\forall E, \forall M, \forall N, \exists X)$ also happens to work here only because E and M functionally determine N Should this not be the case, the two quantifications would yield different results To solve quantification problems, like the one proposed above, LOGRES provides the use of associations , which enable explicit control over the elimination of duplicates Therefore, the above example can be written as follows

Example 3.4

Associations section:
PAIR = (EMPLOYEE, MANAGER)

Classes section:
IP = PAIR

Rules section:

$$\begin{aligned} pair(emp\ \ E, & \\ \quad mgr\ \ M) &\leftarrow emp(E, name\ \ N, works\ \ D), \\ & \quad dept(D, depmgr\ \ M), \\ & \quad emp(M, name\ \ N) \\ ip(X, C) &\leftarrow pair(C) \end{aligned}$$

□

Negative literals in the rule heads are interpreted as tuple deletions Such an extension is required for two reasons First , a construct allowing explicit deletions appears desirable in an update language Second, the language extends greatly its expressive power, achieving the same power as its procedural counterparts Example of negative literals in rule head are shown in section 4

Given a set of ground facts E (extensional database) and a program R over schema $\mathbf{S} = (\Sigma, isa)$, which also includes the rules generated as active referential integrity constraints (see 2 1), its deterministic inflationary semantics is a binary relation $\gamma(R)$ between sets of facts E and instances I of $\mathbf{S}$ Intuitively, the pair (E, I) belongs to $\gamma(R)$ if I is the *result* obtained by applying the semantics to the extensional *input* E The instances I in $\gamma(R)$ are obtained as the *inflationary fixpoint* $\gamma^{\omega}(R, E)$ This operator works on sets of facts, each application of the transformation yields one step of the sequence $F^0 = E, F^1, F^2, \quad , F^{\omega}(R, E) = \gamma^{\omega}(R, E)$ The inflationary deterministic semantics is used to give a uniform semantics to all LOGRES programs Formal details can be found in Appendix B Note that, due to the use of negation and data functions, stratification properties may be considered in order to obtain the model intended by the user If we use inflationary semantics within each *stratum* of a stratified program, this yields the perfect model semantics Whenever the program is not stratified with respect to negation or data functions, it can also be assigned a meaning, by computing it as a whole still under inflationary semantics Stratification in LOGRES plays the same role as *sequential composition* in [Abit89]

3.2 Comparison with related approaches

The main features of the LOGRES rule-based language are the deterministic bottom-up strategy of computation, strong typing, the use of inflationary semantics to introduce a limited form of control These aspects mark the difference from preceding proposal like LDL [Naqv89] or NAIL! [Morr86], similar features are present in RDL1 [dMS88], which does not support classes or complex objects, and in IQL [Abit89] Though the most innovative proposals of LOGRES regard modularization and updating facilities, a number of important extensions with respect to above languages can be listed

- The use of tuple variables to enrich the expressiveness of rules when dealing with objects,
- The introduction of active referential integrity constraints generated by type equations,
- The introduction of data functions to build nested terms and collections,
- A different interpretation of the EDB, which is not regarded as an instance of the database A *database state* is the triple (E, R, S) the set of tuples extensionally stored, the rules (which define more facts), and the schema of the database The database instance is the result of applying the rules R to E

4 Modules, Queries, and Updates

The LOGRES approach to updates preserves the declarative semantics of rules and puts all the control strategy into modules Modules partition the rules of the database into several parts, by selecting the option of application of a module, the effect on the database can be changed Goals can be specified within each module

In this section we first define the effect of the application of a module on a database instance, then we show how to use this method to accomplish several kinds of updates, finally, we compare our approach to others

4.1 Modules

To any consistent database state $(E_0, R_0, \mathbf{S}_0)$ corresponds an instance I_0 such that (E_0, I_0) belongs to the semantics $\gamma(R_0)$ A *module* in LOGRES is a triple $(R_M, \mathbf{S}_M, G_M)$, where R_M is a set of rules, $\mathbf{S}_M$ is a set of type equations, G_M is a goal A *module* can be applied to the database to obtain a new instance I_1 The effect of module application is twofold it may transform the underlying initial database state $(E_0, R_0, \mathbf{S}_0)$, yielding a new state $(E_1, R_1, \mathbf{S}_1)$, and provide an answer to the user's goal G_M The new database state generates an instance I_1 such that (E_1, I_1) belongs to $\gamma(R_1)$

A LOGRES module M defines a partial mapping $\mathcal{M}$ from database instances to database instances $\mathcal{M}$ is partial, as it is undefined over instances for which I_1

is inconsistent The application of a LOGRES module M to a database state $(E_0, R_0, \mathbf{S}_0)$ is *legal* if the initial state is consistent, and the instance $I_1 = \mathcal{M}(I_0)$ is defined The application of a module M to a database state is qualified by an *option* O, which indicates the side effects that the module application should produce on the database state The option O has six possible values

- *RIDI - Rule Invariant Data Invariant* Such a module corresponds to an ordinary query the type equations S_M are added to the initial type equations S_0, the rules R_M are added to the rules R_0, and the goal G_M is evaluated over $R_0 \cup R_M$, against the extensional database E_0 The database state does not change after this operation $E_1 = E_0,\ R_1 = R_0,\ S_1 = S_0$
- *RADI - Rule Addition Data Invariant* Such a module is used to *add* new rules to the persistent intensional database The result of the application of such a module to the state $(E_0, R_0, \mathbf{S}_0)$ is the following $E_1 = E_0,\ R_1 = R_0 \cup R_M,\ S_1 = S_0 \cup S_M$, if this new state is consistent Otherwise the update is rejected since the new instance $I_1 = \mathcal{M}(I_0)$ is undefined The module application (if consistent) may also provide an answer to the goal G_M, as in the RIDI case
- *RDDI - Rule Deletion Data Invariant* This option is used to *delete* rules from the persistent intensional database The result is similar to the RADI case $E_1 = E_0,\ R_1 = R_0 - R_M,\ S_1 = S_0 - S_M$, $I_1 = \mathcal{M}(I_0)$ is such that (E_1, I_1) belongs to $\gamma(R_1)$, if this new state is consistent
- *RIDV - Rule Invariant Data Variant* Such a module is only used to update the EDB The other components of the new database state are defined as follows $S_1 = S_0 \cup S_M(EDB),\ R_1 = R_0$, where $S_M(EDB)$ denotes the part of type equations in S_M describing new types added to the EDB Thus, the new rules are *not* added to the persistent IDB E_1 is computed by applying the update rules R_M to E_0 No goal answer is provided
- *RADV - Rule Addition Data Variant* Such a module is used to add new rules to the database, and to perform an update to the EDB As above, E_1 is computed by applying update rules R_M to E_0, and we further have $S_1 = S_0 \cup S_M,\ R_1 = R_0 \cup R_M$ No goal answer is provided
- *RDDV - Rule Deletion Data Variant* Such a module is used to delete rules from the database, and to perform an update to the EDB Here we have $E_1 = E_0 - E_M$, where E_M is the set of facts obtained as instance of $(\emptyset, R_M)$ We also have $S_1 = S_0 - S_M,\ R_1 = R_0 - R_M \quad I_1 = \mathcal{M}(I_0)$ is such that (E_1, I_1) belongs to $\gamma(R_1)$, if this new state is consistent No goal answer is provided

Note that, in the last three options, there is no goal answer, thus the goal G_M must not be specified

4.2 Updating a LOGRES database instance

As pointed out in Section 3 1, the *instance* I of a LOGRES database is not fully extensional A fact which belongs to I may either be present in the set of extensional facts E (which is the *persistent* EDB), or be inferable from rules of the program R (which is the *persistent* IDB) Both E and R can be changed by applying modules A predicate P (class or association) can be defined partly extensionally (because has tuple in E) and partly intensionally (by means of rules in R) A fact f which belongs to E is also present in I unless it is explicitly negated by some rule in R It is also possible that the fact f be *indipendently* inferable by some rule in R

We now describe how this mechanism can be used to accomplish the following operations materializing the database instance I, adding and deleting tuples, updating extensional relations, updating derived relations, stating integrity constraints, and stating active constraints

Materializing the instance The traditional deductive database approach, in which the EDB and the IDB do not overlap, is generalized by LOGRES it is sufficient to keep the sets of predicates defined in E and in R disjoint RIDI or RADI modules will be used to address queries to the database

LOGRES also generalizes the approach in which rules are used as triggers to maintain database consistency We can obtain the same situation in LOGRES by declaring all the rules in R as RIDV the effect is to have $E = I$ This can either be done as a general database strategy, or dynamically at a particular moment of the lifetime of the database Then, the extensional part E will coincide with the instance I

Adding and deleting tuples Insertion and deletion of tuples in E is straightforward A module with RIDV option will be used, addition of tuples requires rules with positive heads, deletion of tuples rules with negative heads The program R is left unchanged, and will automatically include in the instance I the new tuples in E_1 and also tuples generated as active referential integrity by rules in R, which act as triggers, as in the following example

Example 4.1 Suppose that $E_0 = \{Italian(Sara)\}$, $R_0 = \emptyset$

If we add a module with the RIDV option, and
$R_M = \{Italian(Luca) \leftarrow,$
$Roman(Ugo) \leftarrow,$
$Italian(X) \leftarrow Roman(X)\}$
The outcome is
$E_1 = I_1 = \{Italian(Sara),\ Italian(Luca),$
$Italian(Ugo),\ Roman(Ugo)\}$ □

Updating extensional relations The RIDV option is also used to update tuples in E without changing R, as in the following example

Example 4.2 Let S_0 include the association P = (*D1 integer, D2 integer*) and let E_0 = $\{p(1,1)\ p(2,2)\ p(3,3)\ p(4,4)\}$ We want to add 1 to the second field of all the tuples which have an even first field We use the following module, with RIDV option

Association section
MOD=(D1 integer, D2 integer)

$$
\begin{aligned}
p(D1\ \ X, D2\ \ Z) &\leftarrow p(D1\ \ X, D2\ \ Y),\\
&\quad even(X), Z = Y + 1,\\
&\quad \neg mod(D1\ \ X, D2\ \ Y)\\
mod(D1\ \ X, D2\ \ Z) &\leftarrow p(D1\ \ X, D2\ \ Y),\\
&\quad even(X), Z = Y + 1,\\
&\quad \neg mod(D1\ \ X, D2\ \ Y)\\
\neg p(Y) &\leftarrow p(Y, D1\ \ X), even(X),\\
&\quad \neg mod(Y)
\end{aligned}
$$

This program yields the result
$E_1 = \{p(1,1)\ p(2,3)\ p(3,3)\ p(4,5)\}$ □

Updating derived relations Various strategies for updating derived relations are feasible in the LOGRES framework

- The user defines rules to delete or modify the (extensional) premises of the rules of R defining the relation to be updated A module with RIDV is used in this case
- A new definition of the relation we want to update is written in a module RADV The module includes also rules with negative heads which inhibit the derivation of old tuples New definitions thus overrule the old ones
- As above but the old rule is deleted This is the cleanest way of updating an intensional relation, and can be accomplished as follows
  - the relation to be updated is materialized by making RIDV the rules which define it
  - old rules are (partly) deleted by making them RDDV facts deriving from them are deleted
  - new rules are added in a RADI module (or in RADV module to change E)

Constraints We call *active* constraints those which preserve the database consistency by adding or deleting tuples *Passive* constraints are those which must be verified in order to ensure database consistency We already saw that type equations contribute to generate integrity constraints Additional active constraints can be included in R by using RADV modules or can be used at any moment with a RIDV module Passive constraints are rules with an empty head (*denials*), like for example $\leftarrow \neg married(X), divorced(X)$ Their evaluation will result in an inconsistency if the database state satisfies the body Also passive constraints can be included permanently in R (RADI option) or used in the body of a transaction (RIDI option)

4.3 Comparison with related approaches

The most critical problem of deductive databases is that of merging update features in a declarative language Since *update* = *logic* + *control*, many authors agree on the need of introducing some form of control without moving to the all-procedural solution of object-oriented languages and databases (cfr [Abit88b])

Several ways of introducing control have been proposed Updates in the body of rules have been considered in the logic programming community In this case the control will correspond to the control strategy of resolution in a top-down evaluation of rules This idea is used in languages like LDL [Naqv89] and DLL [MaWa87], the latter being based on the formalism of dynamic logic Here, control is inherently nondeterministic The language allows the binding of values in updates to results of queries and vice versa, and the mixing of query part and update part in rule bodies can be used to express pre- and post-conditions These languages allows recursion in update procedures Termination is not guaranteed However, their expressive power is limited by the impossibility of changing the *active domain* of the database, due to the lack of features similar to the invention of values

The second way of introducing control is based upon a bottom-up evaluation of rules and will lead to updates in the head of rules Inflationary or stratified semantics establish some implicit control strategy, which can be made explicit by using a rule algebra, like that of [ImNa88] This class of languages can have deterministic or nondeterministic semantics and includes DL [Abit88a], and RDL1 [dMS88] DL is a theoretical proposal of an update language with invented values but without explicit deletions in the head In [Abit88a] DL is shown to be nondeterministic update complete, in the sense that it can express every mapping between instances of the database RDL1 is also nondeterministic, it also allows deletions in the head but invented values are not present, the language is therefore not complete in the sense described above

The LOGRES approach is characterized by the following points

- *Logic* is in rules and *control* in modules Their mode of application dictates the effect on the database state Inflationary semantics is used to give a uniform semantics which is as much declarative as possible
- Differently from other proposals, a predicate can be defined partly intensionally and partly extensionally In ordinary deductive databases the extensional relations are the *input*, and the rules are used

for defining views and for consistency checking In LOGRES, facts and rules are the two startpoints from which the current *database state* is derived This mechanism, though of lower level, is powerful and computationally simple when updating derived relations, checking database integrity, defining *active* constraints, etc , as illustrated by examples in Section 4 2

- LOGRES uses explicit deletions in the head, and invented values in a deterministic framework

5 Conclusions

A LOGRES prototype is in progress at Politecnico di Milano Its implementation uses ALGRES, an extended relational language which supports complex objects, extended relational operations and fixpoint operators Translation of the LOGRES data model into the relational one is described in [Ca90]

The LOGRES project integrates database management with rule-based and object-oriented concepts, taking advantage of recent work on the formalization of logic-based languages for supporting complex objects, object identity, and object sharing We believe that the main merit of LOGRES is to present a coherent proposal, providing a rich set of data modelling concepts, an expressive and powerful rule-based language, and the modularization mechanisms to provide encapsulation and hiding, in particular by controlling the effects of updates

Future work will be in the following directions·

- We will further refine the LOGRES language, in particular the features related to modularization and updates We will investigate consistency conditions on module applications, and translation of user-defined updates into module application We will also evaluate how effectively the notions of methods and of encapsulation, which are very popular in object-oriented systems, are supported within LOGRES
- We will continue the prototyping of LOGRES on top of ALGRES
- We will evaluate the expressiveness of LOGRES for building applications, by performing some case studies
- We will consider developing a complete programming environment for LOGRES, with tools supporting the design, debugging, and monitoring of LOGRES databases and programs, and we will investigate the methodological issues involved in designing applications using LOGRES

6 Acknowledgement

We like to thank Paolo Atzeni for very useful comments on the LOGRES data model and rule-based language, we also like to aknowledge the work of P Samarati, B Cattaneo, L Gasperini, P Tavazzani, G Torrente, I Zaffaina, who are currently implementing several features of LOGRES as part of their thesis

References

[ABDDMZ89] Atkinson, M , F Bancilhon, D De Witt, K Dittrich, D Maier, S Zdonik *The Object-Oriented Database System Manifesto* Proc First Int Conf on Deductive and Object-Oriented Databases, Kyoto, 1989

[AbGr88] Abiteboul, S , S Grumbach *COL a Logic-based Language for Complex Objects* Proc 1988 EDBT

[Abit88a] Abiteboul, S , and V Vianu *Datalog Extensions for Database Queries and Updates* INRIA Rep n 900, September 1988

[Abit88b] Abiteboul, S *Updates, a New Frontier* Proc 1988 ICDT

[Abit89] Abiteboul, S , P C Kanellakis *Object Identity as a Query Language Primitive* Proc 1989 SIGMOD

[AbSi89] Abiteboul, S , E Simon *Fundamental properties of deterministic and nondeterministic extensions of Datalog* Proc Journées Bases de Données Avancés, September 1989

[Beer88] Beeri, C *Data Models and Languages for Databases* Proc 1988 ICDT

[Ca90] Cacace, F *Implementing an Object-Oriented Data Model in Relational Algebra Choices and Complexity* PdM Report n 90-009

[Card88] Cardelli, L *A semantics of multiple inheritance* Information and Computation, 76 138-164, 1988

[CCCTZ89] F Cacace, S Ceri, S Crespi-Reghizzi, L Tanca, R Zicari *The LOGRES project Integrating Object-Oriented Data Modelling with a Rule-Based Programming Paradigm* PdM Report n 89-039

[Ceri88] Ceri, S et al *The ALGRES project* Proc 1988 EDBT

[CCLLZ89] Ceri, S , S Crespi-Reghizzi, G Lampertı, L Lavazza, R Zicari *ALGRES An advanced database system for complex applications*, to appear in IEEE-Software, 1990

[CGT89] Ceri, S , G Gottlob, L Tanca *Logic Programming and Databases* Springer Verlag (to appear in 1990)

[ImNa88] Imielinski, T , S Naqvi *Explicit control of Logic Programs through Rule Algebra* Proc 1988 PODS

[KiWu 89] Kifer, M , J Wu *A Logic for Object Oriented Programming (Maier's O-Logic Revisited)* Proc 1989 PODS

[KV84] Kuper, G M , M Y Vardi *A New Approach to Database Logic* Proc 1984 PODS

[LRV88] Lecluse, C , P Richard and F Velez *O2, an Object-Oriented Data Model* Proc 1988 SIGMOD

[dMS88] de Maindreville, C , E Simon *Modelling queries and updates in a deductive database* Proc 1988 VLDB

[Mai86] Maier D *A Logic for Objects* Proc Workshop on Foundations of Deductive Databases and Logic Programming, Washington USA, 1986

[MaWa87] Manchanda, S , D S Warren *A logic-based language for database updates* In Foundations of Logic Programming and Deductive Databases, ed J Minker (1987)

[Morr86] Morris, K , J D Ullman, A Van Gelder *Design overview of the NAIL! system* Proc ICLP, 1986

[Naqv89] Naqvi, S , S Tsur *A Logical Language for Data and Knowledge Bases* Computer Science Press, New York, 1989

[Ull88] Ullman, J D *Principles of Databases and Knowledge-Base Systems* Volume I, Computer Science Press, Potomac, MD, 1988

A Formal definition of the data model

We assume the existence of three disjoint sets of *names* the set $\mathcal{A}$ of *association names*, and the set $\mathcal{C}$ of *class names*, the set $\mathcal{D}$ of *domain names* The set $\mathcal{L}$ of *label, (or attribute) names* may share elements with the other sets of names We also assume the existence of the elementary types I (integer numbers) and S (finite strings) [2]

Definition 1. The set T of LOGRES *type descriptors* (or, briefly, *types*) is specified as follows

a I, S, D, C are types, where $C \in \mathcal{C}$, $D \in \mathcal{D}$

b $(L_1 \quad \tau_1, \quad , L_k \quad \tau_k)$ is a type (a *tuple type*), where $K \geq 0$, the L_i's are distinct labels from $\mathcal{L}$ and the τ_i's are types

c If τ is a type, then $\{\tau\}$ is a type (a *set type*)

d If τ is a type, then $[\tau]$ is a type (a *multiset type*)

e If τ is a type, then $< \tau >$ is a type (a *sequence type*)

Definition 2. A *schema* of LOGRES is a pair (Σ, isa), where Σ is a function

$$\Sigma \quad \mathcal{D} \cup \mathcal{C} \cup \mathcal{A} \longrightarrow T$$

associating, with each domain, class, or association name, its type descriptor isa is a partial order over $\mathcal{C}$, called *isa hierarchy* Furthermore, the following conditions hold

- For each domain D, the type descriptor $\Sigma(D)$ does not contain any class name
- If C_1 and C_2 are class names and C_1 *isa* C_2, then $C_1 \preceq C_2$ according to the following definition

A type τ_1 is a *refinement* of a type τ_2 (written $\tau_1 \preceq \tau_2$) iff one of these conditions holds

1) $\tau_1 \in \mathcal{D} \cup \mathcal{C} \cup \{I, S\}$ and $\tau_1 = \tau_2$
2) $\tau_1 \in \mathcal{D} \cup \mathcal{C}$ and $\Sigma(\tau_1) \preceq \tau_2$
3) $\tau_1, \tau_2 \in \mathcal{C}$ and $\Sigma(\tau_1) \preceq \Sigma(\tau_2)$
4) τ_1 has the form $(L_i \quad \tau_{1,i}), (1 \leq i \leq p)$ and τ_2 has the form $(L_k \quad \tau_{2,k}), (1 \leq i \leq q), q \leq p$ and $\forall k \exists! i \quad L_i = L_k, \tau_{1,i} \preceq \tau_{2,k}$
5) τ_1 has the form $\{\tau_1'\}$, τ_2 has the form $\{\tau_2'\}$ and $\tau_1' \preceq \tau_2'$
6) τ_1 has the form $[\tau_1']$, τ_2 has the form $[\tau_2']$ and $\tau_1' \preceq \tau_2'$
7) τ_1 has the form $< \tau_1' >$, τ_2 has the form $< \tau_2' >$ and $\tau_1' \preceq \tau_2'$

The notions that follow lead to the concept of LOGRES *instance*

We assume the existence of a countable set $\mathcal{O}$, whose elements are called *object identifiers* or *oid's*

Definition 3. An *oid assignment* is a function π which associates, with each C in $\mathcal{C}$, a finite subset of $\mathcal{O}$, i e , a finite set of oid's

Given an oid assignment π, each type τ is assigned a set $[\tau]_\pi$ in the following way

$[I]_\pi = I$ $[S]_\pi = S$ $[C]_\pi = \pi(C)$ $[D]_\pi = [\Sigma(D)]_\pi$

$$[(L_1 \cdot \tau_1, \quad , L_k \quad \tau_k)]_\pi = \{t \quad \{L_1, \quad , L_k\} \longrightarrow \bigcup_{i=1}^{k} [\tau_i]_\pi \mid t(L_i) \in [\tau_i]_\pi (1 \leq i \leq k)\}$$

$[\{\tau\}]_\pi = \{X \mid X \subseteq [\tau]_\pi$ and X is finite $\}$

$[[\tau]]_\pi = \{X \mid X$ is a finite set of couples composed by an element of $[\tau]_\pi$ and an occurrence integer number $n\}$

$[< \tau >]_\pi = \{W \mid W$ is a finite sequence of elements from $[\tau]_\pi\}$

Note that, by definition, $[I]_\pi$, $[S]_\pi$, and $[D]_\pi$ are always independent of π

Definition 4. An *Instance* I of a schema (Σ, isa) is a triple (ρ, π, ν) where

- π is an oid assignment for $\mathcal{C}$, such that
  a) If C *isa* C', then $\pi(C) \subseteq \pi(C')$

[2] Other elementary types (e g reals) may also be added if required

b) If $\pi(C) \cap \pi(C') \neq \emptyset$ then there exists $C'' \in \mathcal{C}$ such that C isa C'' and C' isa C''

- ν is a partial function

$\nu \quad o \mapsto \nu(o)$ such that $\Pi_{\Sigma(C)}\nu(o) \in [\Sigma(C)]_\pi$ for all C such that $o \in \pi(C)$ ν is called the *o-value assignment of o*

- ρ, called the *association assignment* for $\mathcal{A}$, is a (partial) function

$\rho \quad A \mapsto \rho(A) \subseteq [\Sigma(A)]_\pi$, such that $\rho(A)$ is finite

ν is such that if C' occurs in $\Sigma(C)$, where $C \in \mathcal{C}$, then $\forall o \in \pi(C), \Pi_{C'}\nu(o) \in [\Sigma(C')]_\pi$ If C' occurs in $\Sigma(A)$, where $A \in \mathcal{A}$, then $\forall t \in \rho(A), \Pi_{C'}t \in [\Sigma(C')]_\pi$ This the condition which ensures the referential constraint between C and C'

Note that the uniqueness of the value of the function ν ensures that to each oid corresponds a unique o-value in I

B Semantics of LOGRES programs

We now formally define the inflationary fixpoint operator of a program using valuations, satisfaction and the one step inflationary operator [Abit89] This operator works on sets of facts, each application of the transformation yields one step of the sequence $F^0 = E, F^1, F^2, \quad , F^\omega(R,E) = \gamma^\omega(R,E)$ To each F^i is associated a triple $(\rho_{F^i}, \pi_{F^i}, \nu_{F^i})$, The triple $(\rho_{F^\omega}, \pi_{F^\omega}, \nu_{F^\omega})$, corresponding to the fixpoint $F^\omega(R,E)$, if it exists, is an instance I of the schema of S The termination of this computation is not guaranteed, and it is not even decidable [AbSi89]

Note that due to the presence of *invented* oid's, the deterministic semantics of LOGRES is not exactly *functional* in the sense of [AbSi89], that is, the binary relation $\gamma(R)$ is not a mapping However, all the I corresponding to an E in $\gamma(P)$ are isomorphic to each other Therefore, LOGRES programs are *determinate*, i e they define partial functions up to renaming of oid's Thus, we can talk of *the* instance of (E,R,S)

We first define the triple $(\rho_{F^1}, \pi_{F^1}, \nu_{F^1}) = (\rho_E, \pi_E, \nu_E)$, as follows

$\forall P \in \mathcal{A}, \rho_E(P) = \{(\alpha_1, \quad , \alpha_n) \quad P(\alpha_1, \quad , \alpha_n) \in E\}$

$\forall P \in \mathcal{C}, \pi_E(P) = \{S \quad S \neq nil \wedge P(self\ S, \alpha_1, \quad , \alpha_n) \in E\}$

$\forall P \in \mathcal{C}, \forall S \in \pi_E(P) \quad \nu_E(S) = (S, \alpha_1, \quad , \alpha_n)$ if $P(S, \alpha_1, \quad , \alpha_n) \in E$[3]

Definition 5 Given a set of facts F, with an associated triple (π_F, ρ_F, ν_F), a *valuation* θ of F is a partial function from constants, variables and terms to o-values, such that

- If k is a constant of type τ, and $k \in [\tau]_{\pi_F}$, then $\theta(k) = k$
- If X is a variable of type τ, $\theta(X)$ is in $[\tau]_{\pi_F}$, and the constants of $\theta(X)$ are present in F
- $\forall C \in \mathcal{C}, \quad \theta(C) = \{S \mid S \in \pi_F(C)\}$
- $\forall A \in \mathcal{A}, \quad \theta(A) = \{v \mid v \in \rho_F(A)\}$
- $\theta(t_1, \quad , t_k) = (\theta(t_1), \quad , \theta(t_k))$

Definition 6 Given a set of facts F, with an associated triple (π_F, ρ_F, ν_F), and a valuation θ of F, defined on terms t_1 and t_2, and given literals $t_1(t_2)$ and $t_1 = t_2$, we have

1) $F \models \theta(t_1(t_2))$ if $\theta(t_2) \in \theta(t_1)$
2) $F \models \theta(t_1 = t_2)$ if $\theta(t_2) = \theta(t_1)$
3) $F \models \theta(\neg t_1(t_2))$ if $\theta(t_2) \notin \theta(t_1)$
4) $F \models \theta(\neg(t_1 = t_2))$ if $\theta(t_2) \neq \theta(t_1)$

We say that F *satisfies* the considered literal, with respect to the valuation θ If p is a built-in predicate, $F \models \theta(p(t_1))$ if $p(\theta(t_1))$ is true according to the ordinary sense of the built-in predicate p For instance, suppose p is the predicate *union* and t_1 is the tuple (X,Y,Z) Then, $F \models \theta(union(X,Y,Z))$ iff $\theta(Z) = \theta(X) \cup \theta(Y)$

Given a rule r and a valuation θ, we can consider the application of θ to the literals of r Then, we say that $F \models \theta(body(r))$ iff F satisfies all the literals in the body of r, with respect to θ

Definition 7 Given a set of facts F and a set of rules R, we define the *valuation domain* $VD(R,F)$ as

$VD(R,F) = \{b(r) \mid r \in R, \exists\theta$ valuation on F such that ($b(r) = \theta(body(r))$,

$F \models b(r)$, and there is no θ' such that θ' is an extension of θ and $F \models \theta'(head(r)))$ $\vee$ (r is a fact and $b(r) = \emptyset$)$\}$

The significance of the valuation domain is that if one thinks of F as the "current state", then each (r,θ) contributes to augmenting F Thus, the valuation-domain is the set of "valuated bodies" that participate in the derivation of new facts The ground facts can be added to F either using old oid's and constants, or inventing oid's

Definition 8 Given a rule r, and a valuation domain $VD(R,F)$, a *valuation map* $\eta(b(r))$ associates, with each of the elements $b(r)$ of $VD(R,F)$, a valuation of the variables and constants in the head of r in the following way

$\forall k \in$ head(r), $\eta(b(r))k = k$

$\forall X \in$ head(r)

a) if X is in the body of r, then $\eta(b(r))X = \theta(X)$

b) if X occurs as the Self of a class type C, and C is the predicate in the head of r, and X does not appear in the body of r then $\eta(b(r))X$ is in $\mathcal{O} - \pi_F(C)$ In this case, $\eta(b(r))X$ is unique for each valuation domain

c) if X is of a class type C, and C is not the predicate in the head of r, and X appears in the head but not in the body of r, then $\eta(b(r))X = nil$

Finally, if X_1 and X_2 appear in the head but not in the body of r_1 and r_2 respectively, with $\eta(\theta_1(body(r_1)))X_1 = \eta(\theta_2(body(r_2)))X_2 \neq$ nil, then it must be that $r_1 = r_2$, $X_1 = X_2$ and $\theta_1 = \theta_2$ This condition ensures that the same valuation map be not

[3] Recall that LOGRES rules may also contain literals where not all the arguments are quoted However, in giving the valuation, we suppose that these rules be rewritten so that all the arguments are present

associated to two different variables, or to the same variable with two different substitutions

Point b) performs the generation of new oid's Note that the unicity condition imposed in point b) ensures the determinism of the computation, since each computation step can only generate one oid for each variable Note also that, by definition of valuation domain, once a rule r has been fired for a certain substitution, and an oid has been generated, that rule cannot generate any more oid's for the same substitution

We now define the two sets of *positive facts* and *negative facts* produced by a deduction step For any pair (R, E) we have

$\Delta^+(R,E) =$
$\{P(\eta(b(r))X_1, \quad, \eta(b(r))X_n)$ such that $\exists r \in R$ with a positive head
$P(X_1, \quad, X_n)$ with $b(r) \in VD(E,R)$ and $b(r)$ is a ground instance of the body of $r\}$

$\Delta^-(R,E) =$
$\{P(\eta(b(r))X_1, \quad, \eta(b(r))X_n)$, such that $\exists r \in R$ with a negative head
$\neg P(X_1, \quad, X_n)$ with $b(r) \in VD(E,R)$ and $b(r)$ is a ground instance of the body of $r\}$

The one-step inflationary operator is defined by means of Δ^+ and Δ^- Note that Δ^+ and Δ^- *are not necessarily disjoint* We define the one-step inflationary operator for the step $i+1$

ONE-STEP INFLATIONARY OPERATOR – step i+1

Given $F^i = (\rho_{F^i}, \pi_{F^i}, \nu_{F^i})$, we define $\gamma_{inf}(R) = (F^i, F^{i+1})$ with $F^{i+1} = (\rho_{F^{i+1}}, \pi_{F^{i+1}}, \nu_{F^{i+1}})$ as follows

Let $VAR^i = ((F^i \oplus \Delta^+(R,F^i)) - \Delta^-(R,F^i)) \oplus (F^i \cap \Delta^+(R,F^i) \cap \Delta^-(R,F^i))$

Then

$\forall P \in \mathcal{A}$, $\rho_{F^{i+1}}(P) = \{(X_1, \quad, X_n)$ such that $P(X_1, \quad, X_n) \in VAR^i\}$

$\forall P \in \mathcal{C}$, $\pi_{F^{i+1}}(P) = \{S$ such that $P(self\ S, X_1, \quad, X_n) \in VAR^i\}$

$\forall P \in \mathcal{C}$, $\forall S \in \pi_{F^{i+1}}(P)$ if $P(self\ S, X_1, \quad, X_n) \in VAR^i$ then $\nu_{F^{i+1}}(S) = (S, X_1, \quad, X_n)$

Where, given two sets of ground facts G and G', the *composition* $G \oplus G'$ is defined as follows

$\forall P \in \mathcal{A}, \rho_{G_1 \oplus G_2}(P) = \rho_{G_1}(P) \cup \rho_{G_2}(P)$

$\forall P \in \mathcal{C}, \pi_{G_1 \oplus G_2}(P) = \pi_{G_1}(P) \cup \pi_{G_2}(P)$

$\forall P \in \mathcal{C}$, and $\forall S \in \pi_{G_1 \oplus G_2}(P)$

- if $S \in \pi_{G_2}(P)$, then $\nu_{G_1 \oplus G_2}(S) = \nu_{G_2}(S)$
- if $S \in \pi_{G_1}(P)$ and $\neg S \in \pi_{G_2}(P)$, then $\nu_{G_1 \oplus G_2}(S) = \nu_{G_1}(S)$

In practice, we eliminate from the union of G and G' the set of ground facts of G that have the same oid but different o-values from some fact of G' Note that $\oplus$ is *noncommutative*

The resulting instances $I = (\rho_I, \pi_I, \nu_I)$ such that (E, I) belongs to the semantics $\gamma(R)$ for the program R with extensional instance E are obtained as follows

$F^0 = E$,

F^{i+1} is such that $(F^i, F^{i+1}) \in \gamma_{inf}(R)$

$I \in \gamma^\omega(R,E) = F^\omega = F^k$ such that $F^k = F^{k+1}$ if such a k exists

$\rho_I = \rho_F^\omega$, $\pi_I = \pi_F^\omega$, $\nu_I = \nu_F^\omega$

The deterministic semantics of a program is undefined if there is no fixpoint of the sequence $F^0, F^1,$, or if all the fixpoints generated are inconsistent