The LAURE Model for Object-Oriented Logic Databases

Yves Caseau
Bellcore, 445 South Street, Morristown NJ 07960-1910, U.S.A.

Abstract: We present the model that we used to implement the object-oriented deductive system LAURE. This data model supports advanced object-oriented programming including a complete type system, polymorphism, multiple inheritance and parameterized object invention. We present a logical language associated with this model and an equivalent relational algebra. We propose some techniques based on this algebra which permit tractable implementation of deductive rules and constraints in this object-oriented database.

1. INTRODUCTION

Many attempts have been made to integrate logic programming into a object-oriented system, both from the practical and theoretical side [Ku85][Ait86][Ga86][Mai86]. Practical systems, such as object-oriented expert system shells, have limited the logic paradigm to a smaller extent, and use implementation techniques which are less powerful than what is achieved in relational systems like LDL[NT89]. On the other hand, since merging classical logic programming with object-oriented databases leads to "impedance" mismatch, a large number of attempts have been made to propose new theoretical foundations to integrate both paradigms. Most of these models [AK89][KW89] subsume both relational and current object-oriented data models [A&A89] but do not provide implementation strategies that could compete with advanced PROLOG systems.

Here, we propose a model that is restricted on certain aspects to achieve tractability, though its general expressiveness includes object-oriented programming and powerful logic programming. This model has been used to successfully implement a deductive object-oriented system, LAURE [Ca88].

LAURE is a C-based main-memory deductive system, which has been used for expert systems and simulation projects. Its expressive power is higher than usual object-oriented systems, since it captures parametrization and a certain form of disjunctive information (non-determinism). It also incorporates a declarative logic language from which deductive rules and constraints can be formed.

Section 2 recalls the motivations that have lead to this data model. The need for logic programming inside an object-oriented system is analyzed and some goals for logic resolution are set. Section 3 presents a model that achieves these goals. A data model and its associate logic language are given. A powerful type system derived from this data model is presented, which supports multiple inheritance and polymorphism. Section 4 deals with a relational algebra that is equivalent with the logic language. Its application to logic programming is shown, as well as some useful properties. Section 5 shows how these properties can be applied to achieve efficient resolution of rules and constraints in our model.

2. MOTIVATIONS

2.1 An Extended Object-Oriented Model

This model should be extensible and support a full programming language. The LAURE data model was first developed to accommodate a programming language developed for AI applications. The main feature of such a language is the extensibility, that is the ease of adding a new feature to the language. Our approach to extensibility is to develop a reflective model, which can be used to describe itself [Ca88]. Extensions of the model are, therefore, easy to implement since the main components of the model are described within the language. The general extensibility principle is to apply the advantage of object-oriented programming (reusability, inheritance, sharing) to the object model itself.
This model should support **multiple inheritance** and **polymorphism**. Multiple inheritance is necessary for AI applications, but should be given a proper semantics (as declarative as possible) for solving conflicts. By polymorphism, we mean that the same operations can be understood differently (both at run-time and compile-time) according to the type of its arguments, either for its interpretation and its typing. For instance, the + operation can be applied to integers, strings and floats (syntactic polymorphism). We also want the compiler to know that the method `car` applied to a list-of-X returns an object or type X (or that the function `top` applied to a stack-of-Y returns an object of type Y), so `car` is a polymorphic function (applies to many types) with a polymorphic type description.

A weakness that is often found when comparing a relational database [Co70] with an object-oriented database is the lack of n-ary extensional relations other than binary. This is, on one hand, the strength of object-orientation since the storage of binary relations by their classes is easy to implement efficiently (the access to data is always "set-at-a-time"), but it is also a weakness, since we must create an object for each tuple, which is not always natural. This process also transforms a well behaved Horn-clause (1) like:

\[
ALINE(x, y, z), ALINE(y, z, t) \Rightarrow ALINE(x, y, t)
\]

into a logic rule (2) with explicit object invention [HS89] like:

\[
\exists o_1, o_2 \in \text{ALIGNMENT}; \quad x(o_1) = y(o_2) \land y(o_2) = z(o_1) \Rightarrow \\
\exists o_3 \in \text{ALIGNMENT}; \\
x(o_3) = x(o_1) \land y(o_3) = y(o_1) \land z(o_3) = z(o_1)
\]

We need a way to represent this knowledge in our model in such a way that practical top-down resolution is feasible.

### 2.2 Object-Oriented Resolution

Automated deduction is necessary in an object-oriented database. Performing deduction and value propagation with imperative programming, although possible, leads to programs that are difficult to debug and to maintain. Our experience is that declarative programming, which is not as general, is preferable for such operations [Ca89]. Efficiency is critical, since the applications are usually large and since the imperative language is available as an alternative.

The extensional part of the object database varies and its variations must be supported by the logic resolution system. Most of the updates are positive: some information is added to the database, but some kinds of information may also be removed. Since propagation of negative updates is difficult, top-down resolution must be provided. On the other hand, a lot of information is monotonic and is entered once for all. Therefore, bottom-up resolution should also be present. A combination of both strategies, such as the yoyo method [SKGB87] is actually another interesting alternative.

We need a robust system, where completeness and a low sensitivity to input order are necessary. Our experience with object-oriented databases is that the tasks to be performed by rules are simple but numerous. Full recursion (not only linear recursion) must be supported, since most of the common rules, such as path analysis, are recursive with cyclic data. The intensional description must be taken into account in the logic system, since it plays an important part in an object-oriented system (for instance, all the arithmetic knowledge is described as a set of external functions).

Since we need to find a top-down resolution strategy, we cannot allow any kind of object invention, as in [AK89] with an inflationary semantics. To use skolemization on the rule (2) we need a construct analogous to functional terms in PROLOG. For instance, the clause (1) can also be written:

\[
HOLD(\text{line}(x, y), z), HOLD(\text{line}(y, z), t) \Rightarrow HOLD(\text{line}(x, y), t)
\]

If we consider that the association `(x,y) + Zine(x,y)` is a constant bijection, we may represent it more easily in the object world, and the resulting rule can be evaluated in a top-down manner.

### 2.3 Non-Determinism and Constraints

Besides the purely deductive (and deterministic) tasks, logic programming should also be used in an object-oriented database for problem solving. Problem solving can be more or less achieved in PROLOG through the use of auxiliary predicates, but a separate mechanism is needed in the case of an object-oriented database [Ca90]. Constraints (which could be called choice rules) need to be introduced, which lead to non-deterministic resolution. A constraint is a logic expression which is necessary instead of...
being sufficient; whereas a rule tells that a given value may be deduced if some condition is satisfied, a constraint tells that some condition must be verified before a value may be chosen.

Non-deterministic rules are useful only if incomplete information may be represented. Our model must take into account disjunctive information, also seen as incomplete information [AKG87] [IVS91]. A constraint holds on some objects’ attributes (mono-valued), where a value is to be found by the resolution system. Problem solving really consists in completing some disjunctive description of a set of objects, by finding one unique value for each mono-valued attribute, so that each constraint is verified. Each attribute of each object is given a set of possible values, either directly such as in:

\[ \text{size(John)} \in \{162, 177, 190\}; \text{color(my\_ball)} \in \{\text{blue, green}\} \]

or for a set of given objects such as in the following declaration:

\[ \forall x \in \text{Toy}, \text{color}(x) \in \{\text{yellow, red, purple, orange, brown}\} \]

Non-deterministic rules such as constraints imply many ways of resolution. A set of constraints defines a set of admissible values that satisfy each constraint. This set is a subset of the set of possible solutions and may be empty. The most usual type of resolution is to find one value of this set. It is also needed to sometimes find all the solutions. However, we have found that many other kinds of resolution were necessary, including hypothetical reasoning, where resolution is conditioned by some additional information, and reactive resolution, which consists in finding a solution when some external conditions change.

3. THE LAURE DATA MODEL

3.1 The Object Data Model

The data set is made of a finite set of objects \( O \) and two distinguished values \( \bot \) (unknown) and \( \top \) (error). Using a finite set with the "error" extension is another solution to model infinite sets such as integers. We see integers in our model as a finite subset with a possible overflow for arithmetic operations, which is represented in the following ways:

\[ \text{MAXINT} + 1 = \top \]

where \( \text{MAXINT} \) is the largest known integer in the system.

\( O \) holds a hierarchical decomposition into many subsets called classes. Classes are reified [KL89], which means that the set of classes \( C \) is included in \( O \). For each member \( c \) of \( C \), the set of objects that belongs to set represented by \( c \) is written \( \chi(c) \). Each object is described with its properties from \( P \), which are binary relations among objects. The set of properties is divided into two subsets \( P^{*} \) and \( P^{*} \), which respectively contains multi-valued and mono-valued relations. As a part of reification, the hierarchy among classes is represented by a distinguished property subset \( \subset E P' \).

Non-binary relations among objects are represented by object functions from a set \( F \). Object functions are classified according to their mathematical properties. \( F \) contains three distinguished subsets: \( F_{o} \) (for object operation), \( F_{c} \) (for comparison) and \( F_{p} \) (for parametrization).

A triple made of such sets \( M = (O,P,F) \) is called an object model. An object system is an instance of such model. Precisely an object system \( (M,I) \) is made of a model and a function \( I \) such that:

\[ \forall p \in P^{*}, I(p) \in (O \rightarrow (O \cup \{\bot, \top\})) \]

\[ \forall p \in P', I(p) \in (O \rightarrow \text{Powerset}(O)) \]

\[ \forall f \in F_{o}, I(f) \text{ is a commutative and associative operation from } O \times O \text{ to } O \cup \{\bot, \top\} \]

\[ \forall \theta \in F_{c}, I(\theta) \in ((O \times O) \rightarrow \{\text{true, false}\}) \]

\[ \forall f \in F_{p}, I(f) \in ((O \times O) \rightarrow (O \cup \{\bot, \top\})) \text{ and there exists three classes } c_{1}, c_{2}, c_{3} \text{ such that } I(f) \text{ is a bijection from } c_{1} \times c_{2} \text{ to } c_{3}. \]

\[ \text{I(subset)} \text{ is a partial order relation which induces a lattice structure.} \]

\[ \forall c_{1}, c_{2} \in C, c_{1} \subset \text{I(subset)}(c_{2}) \implies \chi(c_{1}) \subset \chi(c_{2}) \]

Example:

Let \( M = (O = [0, \text{Maxint}], P = \text{succ}, F = \{+, <, =\}) \) be an object model. The interpretation function \( I \) may be:

\[ I(\text{succ}) = \lambda(x). x+1 \]

\[ I(+) = \lambda(x,y). x+y \text{ (integer addition).} \]

\[ ^{2} \text{A multi-valued relation is any binary relation. A mono-valued relation } r \text{ is a functional relation, where there is at most one object } y \text{ for each object } x \text{ such that } r(x,y). \]

\[ ^{3} \text{In this paper, we assume for the sake of simplicity that } F_{p} = F_{p}^{1} \cup F_{p}^{2} \text{ (unary and binary functions).} \]
The properties of addition put + in $F$, where < and divisibility are two order relations placed in $F$.

The restrictions on object functions will permit an efficient integration in the top-down logic resolution [Ca90]. We need these mathematical properties to use equations made from these functions (section 4.2). The restriction on $I(\text{subset})$ is a key point for a clean resolution of multiple inheritance conflicts or for the completeness of the type system (section 4.3). The good news is that there exists an algorithm to extend any class system of an object model into an equivalent class system (only larger), that can be represented by a lattice[Ca88].

### 3.2 Logic Relations

The variable extensional knowledge is represented by a set of binary relation variables. Each property of $P$ is a constant in the logic world. Properties are used to represent information that does not vary, such as the name of a person or the weight of a physical object. On the other hand, object relations are used to represent knowledge that will vary according to logic rules and user updates. The separation between properties and relations is also found in [AK89], but here we restrict relations to arity 2, and we make a distinction between multi-valued relations and mono-valued relations.

A database scheme is a pair $(S, R)$, where $S$ is an object system and $R$ a finite set of variables $\{R_1, \ldots, R_n\}$. The set of relation variables $R$ is partitioned into $R^*$ and $R^+$, as previously.

Each variable from $R^*$ represents a binary relation, organized into sets. The class of an object $x$ according to a relation $r$ is the set $\{y \mid r(x,y)\}$. This is the way a binary relation is usually implemented in an object-oriented system. Throughout this paper, we write $r(x)$ the class of $x$ according to the multi-valued relation $r$. A variable from $R^+$ represents a mono-valued relation. We shall write $r(x)$ the unique object bound to $x$ by the mono-valued relation $r$, and write $r(x) = \perp$ when no such object exists. If the value of the relation for an object is not known, we want to represent a set of possible values instead. Therefore, we define a database instance of a database model as an assignment from $R$ to $\{(O \rightarrow \text{Powerset}(O))\}$.

**Definition**: A database instance is a function of $D = (R \rightarrow \{(O \rightarrow \text{Powerset}(O))\})$.

For each $x \in O$, if $R_i \in R^*$, $d(R_i)(x)$ represents the class of $x$ according to the relation denoted by $R_i$. If $R_i \in R^+$, $d(R_i)(x)$ represents the set of possible values for $R_i(x)$. If this set has one unique member $y$, then it means that $R_i(x) = y$.

There is a lattice structure on databases instances, derived from the following order:

- $d_1, d_2 \in D$, $d_1 < d_2 \iff$
  - $\forall R_i \in R^*, \forall x \in O, d_1(R_i)(x) \subset d_2(R_i)(x)$
  - $\forall R_i \in R^+, \forall x \in O, d_2(R_i)(x) \subset d_1(R_i)(x)$

The greatest lower bound operation is:

- $\forall R_i \in R^*, \forall x \in O, d_1(R_i)(x) \cap d_2(R_i)(x)$
- $\forall R_i \in R^+, \forall x \in O, d_1(R_i)(x) \cup d_2(R_i)(x)$

The intuitive meaning of this order is that if $d_2 > d_1$, then $d_2$ contains the knowledge in $d_1$ plus some additional information. A database instance $d$ usually contains some incomplete information through the value of relations from $R^*$. A complete database instance $d$ is such that $|d(R_i)(x)| \leq 1$ for each $R_i \in R^*$. The completed database instance $c(d)$ is defined by:

- $\forall R_i \in R^*, c(d)(R_i) = d(R_i)$
- $\forall R_i \in R^+, \forall x \in O, i d(R_i)(x) = 1$
- $\text{then } c(d)(R_i)(x) = d(R_i)(x)$ else $c(d)(R_i)(x) = \emptyset$

$c(d)$ is the smallest complete database instance that contains $d$ and represents the information that is known, with no disjunctive forms. The set of complete values is a semi-lattice, since the intersection of two complete values is a complete value, but the union is not.

### 3.3 The Laure Logic Language

The Laure Logic Language ($L_3$) is an extension of binary Datalog to object functions. Binary Datalog is the restriction of Datalog to binary predicates, which is adequate for the object model. We extend it by allowing the use of object functions from $F_o \cup F_c \cup F_p$. An assertion with free variables in the set $V$ is described by the
following grammar\textsuperscript{5}.
\[
<\text{assertion}(V)> :: <R(P)> <\text{expression}(V)> <\text{expression}(V)>
\]
\[
\mid [<\text{expression}(V)> <\text{expression}(V)> ]
\]
\[
\mid [<\text{expression}(V)> \land <\text{expression}(V)> ]
\]
\[
\mid [<\text{expression}(V)> \lor <\text{expression}(V)> ]
\]
\[
\mid [\exists <x \in V> <\text{expression}(V) \cup \{z\}> ]
\]
\[
<\text{expression}(V)>> :: V 1 \mid <R(P)> <\text{expression}(V)>
\]
\[
\mid [<\text{expression}(V)> <\text{expression}(V)> ]
\]
\[
\mid [c<\text{expression}(V)> cF<\text{expression}(V)> ]
\]
\[
\mid [3 -2 65 V> <\text{expression}(V)> ]
\]

Given the notion of an assignment function \( v \) of \((V \rightarrow O)\), the semantics of this language is straightforward to define. Given a database instance \( d \) and an assignment function \( v \), if \( E \) is an expression \( (V) \), \( [E]_{v} \) is an object of \( O \); if \( A \) is an assertion \((V) \), \( [A]_{v} \) is a boolean value.

Queries may be used to define values, sets or relations inside the object model. A query is an assertion from the LAURE logic language \((L)\). The query \( A(V) \) defines a boolean if \( V = \emptyset \). It defines a set if \( |V| = 1 \) (the set of objects \( x \) such that assigning the unique variable \( x \) leads to the true value). If \( |V| = 2 \), the query defines a binary relation.

Logic rules are defined as usual from the query language:

Definition: A rule is a pair \((A(\{y\}) \Rightarrow R_{1}(x, y))\), where \( A \) is an assertion on \( \{y\} \) and \( R_{1} \) a relation from \( R_{1} \).

The intuitive semantics is that such a rule is satisfied in the database instance \( d \) if the binary relation represented by the query is included in the relation represented by \( R_{1} \). For instance, here is a relational constraint:

\[
\text{pressure}(x, y) \Rightarrow [y \ast \text{volume}(x)] = [n(x) \ast [R \ast \text{temperature}(x)]
\]

Relational constraints are derived (see section 4.2) from objects constraints such as:

\[
([\text{pressure}(x) \ast \text{volume}(x)] = [n(x) \ast [R \ast \text{temperature}(x)]
\]

### 3.4 Types and Inheritance

This model has been used to build an imperative object-oriented language. The purpose of this language is to build an object system, that can be used later with the deductive system that we are going to describe in the next sections. The imperative language permits us to build a class system, to build binary properties from \( P \), and to define object functions. For instance, it is this language that ensures the lattice property of the subset relationship. Describing objects (defining properties and functions) is based on the notion of a restriction:

Definition\textsuperscript{6}: a restriction of a property \( p \) is a partial relation on \( O \times \ldots \times O \) and a signature. The signature is a cartesian product \( C_{1} \times \ldots \times C_{n} \) which defines the range of validity of the relation \( p \) can be computed on \( C_{1} \times \ldots \times C_{n} \) with the associated relation.

The relations may be defined in many ways (including lambda-abstractions with the object programming language, called methods). Because of the set semantics, inheritance is implicit (a restriction on \( C_{1} \times \ldots \times C_{n} \) can be applied to any member of any subset of \( C_{1} \)). Inheritance resolution is based on the fact that the set of classes is closed under intersection \([Ca88]\). A conflicts occurs when two restrictions of the same property have signatures with a non-empty intersection. The closure property allows one to solve these conflicts systematically in a declarative manner.

Typing is based on the class lattice but incorporates powerful extensions. A type is a more complex form of object set and subtyping is based on set inclusion. The main goal of the type

\textsuperscript{5} This syntax is made similar to the LAURE programming language to facilitate integration.

\textsuperscript{6} Objects functions are also described through the restriction paradigm. The whole signature is taken into account to find the correct restriction to apply, not simply the first argument, as in simple languages like SMALLTALK.
The type system is to improve compilation by predicting the "type" of the result of some computation. Two properties are necessary: subtyping, to find if a restriction applies in a given case, and intersection, to eliminate a wrong restriction or to get a better prediction. Types are reified, which means that the finite subset \( \text{TYPE} \) of types used in the system is included in \( \mathbb{O} \). Here is the type system used in LAURE:

\[
\text{<type> :: <enumeration>l <class>
| <type> u <type>
| <type> n <type>
| \{ <type> - <enumeration> \}
| \{ <property>:<type> \}
| <type> set-of
\]

Each of these type expressions represent a set of objects in a given object system \( S \). There is an associated subtyping based on inclusion which is suitable for compilation. The type semantics is given by the set interpretation \( \chi(t) \) which takes an object system \( S \) and returns a set of objects:

\[
\chi(\{o_1, o_2, \ldots, o_n\}) = \{o_1, o_2, \ldots, o_n\}
\]
\[
\chi(\{e_1 \cup e_2\}) = \chi(e_1) \cup \chi(e_2)
\]
\[
\chi(e_1 \cap e_2) = \chi(e_1) \cap \chi(e_2)
\]
\[
\chi(e_1 - \{o_1, o_2, \ldots, o_n\}) = \chi(e_2) - \{o_1, o_2, \ldots, o_n\}
\]
\[
\chi([p:e]) = \{e \in \mathcal{E}, \forall y \in \mathcal{U}, (l(p)(y) = y) \Rightarrow y \in \chi(e)\}
\]
\[
\chi([x:set-of]) = \{e \in \mathcal{E}, \forall y \in x, y \in \chi(e)\}
\]

This type system is an intersection lattice, where subsumption and lattice operations are symbolic operations, whose complexity does not depend on the object system \( S \). This type system is powerful enough to attach some polymorphic type information to some restrictions such as:

- \( \text{car}: \{\text{LIST} \cap \{X \text{ set-of}\}\} \rightarrow X \), or
- \( \text{top}: \{\text{STACK} \cap \{\text{range: X}\}\} \rightarrow X \).

\section*{4 The Associated Tools}

\subsection*{4.1 The Query Algebra}

A relational algebra can be formed from some classical operations on binary relations [McL81]. For our model, these operations are composition (written \( \circ \)), which is the binary case of the relational join, intersection (written \( \cap \)), union (written \( \cup \)), inversion (written \( \cdot \)) and cartesian product (\( A \times B \) is the graph of a binary relation). An important property is that each operation can be evaluated on a "set-at-a-time" basis. Moreover, efficient compilation techniques allows us not to physically compute the sets involved in intermediate computations [Ca89]. The algebra \( \text{AR} \) is made from the set of variables \( R \) and the relational operations. Some other operations are introduced to capture the object functions. Precisely we define:

\[
\forall e \in T, \phi(r_1, e, r_2(x,y)) \Rightarrow \exists x_0, x_2, r_1(x_0) \land r_2(x_2) \land c(x_0, x_2) = \text{true} \land x = y
\]
\[
\forall f \in T \cup T^2, \psi(r_1, f, r_2(x,y)) \Rightarrow \exists x_0, x_2, r_1(x_0) \land r_2(x_2) \land f(x_0, x_2) = y
\]

We add the notion of term variable to the induced algebra and obtain the query algebra \( \text{AR} \) defined by

\[
\text{<term(R)> :: <R> | <T> | <R> x <T>}
| <term(R)> \cdot | \text{<term(R)> o <term(R)>}
| <term(R)> \cup <term(R)>
| <term(R)> \cap <term(R)>
| <term(R)> \phi <term(R)> <term(R)>
| <term(R)> <F> \cup <term(R)>
| \text{<term(R)> <T> <term(R)>}
| \text{<term(R)> <R> <term(R)> <term(R)>}
\]

Each term \( t \) of the algebra represents a binary relation on \( \mathbb{O} \) for any given database instance, written \( d(t) \). For instance, the semantics of the last introduced construction is:

\[
d([z:t_1, t_2]) = d(t_2), \text{ with the extension } d(z) = d(t_1).
\]

A logic assertion and an algebraic term are equivalent if they represent the same relation for every database instance. We have shown that each assertion could be translated into an equivalent algebraic term, and reciprocally, which means:

\[\text{Theorem 4.1: The expressive power of the LAURE Algebra and the LAURE Logic Language are similar.}\]

\subsection*{4.2 Rules and Constraints}

Algebraic rules have a simple formulation and a simple semantics. An algebraic rule is a pair \( \langle T, C, R_i \rangle \), where \( T \) is a term of the algebra and \( R_i \) a variable from \( R \). A deductive database is a database model \( \langle S, R \rangle \) and a set of rules \( SR = \{\langle T, C, R_i \rangle, i = 1 \ldots n\} \). Since the union is an operation of this algebra, we can assume that only one rule is defined for each relation. Resolution goal is easy to define:

\[\text{Definition: a database instance is a solution of } \langle S, R, SR \rangle \text{ if and only if } \forall i, c(d(T)) \subset c(d(R)).\]

\[\text{Theorem 4.2: There is a unique minimal complete solution which contains a given database } d_0.\]

Using the completed database instance means that disjunctive information is ignored for deduction. Direct application of Tarski theorem [Ta55] in the semi-lattice structure guarantees the existence of a unique minimal solution that
contains any given (initial) database instance \(d_0\). This database instance is the goal of logic resolution.

Algebraic constraints have a simple formulation and a more complex semantics. An algebraic constraint is a pair \([R_i \subseteq T_i]\), where \(T_i\) is a term of the algebra and \(R_i\) a variable from \(R\).

**Definition:** a database instance satisfies \([R_i \subseteq T_i]\) if
\[
\forall i, \forall x \in O, d(T_i(x)) \subseteq d(R_i(x))
\]
**Definition:** a database instance is a solution of \([R_i \subseteq T_i]\) if and only if it satisfies this constraint and \(d\) is complete.

A database instance satisfies a constraint if there is no violation, and solves it if either a unique value has been chosen or it was decided that no values were possible. This models our desire that constraint resolution means to produce a complete value from an uncomplete value. Unfortunately, there is not necessarily a minimal complete database instance that satisfies a set of constraints. This is due to the fact that a set of constraints may have many different (non-comparable) solutions. The absence of solutions is represented here by the empty value \((d(R)(x) = \emptyset)\).

If we apply the fixed-point theorem in the D lattice, we get a special database instance, which is defined as the maximal value which is contained in all the solutions of the set of constraints. This computation is deterministic and may be used to produce one solution.

**4.3 Translation**

The actual translation of an L\(_2\) query into an algebraic query is interesting because among many possible algebraic translations, there is usually one optimal solution. Translation into the algebraic form is based on rewriting and involves a lot of knowledge about object functions [Co90].

The principle is to solve the equation \(\text{assertion}(x,y)\), while considering that \(x\) is known and that \(y\) is searched. The result of the resolution is a relational algorithm which explains how to get \(y\) from \(x\) and is represented as a term in our relational algebra.

Translation of rules into algebraic rules is straightforward. If a rule \(\text{condition}(x,y) \Rightarrow R_i (x,y)\) is given, the assertion \(\text{condition}\) is translated into a term \(T\) and the equivalent algebraic rule \([T \subseteq R_i]\) is obtained. Algebraic constraints are generated from a logic object constraint in two stages. The first stage is to get relational constraints from an object constraint. This is done with the same rewriting techniques and will transform

\[\text{length}(x) \ast \text{width}(x) = 100\]

into

\[\text{length}(x,y) \Rightarrow (\text{width}(x) \mid 100) \land (y = 100 / \text{width}(x))\] and
\[\text{width}(x,y) \Rightarrow (\text{length}(x) \mid 100) \land (y = 100 / \text{length}(x))\]

The second stage is to transform each condition of a relational constraint into its equivalent algebraic term, and obtain an equivalent algebraic constraint. Here we should obtain:

\[\text{length} \subseteq (\psi (\text{width} / 100) \circ \phi (\text{width} \mid 100))\] \[\text{width} \subseteq (\psi (\text{length} / 100) \circ \phi (\text{length} \mid 100))\]

**4.4 Algebraic Properties**

Differentiation is a formal operation in this algebra. We define the induced functional algebra \(F(R_1, ..., R_n)\) as \(A(0, 1, R_1, ..., R_n)\), where \(0\) and \(1\) are reserved names. Each term \(f\) of this algebra represents a function from \(O \times O\) to \(\text{Pow}(O \times O)\) for each database instance \(d\). By extension, we write this function \(d(f)\), which is defined by:

\[d(0) = \emptyset, d(1) = \{(0,0,0)\}\] and \(d(f) = d(g)\) for all other \(i\):

\(\forall f, g \in F(R), d(f) = d(g)\)

The first important property of this algebra is the existence of a formal operation \(\partial/\partial R_i\) called differentiation on \(A(R) \times R \rightarrow F(R)\), defined by formal rules. We write \(\partial t/\partial R_i\) for the differentiate of the term \(t\) according to \(R_i\) . The interest of differentiation holds in this result:

**Theorem 4.3:**

\[\forall d \in D, \forall R_i \in R_i, \forall (o_1,o_2) \in O \times O, \forall t \in A(R), \text{ if } (o_1,o_2) \text{ does not belong to } d(R) \text{ and if we define a database instance } d' \text{ by } d'(R_i) = d(R_i) \cup \{(o_1,o_2)\}\] and \(d'(R_i) = d'(R_i)\) for all other \(j\):

\[d'(t) = d(t) \cup d(\partial t/\partial R_i(o_1,o_2))\]

\(\partial t/\partial R_i\) is the smallest term from \(F(R)\) which satisfies the previous equation (any other similar term will represent a function that always contains \(\partial t/\partial R_i\)).

The idea of differentiation is very common. It can be found in the RETE algorithm [Fo82], where it is a tree operation. In a relational database, it is defined by a database computation [BRG86]. In this model, we obtain a formal differentiation (on abstract functions instead of database instances), which provides a better implementation.
Abstract interpretation of database relational calculus is a powerful tool to master the computation cost of constraint prediction (Section 5.3). An abstract interpretation scheme \[CC77\] is a tuple \(<D,\prec,D^*,\prec,E,\alpha,\gamma>\) where \((D,\prec)\) and \((D^*,\prec)\) are two lattices, \(E\) and \(E^*\) are two monotonic functions inside each lattice, \(\alpha\) is the abstraction monotonic function from \(D\) to \(D^*\), and \(\gamma\) is the concretization monotonic function from \(D^*\) to \(D\). Certain conditions must be satisfied, such that \(\alpha\) and \(\gamma\) form a Galois connection \[MSS86\].

Here we chose \(D\) as the powerset of \(0 \times 0\) (binary relations), \(\prec\) is the inclusion, and \(E\) is the extended database instance function which returns the value of algebraic terms. The abstract domain is obtained by replacing the powerset of \(0\) by a collection of preferred subsets \(\{\text{small}, R_i(x), [n-m], \text{class}\}\). This means that we want each set involved in the computation to be represented either as a list of fixed size (small), as the value of some object \(x\) according to some relation \(R_i(R_i(x))\), as an interval \([n-m]\) or as a class. When a set does not fit, it is translated into a larger one. The abstract database instance \(d^*\) is defined by induction for each operation of our algebra. For each subset \(S\) of \(0\), \(\omega(S)\) is the smallest set inside the abstract domain that contains \(S\), and \(\gamma\) is the identity. The interest of abstract interpretation is given by the following theorem:

**Theorem 4.4:**
\[
\forall x \in O, \forall t \in A(R), \ d^*(t)(x) \text{ is an abstract set which contains } d(t)(x), \text{ the complexity of the computation of } \\
d^*(t)(x) \text{ is independent from the database instance } d.
\]

Another interesting property of this algebra presented in \[Ca89\] is the ability of efficient compilation. Each relational computation can be mapped into a procedural instruction from the object-oriented language, such that each intermediate set produced in the computation is built virtually.

5. APPLICATION TO LAURE

5.1 Production Rules

The first naive application of differentiation is to get an efficient implementation of the semi-naive technique of fixed-point computation \[SKGB87\]. This is used in LAURE to provide bottom-up evaluation of rules. Because formal differentiation is independent from the database instance, this supports a complete compilation of rules into procedures \[Ca89\]. This technique can be extended to production rules \(\text{condition}(x,y) \Rightarrow \text{action}(x,y)\) where the conclusion of the rule can be any expression of the imperative object-oriented language. LAURE can be used as an expert system shell with a better efficiency that the RETE algorithm. This is due to two facts: the differentiate function is a procedural description of the optimal computation that must be performed and no additional data structure is needed. This technique can also be used to implement integrity constraints such as:

```plaintext
integrity_constraint[
  if [x exists [s = salary(x)]]
  [sm = salary(manager(x))]
  check [s < sm]]
```

Each integrity constraint is a deterministic production rule, which evaluates a test when a logic condition is verified, which may produce an error.

5.2 QSSQD Resolution

Differentiation allows an efficient variation of the query/sub-query algorithm \[Vi86\]. As any query/sub-query algorithm, we use a top-down approach with memoization. When a recursive goal is detected, the exploration stops. The difference in this case is how we complete the temporary answer as soon as new information about the recursive goal becomes available. Because of the differentiation function, this propagation can be performed in a procedural manner \[Ca89\]. The difference (propagation of values as a tree operation vs. as an imperative computation) is exactly similar to the previous one between RETE and LAURE. An important practical consideration is that efficient memoization is easy for binary relations.

This algorithm is sound and complete; it has proven to be efficient even for a multiply recursive example with functions such as the line example. As magic sets \[BNSU86\], it supports a totally compiled implementation. The availability of differentiated functions permits us to memoize any partial subset of a computed goal and to maintain it with respect to positive updates. Since all relations (including derived relations) can be stored extensionally in this model, there is no view update problem.

5.3 Constraint Resolution

Constraint resolution implies backtracking on the set of possible values to find admissible solutions. A deterministic computation to reduce the set of possible solution is called a prediction. As we have seen previously, the computation of
the incomplete database fixed-point is the best prediction that we can make. Unfortunately, this is an unrealistic strategy to implement, since this computation is extremely expensive. Cost of prediction is the main reason why many sophisticated constraint solvers are slower than compiled PROLOG on simple examples.

The resolution strategy that we have chosen is based on two stages. The first is a deterministic computation of an approximation of the fixed-point. Each algebraic constraint is used once to reduce the set of possible values [VD86][DSVH87]. We use the abstract interpretation to master the cost of this prediction. We obtain a bigger database instance (according to the previously defined order, i.e: with smaller possible value sets) which will still contain any possible solutions.

Second stage is an exploration of the subtree of possible database instances, with a backtrack mechanism. The order on database instances allow an efficient implementation of hypothesis testing/bactracking. Each constraint is also used once only to check if a deterministic value can be found (if we know the length of the object of our last example, its width can be deduced directly). The complete algorithm is described in [Ca90]. We have found this compromise between prediction and testing to give the best practical results. An interesting feature of this model is to incorporate rules and constraints in the same data model. The result is that the practical system allows the use of both kinds of resolution to perform a given task, which is useful in an object-oriented database [Ca90].

Since the solution of a set of constraints is considered as a subtree of the possible database instances, its exploration is often interesting. We can use an evaluation function to find an optimal solution, or to make a partition between different kinds of solutions. This is currently performed with the object-oriented language. Hypothetical reasoning is based on the history mechanism in LAURE, which keeps as a stack of increasing database instances. Creating a hypothesis consists of creating a new larger database instance and applying the previous resolution tools to this situation.

Reactive resolution is similar to the update problem. If a solution has been computed, there is a need to find another solution if some input condition varies. The problem is a distance problem and not simply an efficiency problem. Since resolution is non-deterministic, the reason we do not want the system to recompute a new solution from the beginning is that this new solution my be very different from the previous one, even though there exists a "closer" one. The metrics on the solution space is defined by the user through relaxation paths., which tells how a variation in one attribute should affect (if possible) some other object attribute. For instance we may prefer that the variation of some gas pressure (cf. first example) produces a variation on the volume rather than the temperature. The purpose of reactive resolution is to find a solution with as many common values as possible with the old one, where the changed values are obtained from the shortest relaxation paths. This type of resolution was motivated by user-interface applications, where the logic constraint resolution must be dynamically directed by the user's requests.

**Conclusion**

We have presented here a model which is partially a subset of previous proposed models [AK89][CW89][KW89]. The emphasis in this work is on tractability, as opposed to expressive power. With this model were presented a relational algebra and a set of algorithms that permit an efficient implementation. Formal differentiation is a key technique to achieve efficient bottom-up evaluation. It permits us to compile the rules into imperative functions of the object-oriented language, used as demons. These techniques have been used successfully in the development of the LAURE system during the three previous years. The result is that LAURE was successfully used for object-oriented logic applications such as [GGNS90].

**Acknowledgements**

This work has benefited from many fruitful discussions with Shamim Naqvi and Sam Epstein. I am grateful to Michael Kifer; Madhur Kohli, Yehoshua Sagiv and Catriel Beeri for their comments on LAURE. I would also like to thank Drew Adams, Francois Monnet and Diane Hoffoss for their support and encouragements.
REFERENCES


