A predicate transition net for evaluating queries against rules in a DBMS

Christophe de Maindreville, Eric Simon

INRIA

CENTRE DE ROCQUENCOURT

Institut National
de Recherche
en Informatique
et en Automatique

Domaine de Voluceau
Rocquencourt
B.P.105
78153 Le Chesnay Cedex
France

Tél. (1) 39 63 55 11

Rapports de Recherche

N° 604

A PREDICATE TRANSITION NET FOR EVALUATING QUERIES AGAINST RULES IN A DBMS

Christophe de MAINDREVILLE
Eric SIMON

Février 1987

EVALUATION DE QUESTIONS / REGLES DANS UN SGBD RELATIONNEL A L'AIDE D'UN RESEAU DE TRANSITIONS A PREDICATS

Christophe de Maindreville et Eric Simon

Projet SABRE INRIA Rocquencourt
BP 105, 78153 Le Chesnay Cedex, France

Résumé :

Dans ce rapport, nous étudions les problèmes posés par l'intégration d'un langage de règles de production, appellé RDL1, avec un SGBD relationnel. Nous proposons une technique générale de compilation pour transformer les règles de production en un nouveau modèle d'exécution basé sur les réseaux de Petri à prédicats (PrTN). Les principaux avantages de ce modèle consistent en sa puissance de représentation et les techniques d'optimisation qu'il supporte. Après la présentation du modèle, nous proposons plusieurs techniques d'évaluation de questions. Tout d'abord nous charactérisons syntaxiquement une classe de règles, appellées règles ensemblistes, qui peut être exécutées par des programmes de l'algèbre relationnelle étendue. Puis, nous décrivons plusieurs algorithmes d'optimisation d'exécution de ces règles ensemblistes. Ces algorithmes sont utilisés pour (i) transformer le PrTN initial en PrtN optimisé, i.e un PrTN dont l'exécution ne génére pas de résultats intermédiaires inutiles, (ii) déterminer l'ordre dans lequel les règles seront exécutées, (iii) transformer le PrTN optimisé en un programme de l'algèbre relationnelle, (iv) permettre une optimisation globale du programme généré.

Mots clés : langage de règles de production, evaluation de questions, optimisation de questions, réseau de transition à prédicats,programme d'algèbre relationnelle.

A PREDICATE TRANSITION NET FOR EVALUATING QUERIES AGAINST RULES IN A DBMS

Christophe de Maindreville and Eric Simon

SABRE Project - INRIA Rocquencourt
BP 105, 78153 Le Chesnay Cedex, France

Abstract :

In this paper, we consider the problem of integrating a production rule language, called RDL1, with a relational DBMS. We introduce a general compilation technique for transforming production rules into a new execution model based on Predicate Transition Networks (PrTN). The features of this model are its descriptive power and its query optimization and truth maintenance support. After presenting our model, we propose several query processing techniques. We first give a syntactic characterization of rules, called set oriented rules, that can be executed by means of extended relational algebra programs. Then we describe algorithms for optimizing queries involving set oriented rules. They are used (i) to transform the PrTN into an optimized one, that is a PrTN which does not generate useless intermediate results for obtaining the result of the query, (ii) to determine in which order the rules must be executed and (iii) to transform the resulting optimized PrTN into a relational algebra program and (iv) to allow a global optimization of the generated program.

Key-words : production rule language, query processing, query optimization, predicate transition net, relational algebra program

SUMMARY :

1. INTRODUCTION

One way to increase the usability of database technology to support future applications in knowledge base systems is to augment the computational power of relational languages. The most familiar approach, usually refered as deductive database approach, is to integrate a rule based language (generally a subset of a Horn clause language called Datalog) with a relational DBMS. The main problem encountered is the technique used to evaluate a rule program (assumed to contain a large number of rules) against the relational database. The problem is to map the rule language into a target language which is in general a relational language augmented with some specific operations. Several such techniques have recently been proposed [Bocca86, Ceri86, Lozinskii86, Van Gelder86]. They have to satisfy three main requirements : (i) to be general, that is to accept the computation of a wide class of rule programs, (ii) the interface between the rule processing component and the relational DBMS must make use of the set-oriented processing capabilities and the inherent parallelism of the DBMS and (iii) new optimisation techniques must be supported in order to process a program consisting of a large number of rules eventually containing recursive rules.

Our approach consists in integrating a production rule language, called RDL1, with a relational DBMS [Gardarin85]. A production rule consists of a conditional part which is a relational calculus expression and in consequent part which is composed of insertions and deletions of tuples in database relations.Thus, a rule in RDL1 is not a logical formula or a clause but is more in the spirit of productions in languages such as OPS5 [Brownston85]. A semantics of this language is given in [Simon86]. In particular, this language is shown to be more general than Horn clause languages since it allows to express general database updates.

In this paper, we first introduce a compilation technique for transforming general production rules into a new execution model based on Predicate Transition Nets (PrTN). After describing the PrTN model, we define its semantics which naturally captures a forward chaining execution of the rules in a "tuple-at-a-time" way. The second part of the paper is devoted to query processing techniques. We first give a syntactical characterization of rules, called set-oriented rules, which can be transformed into an extended relational algebra expression without changing their "tuple-at-a-time" semantics. Then we consider the case of queries involving only set-oriented rules that we call set-oriented queries. When such a query is submitted, a first PrTN modelling the query and the associated rules is built. We present algorithms which are used (i) to transform the PrTN into an optimized one, that is a PrTN which does not generate useless intermediate results for obtaining the result of the query, (ii) to determine in which order the rules must be executed, (iii) to transform the resulting optimized PrTN into a Relational Algebra Program (RAP) and (iv) to allow global optimization of the generated RAP.

The paper is organized as follows. Section 2 presents the bases of the production rule language and gives examples of rule programs. The next section is devoted to the presentation of the PrTN model and to the method used to represent production rules.The representation of a query and the semantics attached to a PrTN, i.e a rule program, are also given. Section 4 introduces the notion of query processing. A characterization of set-oriented rules is first presented. Then we describe the meta-rule used for ordering the execution of rules among all firable rules at a given time. Section 5 presents the two main algorithms which perform the optimization of set-oriented queries. The model of computation based on PrTN and the algorithms presented in the paper are analyzed in section 6. A qualitative analysis exhibits the advantages of the PrTN over other related methods and shows the values and limitations of the proposed algorithms. Finally, we conclude by pointing out some future directions of research.

2. THE PRODUCTION RULE LANGUAGE

The purpose of this section is to present an overview of the production rule language we use. We give the syntaxic definition of rules without providing their formal semantics since that point will be discussed when presenting the modelization of rules by means of a PrTN. Basic definitions of databases are reviewed. We introduce the syntax and the semantics of the tuple relational calculus that we use. Finally the syntaxic definition of rules is given. A more detailed presentation can be found in [Simon86].

2.1. A tuple relational calculus

A *relational schema* R is a finite set of attributes $\{A_1, ..., A_n\}$. Let dom (A_i) be the domain of values of attribute A_i. A *constant tuple* $t = (c_1, ..., c_n)$ over a relational schema R is a function from R into dom $(A_1) \cup$ dom $(A_2) \cup ....\cup$ dom (A_n) such that $\forall i \ 1 \leq i \leq n, c_i \in$ dom(Ai) . An *instance* of a relational schema R, (sometimes called a relation), is a finite set of constant tuples over R. A *database schema* is a finite set of relational schemas. Finally, an *instance* I over a database schema S is a function from S such that for each R in S , I(R) is an instance over R.

Example :

Let Parents be the relational schema {Father, Mother, Child} and dom (Father) = dom (Mother) = dom (Child) = set of names. An instance of Parents is a finite set of triples of names. Thus, an instance of Parents could be I(Parents) = {(Father = Patrick, Mother = Cati, Child = Sarah), (Father = Serge, Mother = Sophie, Child = Jeremie), (Father = Robert, Mother = Raymonde, Child = Kador)}

We specify an extended version of the relational calculus of [Codd71] based on a multi-sort

logic where terms are typed. This relational calculus has two types of predicate symbols : The relational predicates are those for which an interpretation corresponds to the instances of the relational schemas and allow the user to define the type of a tuple variable which ranges over a relation. The non relational predicates are defined below from the syntaxic definition of a formula. Before introducing the syntaxic definition of a formula, a vocabulary is introduced. It is composed of constants c1,c2,..., tuple variables x, y, ..., function symbols f1, f2, ..., attribute names A1,A2,... (which may be considered as specific function symbols), relational predicate names R1, R2,... and non relational predicate names.

Axiom 1 : *Terms* are recursively defined as :

- Every constant is a term,
- Every variable is a term,
- If t1,t2,...,tn are terms and f is a n_ary function symbol then f(t1,t2,...,tn) is a term.

We particularize one type of function defined by the attribute symbols whose meaning is given below:

- If A is an attribute name and x a tuple variable name then A (x) (also denoted x.A) is a term. It denotes the attribute value of x corresponding to A.

We also generalize the notion of constant tuple introduced above by the more general notion of *tuple*. Indeed, let R (A_1, ..., A_n) be a relational schema, t1,t2,...,tn be some terms; if dom (ti) is included in dom (A_i) for each i , i ∈ {1,..., n} then (t1,t2,...,tn) is a *tuple* over the relational schema R. Examples of terms are given below.

Examples :

The constants Sarah, Jeremie are terms.
The attributes x.A, x.B are terms.
The functions of terms + (x.A, x.B), division (y.C, y.D) are terms.
The tuple (First Name = Joe, Family Name = Dalton, Wanted = x.reward) is a term.
PARENTS (x) is a range predicate.
PARENTS (x) AND ANCESTOR (y) is a range predicate.
PARENTS (x) AND ANCESTOR (x) is not a range predicate.
≤ (x.A, y.B) is an atomic predicate.

In the above examples prefix notation is used for functions and atomic predicates. In the rest of the paper we shall use an infix notation.

Axiom 2 : A *range predicate* is defined as follows :

- If R is a relational predicate name and x is a tuple variable name then R(x) is a range

predicate.

- let U and V be two range predicates with no common variable then U AND V is a range predicate.

Axiom 3 : An *atomic predicate* is defined as follows :

- TRUE and FALSE are atomic predicates,
- Let P be a n-ary non relational predicate name, t_1, t_2,..., t_n be some terms then P (t_1, t_2,..., t_n) is an atomic predicate.

Axiom 4 : A *sub-formula* is recursively defined as follows :

- Every atomic predicate is a sub-formula,
- Let F be a sub-formula containing a variable x and R a relational predicate name then ($\exists$ x $\in$ R) (F) and ($\forall$ x $\in$ R) (F) are sub-formulas,
- Let P and Q be two sub-formulas then P AND Q, P OR Q, NOT P, P => Q are sub-formulas.

Axiom 5 : A *formula* is defined as follows :

- A range predicate or a sub-formula are formulas,
- Let U be a range predicate and F be a sub-formula then U AND F is a formula,
- Let F and G be two formulas then F OR G is a formula.

We come now to the interpretation of a formula. Let D be the finite set of sorts composed by all the domains of attributes defined over a database schema *S* and all the instances of relational schemas. Thus, D = {dom (A_1), dom (A_2), ... , I (R),....}. The interpretation of a given formula F is given by an assignment of the terms of F, the atomic predicates of F and an assignment of the free variables and range predicates of the formula. The assignment of terms is given by :

- Each constant corresponds to an element of a sort of D.
- Each function symbol corresponds to a particular function over the sorts of D.
- Each tuple corresponds to an element of a sort of D.
- Each non relational predicate name corresponds to a particular predicate over the sorts of D.
- TRUE and FALSE have the respective meaning true and false.

The assignment of free variables consists in replacing each free variable of F by a constant tuple over a relational schema R. The range predicate R (x) takes a true value if and only if there exists a constant tuple in the instance of R which can be substituted to the variable x. Finally, the evaluation of the formula ($\forall$ x $\in$ R)(F) leads to evaluate F (r_1) $\wedge$ F (r_2) $\wedge$ ... $\wedge$ F (r_n) where r_1, r_2, ..., r_n are all the constant tuples over R. In the same way, the formula ($\exists$ x $\in$ R)(F) is evaluated as F (r_1) $\vee$ F (r_2) $\vee$ ... $\vee$ F (r_n).

In these formulas we say that x is a quantified variable over R and ($\forall$ x $\in$ R) and ($\exists$ x $\in$ R) are called *quantified range predicates.*

Definition : A formula F is said well formed (wff) if and only if one of the three following conditions holds :

- F has no free variables.
- F is a range predicate.
- F is of the form U AND G and all free variables appearing in the sub-formula G also appear in the range predicate U which is conjunctively connected to it.
- F is a disjonction of well formed formulas.

In the sequel, we assume that all formulae are well formed.

Examples :

The range predicates R_1 (x), R_1 (x) AND R_2 (y) are wff.
($\exists$ x $\in$ R_1)(x.A = 5) is a wff.
($\exists$ x $\in$ R_1)(x.A = y.B) is not a wff.
R_2 (y) AND ($\exists$ x $\in$ R_1)(x.A = y.B) is a wff.
R_1 (x) OR R_2 (x) is a wff.

Constant aggregates are added to the tuple calculus. The vocabulary is first enriched with aggregate symbols AGGR1, AGGR2,...

Definition : Let t be a term, AGGR an aggregate symbol, P a list of range predicates, and F a sub-formula, then AGGR ({t ; P; F}) is a constant aggregate .

A tuple variable x, appearing in a range predicate of the list P, is *bound by the operator* AGGR. Each variable appearing in the constant aggregate's term t and not bound by AGGR is a free variable in this term. We will impose that all the variables appearing in the term t are bound by AGGR. Such a restriction does not apply to the variables of the sub-formula F. A constant aggregate is interpreted as the aggregation of t according to the operator AGGR for all the substitutions of tuple variables of P satisfying the formula U and F where U is the conjonction of all the range predicates of the list P. In the sequel, we use the following notation to denote the constant aggregates : AGGR { t | U and F }.

Example :

COUNT {a.Desc | ANCESTOR (a) and a.Asc. = 'Joe Dalton'} is a constant aggregate.

In order to get an explicit representation of negation, the calculus is enriched with a new type of sub-formula called *negative range predicate*. This extension does not augment the expressive power of the previous calculus but is just another notation for universal quantification.

Definition : Let R be a n-ary predicate symbol, x a tuple variable symbol ranging over an instance of T and $(t_1,\ldots, t_n)$ a tuple over R, then :

(i) NOT R (x) is a negative range predicate if T and R are identical relational schemas,

(ii) NOT R ($A_1 = t_1, \ldots, A_n = t_n$) is a negative range predicate.

A Negative Range Predicate (NRP) is a sub-formula.

Examples :

NOT Penguin (x) is an NRP but is not a wff.

Bird (x) AND NOT Penguin (x) is a wff containing an NRP.

Parents (x) AND NOT Ancestor (Asc = x.child) is a wff containing an NRP

Thus, our use of negation is restricted to Range Restricted Formulas [Demolombe84].

2.2. Syntaxic definition of rules

The general form of a rule is the sentence : Condition → Action. To define the precise notion of rule, we need the auxiliary concepts of condition and action.

Definition : A condition over a database schema *S* is a well formed formula of the tuple relational calculus.

The notion of action can now be presented.

Definition : An action over a database schema *S* is defined as follows :

(i) For R in *S* and for each tuple $t = (t_1, \ldots, t_n)$ over R $(A_1, \ldots, A_n)$ then +R ($A_1 = t_1, \ldots A_n = t_n$) and - R ($A_1 = t_1, \ldots A_n = t_n$) are actions (respectively called insertion and deletion in R).

(ii) Let x be a tuple variable ranging over an instance of T, for R in *S* , +R (x) and - R (x) are actions provided the two relational schemas R and T are identical.

(iii) If A1 and A2 are two actions then A1, A2 is an action over a database schema *S*.

Examples :

+ ANCESTOR (Asc = Cati, Desc = Sarah) is an action over ANCESTOR.

+ ANCESTOR (Asc = x.Father, Desc = x.Child) is an action over ANCESTOR.

- EMPLOYEE (Name = Jules, Dept. = toys) is an action over EMPLOYEE.

- ANCESTOR (Asc = 100) is not an action because 100 is not an element of dom(Asc).

Missing attribute values within the arguments of an action generate null values. For example, in the action + ANCESTOR (Asc = Patrick) the missing attribute value for Child indicates a null value by default.

Let C be a condition, we call free (C) the set of free variables appearing in C. Let A be an action we call Var (A) the set of variables appearing in A.

We come now to the definition of a rule.

Definition : A rule is defined as follows :

If C is a condition and A is an action and free (C) $\supset$ Var (A) then

$C \rightarrow A$ is a rule.

Examples :

- PARENTS (x) $\rightarrow$ + ANCESTOR (x) is a rule.
- PARENTS (x) $\rightarrow$ + ANCESTOR (y) is not a rule.
- PARENTS (x) $\rightarrow$ + ANCESTOR (Asc = x.Father, Desc = x.Child) is a rule.
- EMPLOYEE (x) AND x.Name = Jules $\rightarrow$ - EMPLOYEE (Name = Jules, Dept. = toys), + EMPLOYEE (Name = Jules, Dept = sports) is a rule.
- PARENTS (x) AND ($\exists$ y $\in$ EMPLOYEE)(y.Name = Jules) $\rightarrow$ + EMPLOYEE (Name = y.Name, Dept = toys) is not a rule.

Definition : A rule program is a finite set of rules.

It is convenient to specify rules using a layered methodology. A first layer of rules defines a set of *target relations* which are directly derived from a set of *base relations*. At the next layer, the previous target relations may become additional source relations. Thus, a second layer of rules may be specified to model richer rules using base and first level derived relations. This process can be iterated to n levels. The notion of layer leads to the notion of rule module.

Definition : A rule module is a tuple < S0,T, RU > where S0 is a set of relational schemas (called source relations), T is a set of relational schemas (called target relations) and RU is a set of rules satisfying the following conditions :

(i) Each relation of T is only defined in this rule module, that is it does not appear in the right hand side of a rule belonging to another module.

(ii) For each relational schema R of RU which is not in S0, a deletion (i.e - R (t) or - R (x)) can appear in the action part of a rule iff an insertion (i.e. + R (t) or + R (x)) appears in the action

part of another rule in the same module.

The last condition of this definition expresses that a rule can only delete tuples from a relation iff this relation has been previously defined within the same module. Thus, a rule module defines two types of relational schemas :

(1) Source relations which have been defined into a previously defined rule module or which are base relations,

(2) Target relations which are derived from source relations in a constructive way.

The relations which are neither source or target relations are called temporary relations.

Examples :

MODULE ANCESTOR ;
target
ANCESTOR (Asc : text, Desc : text) ;
Rules
PARENT (x) AND ANCESTOR (y) AND
x.Child = y.Asc → + ANCESTOR (Asc = x.Parent, Desc = y.Desc),
PARENT (x) → + ANCESTOR (x);
end.

Let us assume a base relation WIRE (W#, Origin, Ext.), we can compute the "reduction" of the WIRE relation by the following module.

MODULE REDUCTION
target
RESULT (ORIGIN : integer ; EXT : integer),
Rules
WIRE (x) → + RESULT* (x)
RESULT* (x) AND RESULT* (y) AND x.Ext = y.Orig AND NOT RESULT* (W# = y.W#, Origin = x.Ext) AND NOT RESULT* (W# = y.W#, Ext = x.Ext)
→ - RESULT* (x), - RESULT* (y),
+ RESULT* (W# = x.W#, Origin = x.Origin, Ext = y.Ext),

RESULT* (x) AND RESULT* (y) AND x.Origin = y.Origin AND x.Ext = y.Ext AND x.W# ≠ y.W#
→ + RESULT* (W# = x.W#, Origin = x.Origin, Ext = y.Ext),
- RESULT* (x), - RESULT* (y),

RESULT* (x) → + RESULT (Origin = x.Origin, Ext = x.Ext) ;
End.

These rules replace two serial or parallel wires by one resulting wire. Note that RESULT* is a temporary relation.

2.3. Considerations for the semantics of rules

The semantics of RDL1 presents two particularities. First, contrary to many logic programming languages such as Prolog, the rules in a program do not have to be ordered by the programmer for expressing the way to solve a problem or to derive data. The rules in RDL1 can be given in any order. Similarly, the order of the predicates in the condition part of a rule has no effect on the result. When a set of rules is declared, it is analyzed and compiled into an internal execution model that we shall detail in the next section. During this analysis, the system keeps all the possibilities opened for ordering the rules. Thus, the execution strategy is not pre-compiled. The system decides at run time only in which order the rules have to be fired.

A second feature of the language is that the interpretation of the action part of a rule is not performed as a sequential execution of insertions and deletions. Rather, an action is seen as an atomic database update. Suppose for example that a relational predicate P appears as the argument of several insertions and deletions in the action part of a rule. Let us denote P^+ and P^- respectively the set of tuples inserted into P by insertions + P (...) and the set of tuples deleted from P by deletions - P (...). Figure 1 below represents the sets P, P^+ and P^-.

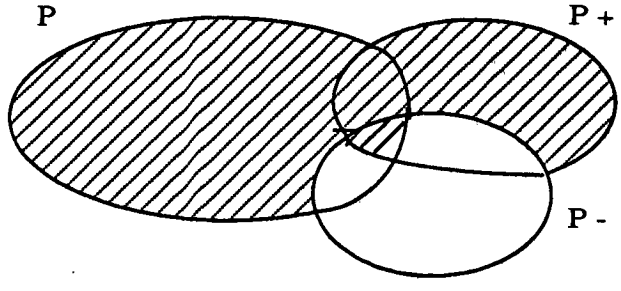

Figure 1 : result of an action over P

The effect of the action part of the rule is described in the figure. The dashed part represents the instance of P after executing the action. Formally, the instance of P is equal to :

$$[(P \cup P^+) - P^-] \cup (P \cap P^+ \cap P^-)$$

This property has several consequences. First, it guarantees that the order of insertions and deletions in the action part is irrelevant. Second, it nullifies an action which inserts and deletes the same tuple.

Examples :

P (x) AND x.A = a → + P (a), -P (a) has no effect on P.

P (x) AND x.A = a → + P (b), - P (b) has no effect on P.

P (x) AND x.A = a → + P (b), - P (x) changes the value of attribute A from a to b.

3. MODELLING RULES USING A PREDICATE TRANSITION NETWORK

3.1. Definition of the Production Compilation Network

As pointed out in section 2, we follow a compilation approach for transforming a rule program into an execution model which serves as a basis for query processing. More precisely, the analysis of a rule program in RDL1 is performed by the following steps. First, a syntactic and semantic analysis of the rules is done. The next step transforms the rule program into an execution model based on Predicate Transition Nets. The third step performs some consistency checks over the net. Finally, the last step stores the net in a pre-compiled form in the rule base by incrementally updating the already existing PrTN. The diagram on figure 2 summarizes the compilation process. In this section, we first describe the model of PrTN that we use. Then the second and fourth step of the compilation process given above are detailed. The reader interested in the two other phases is referred to [Gardarin85].

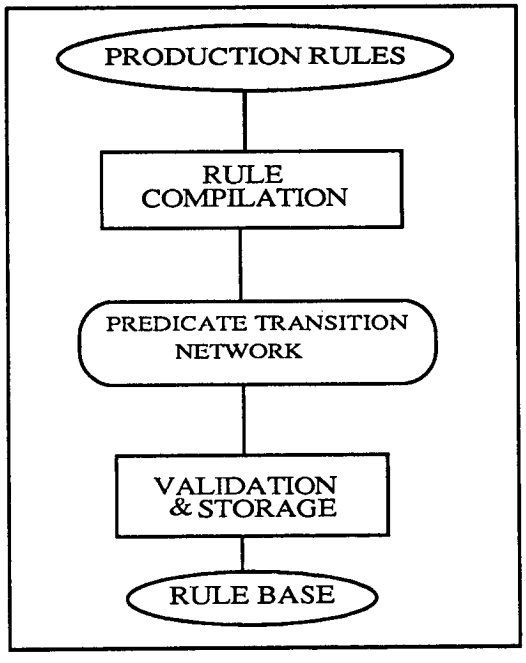

Figure 2 : Production rule compilation process

Predicate Transition Nets (PrTN) derive from Petri nets [Peterson81]. They have been shown to be a powerful tool for modeling production rules in Expert Systems [Giordana85] or for operational specification of process control systems [Bruno86]. The main difference with Petri nets

consists in the fact that tokens are structured objects carrying values similar to database tuples, and furthermore, that transition firing can be controlled by imposing conditions over the token values. A formal definition of PrTN can be found in [Genrich81].

A PrTN consists of [Genrich 81] :

(1) a bipartite, directed graph (P,T,F) where P and T are called *places* and *transitions* respectively, and F is a set of directed arcs, each one connecting a place $p \in P$ to a transition $t \in T$ or vice versa that is $(P \times T) \cup (T \times P) \supset F$. Places correspond to predicates with variable extensions, and transitions represent classes of elementary changes of extensions.

(2) A labeling of the arcs with formal sums of tuples of variables ; the length of each tuple is the arity of the predicate connected to the arc.

(3) A structure Σ, defining a collection of typed objects together with some operations and relations applicable to them. Formulas built up in Σ can be used as inscriptions inside some transitions.

(4) A function k, from P to the set of nonnegative integers, assigning to each place an upper bound to the number of copies of the same token the place can carry at the same time. K (p) is called the *capacity* of p.

A token $r = < a_1, a_2, \ldots, a_r >$ in a place $p \in P$ denotes the fact that the predicate $P(x_1, x_2, \ldots, x_r)$ corresponding to that place is true for that particular instantiation of the tuple of arguments contained in the token r.

The following associations can be made between rule and PrTN concepts.

Rules	PrTN
Predicates	Places
Rules	Transitions
Facts	Tokens
Formulas	Transition's inscription

As a particular case, a theorem in [Genrich78] states that each well formed formula in first order logic can be represented in a PrTN by means of a set of dead transitions (i.e. without transition's formula).

We now describe the method used to model our production rules using a model derived from PrTN that we call Production Compilation Network (PCN). The table portrayed in figure 3 represents the possible associations between rule and PCN concepts. The following method is used.

RDL.1	PCN
RULE	TRANSITION T
RANGE PREDICATE P	PLACE P
Free variables $X_1,\ldots, X_n$ ranging over P or Existentially quantified variables	ARC (P,T) labelled by : $+ X_1 +\ldots +X_n$
Either bound or universally quantified variables $X_1, \ldots, X_n$ ranging over P	ARC (P,T) labelled by : $(*X_1) +\ldots+ (*X_n)$
ACTION + P (t)	ARC (T,P) labelled by + t
ACTION - P (t)	ARC (T,P) labelled by - t
SUB-FORMULA	Transition's inscription

Figure 3 : Correspondance table between production rules and the PCN

(a) Places : any relational predicate P which is defined in a set of rules is modeled by a unique place P. This can be the case for either a range predicate P or an action over a relation P.

(b) Transitions : each rule is modeled by a transition t.

(c) Place to Transition Vertices : an arc connects a place P to a transition t iff the relational predicate P appears in the left-hand side of the rule modeled by t.

Such an arc can receive two types of labels :

- an arc (P,t) is labelled with the formal sum of tuple variables $x_1+x_2 + \ldots + x_n$ iff $x_1, x_2, \ldots, x_n$ are free variables ranging over P in the premisse of the rule.
- an arc (P,t) is labelled with a symbol (.) iff no free variable range over P and P appears in the sub-formula of the rule's condition.t

(d) Transition to Place Vertices : positive and negative arcs are distinguished :

- A positive arc connects a transition t to a place P iff P appears as the argument of an action +P (...) in the conclusion of a rule t (i.e. the right hand side of a rule). Such an arc is labelled with the formal sum of tuples appearing as arguments in each action + P (...) over P.
- A negative arc connects a transition t to a place P iff P appears as the argument of an action - P (...) in the right hand-side of the rule t. Such an arc is labelled with the formal difference (symbol "-") of tuples appearing as arguments in each action - P (...) over P.

(e) Transition's inscription : the inscription of a transition t is given by the sub-formula used to built the left hand-side of the rule t. Remember that the premisse of a rule is a formula of type $U \wedge F$ where U is a range predicate and F is a sub-formula.

We now give one example of PCN built from a set of rules. *Figure 4* represents the module ANCESTOR given in RDL1 in section 2. Two relational predicate symbols appear in the rules. They lead to the places PARENT and ANCESTOR. The first rule PARENT (x) $\rightarrow$ + ANCESTOR (x) is modelled by the transition t1,whose inscription is the sub-formula TRUE. The second rule is modelled by the transition t2, whose inscription is the sub-formula x.Child = y.Asc.

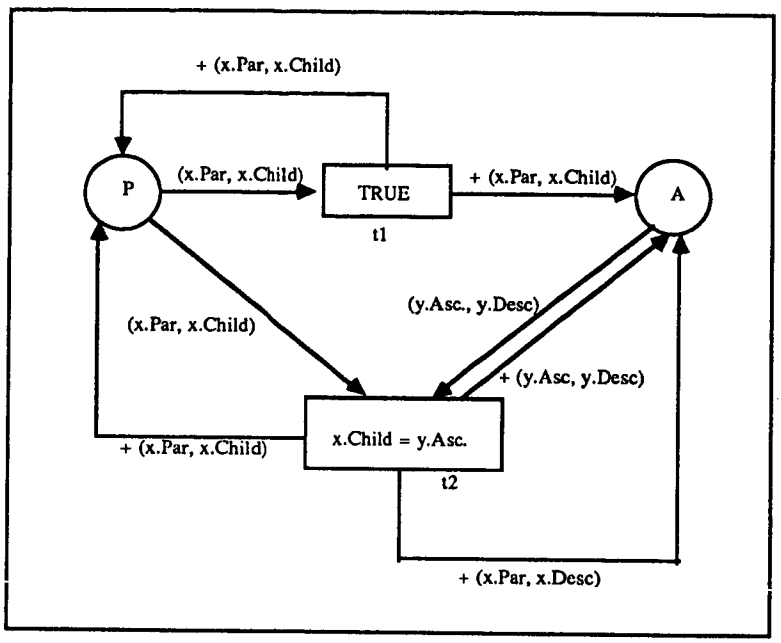

Figure 4: PCN for the Ancestor rules

Tokens are not represented on this net. In fact, the tokens correspond to the tuples belonging to both PARENT and ANCESTOR relations. Intuitively, each time transition t1 is fired it removes a token from PARENT and put it in ANCESTOR. Therefore, in order to maintain the tokens in PARENT which is a base relation, we added an arc from t1 to PARENT labelled by + (x.Par, x.Child).

In the sequel, we consider that the net is built in such a way that the tokens are always kept in the input places of transitions excepted when explicitely removed by negative arcs. For the sake of simplicity we can delete all the bold arcs and replace all the (P,T) oriented arcs by unoriented ones. Also, tuples on the labels can be replaced by the tuple variables. This leads to the simplified Net portrayed on figure 5 (a).

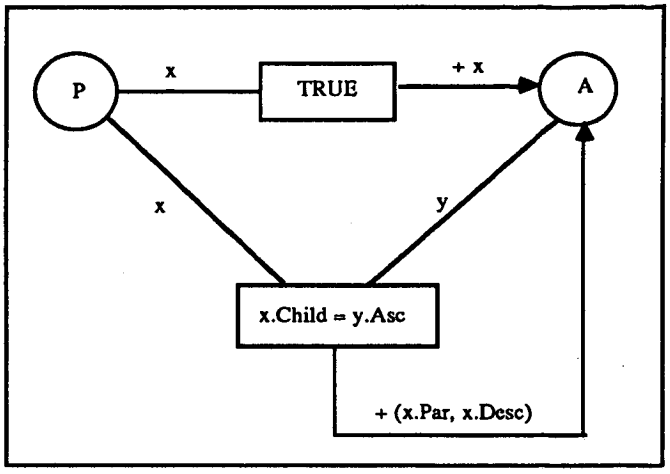

Figure 5 (a): Modified PCN for the Ancestor rules

We give on *figure 5 (b)* the PCN representing the module REDUCTION of section 2. The two recursive transitions can only be enabled if the place R* contains at least two tokens. When two tokens satisfy the transition's inscription t2 or t3 the transition can be fired and the two input tokens are deleted from R*.

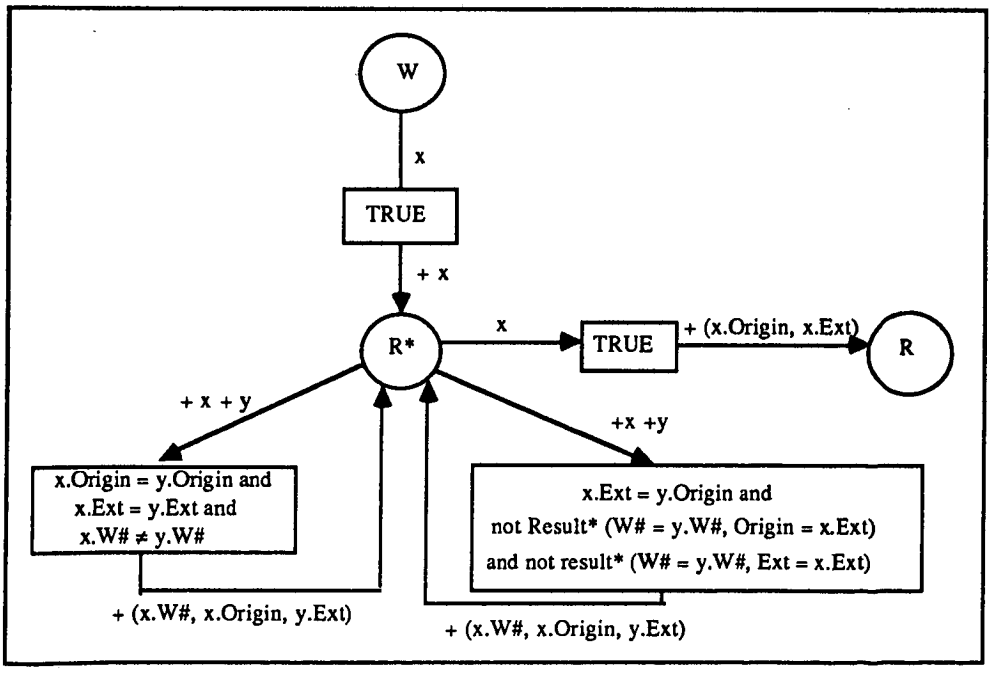

Figure 5 (b): PCN for the module REDUCTION

3.2. Semantics of a PCN

In this section, we give precise definitions of the semantics of a PCN. Two fundamental notions are first introduced.

Definition : A marking is a distribution of tokens over the places of a PCN. An initial marking consists in initializing a PrTN with a particular marking. A base marking is an initial marking where tokens representing database tuples are only distributed in places corresponding to base relations. A source marking is a marking in which only source relations receive tokens.

Definition : A transition T is enabled whenever the three following conditions are satisfied.

(i) Each input place P of T contains at least as many tokens as specified by the number of tuples figuring on the label of the arc (P,T). A tuple of the form (.) counts for zero.

(ii) The tokens occuring in the input places of T have values satisfying the transition condition.

(iii) The capacity of each output place of T is not exceeded by the transition firing.

A restricted semantics of our type of PrTN is now presented. In fact, in all the applications, it makes no sense to draw the same tuple twice using the same rule. Therefore, we impose the capacity of each place, that is the number of copies of the same token a place can carry at the same time, to be bound to 1. This leads to the notion of duplication free PCN. Thus, we have :

Definition : A duplication free PCN is a PCN where :

(i) two tokens of equal value in the same place are merged into a single token,

(ii) a transition which would produce into a place already existing tokens is not enabled.

In practice, condition (ii) in the above definition is easily checked by transforming the transition inscription of the rule. For example, consider the rule PARENT (x) $\rightarrow$ + ANCESTOR (x). The transition's inscription of this rule implicitely becomes : NOT ANCESTOR (x).

When a transition is enabled it can be *fired*. Firing a transition produces several actions. First, it removes from each input place P a number of tokens equal to the number of positive symbols corresponding to the free variables of the transition's inscription labelling the arcs (P, T). Then, it adds to each output place P' the tokens specified by the positive symbols of the label of the positive arcs (T, P'). The third and last action removes from each output place P' the tokens specified by the negative symbols of the label of the negative arcs (T, P').

At a given time, several transitions can be enabled. This set of transitions is called the *conflict set*. Therefore, a PCN evaluator has to execute the following procedure portrayed on figure 6.

```
procedure evaluate (PCN, Initial marking)
    M ← Initial marking
    repeat
      compute the conflict set ,
      select a transition T in the conflict set,
      compute M = Reachable marking from M by firing T.
    until a halting condition is met.
```

Figure 6 : A PCN evaluator procedure

This procedure eventually terminates if a halting condition is met, which means that a stable marking is obtained. The notion of stable marking is explained below.

Definition : For a given initial marking, a place P has a stable marking iff none of its input transitions is enabled.

Definition : A transition T is stable for an initial marking iff all its output places have a stable marking. Then, a PCN has a stable marking for an initial marking iff all its transitions are stables.

In [Gardarin85] we give some sufficient syntaxic conditions for a net to have a fixpoint. The notion of deterministic rule is now detailed.

Definition : A transition T is deterministic iff, for any initial marking, the output places of T have a unique stable marking. A PCN is deterministic iff for any initial marking the net has a unique stable marking.

A very important point is that when considering the semantics of a rule module, the instances of the temporary relations are not taken into account. For instance, the PCN associated with the following module is deterministic if we assume that P {Count} is a source relation, Q {Count} is a temporary relation and GO {Signal}is a target relation.

P (x) →+ Q (x)

Q (x) →+ Q (Count = x. Count + 1), - Q (x)

Q (x) AND x.Count ≥ 100 →+ GO (Signal = 'Start')

Several remarks are in order. First, it is clear that, in general, a PCN has no stable marking and is not deterministic. Howewer, we wish to accept the computation of rules which are not

deterministic but which can lead to deterministic rule modules as shown in section 2.2. Another example is given by the module REDUCTION defined in section 2.2. This set of rules is not deterministic but it presents a straightforward deterministic interpretation. In the next section, we consider such non deterministic programs composed of deterministic rules. The second remark is that a stable marking for a PCN is not easily checked in the general case. As we shall see, an important aspect of set oriented rules is that a stable marking can be efficiently detected.

3.3 Rule base storage

Rules are stored into an internal form directly derived from the PCN representation. More precisely, when a rule module is defined, the form given by the user is saved into a file and then the rule module is compiled into a PCN. This net is added to the rule base by incrementally updating the PCN representing the rule base. This addition will only cause the insertion of new tuples without modifying the existing ones. All syntactic consistency checks are performed on the new defined module independently of the rest of the rule base. A rule base is represented by means of three database relations. The first relational schema is PLACE_TO_PLACE (Module#, I_Place#, I_Rec, Input-Symb, Trans#, Output-Symb, O_Place#, O_Rec) where :

- Module# is an identifier for a module
- I_Place# is an identifier for an input place in a rule
- O_Place# is an identifier for an output place in a rule
- Trans# is a transition number, that is a rule number
- Input-Symb indicates the symbols appearing on an input arc of a transition
- Output-Symb indicates the symbols labelling an output arc of a transition

This first relation stores all the arcs going from a place to another place via a transition. The second relational schema is TRANSITIONS (Module#, Trans#, Rec, Sign, Trans-condition) where Trans-condition is the transition's inscription. Finally the last relational schema is MODULES (Module#, Rule-base, Module-name) where :

- Rule-base is the name of a rule base
- Module-name is the name of a rule module

All the relational schemas of the base or derived relation are described using other specific meta-databases relations. Pre compiled information, useful for query processing, are included in the rule base storage. Two types of information are derived; they consist in detecting which places and transitions are recursive. We first precise these two notions :

Definition : A place P is called recursive iff there exists a path from P to P in the PCN.

For instance the place A, standing for the ANCESTOR relation, in the PCN portrayed on *Figure 7* is recursive.

Definition : Let $Pre_p(t)$ the set of all the antecedent places of a transition t and $Pos_p(t)$ the set of all the posterior places of t . A transition t is recursive iff there exists a recursive place P and a recursive place Q such that $P \in Pre_p(t)$ and $Q \in Pos_p(t)$.

For instance, the transition t2 of the PCN portrayed on *Figure 7* is recursive.
The attribute Rec of TRANSITIONS indicates if a transition is recursive or not while the attributes I_Rec and O_Rec of PLACES_TO PLACES indicate if a place is recursive or not.

Note that these informations are pre-compiled module by module. Thus, the definition of a new module cannot change the value of the above attributes in the already stored rule modules. The last pre-compiled information concerns transitions.

Definition : A transition t is positive (resp negative) iff all its output arcs are positively (resp negatively) labelled.

A positive transition represents a rule composed only of insertions, and a negative transition represents a rule composed only of deletions, (attribute Sign in TRANSITIONS).

The proposed rule base storage presents two main advantages. First, it allows to store rules in a pre-compiled form using the descriptive power of the PCN model. Second, a rule base is stored in a very compact form which is easily maintenable. This last property permits efficient access to the relevant rules for solving a query which constitutes the first step of query processing. We now describe this phase.

A query can be seen as a production rule represented by a single transition, named t$, several input places and a single output place, named $, representing the result of the query. Thus, a query can be represented as the combination of two PCN. The first one represents all the rules needed to define the extension of the input-places of the query. The second net represents the query itself. The retrieval of relevant rules to solve a query can be modelized by a query PCN.

We assume two rules defining a derived relation PERTINENT_RULES (Module#,I_Place#, I_Rec, Input_Symb, Trans#, Output_Symb, O_Place#, O_Rec, Goal) noted PR; We also abbreviate TRANSITIONS by T and PLACE_TO_PLACE by PTP. The retrieval of the pertinent rules consists in traversing recursively the query PCN from the $ place and at each step storing the tuples of the relation PTP in the relation PR. The two rules are:

PTP (x) $\rightarrow$ + PR (x.Module#, x.I_Place#, x.I_Rec, x.Input_Symb, x.Trans#, x.Output_Symb, x.O_Place#, x.O_Rec, x.O_Place)

PR (x) **and** PTP (y) **and** x.O_Place = y.O_Place

→ + PR (y.Module#, y.I_Place#, y.I_Rec, y.Input_Symb, y.Trans#,
y.Output_Symb, y.O_Place#, y.O_Rec, x.O_Place)

If the user query concerns the derived relations P1,...,Pn, the following QUEL-like query to look for all the antecedent places and transitions of the P1,...,Pn places will be generate to the system.

RETRIEVE PR.*, T.*

WHERE PR.Trans = T.Trans **AND** PR.Goal = P1 **or** P2 **or** ... **or** Pn ;

The execution of this single recursive query is illustrated on *Figure 7*, by means of a PCN. The $ place represents the query result which contains all the elements to build in main memory the relevant PCN. In practice, a specific code is written to efficiently compute this query PCN and to build the resulting (user) query-PCN in main memory.

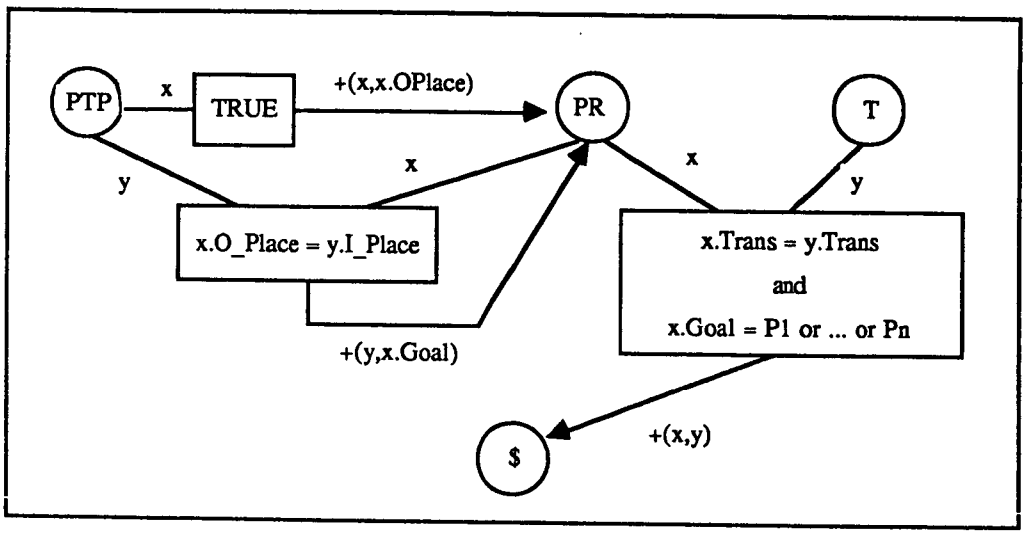

Figure 7: Query PCN retrieving the pertinent rules.

4. QUERY PROCESSING

4.1 Introduction

A query PCN poses the essential problem of choosing at each step, the next transition to be fired as well as the tokens which will be used in this transition firing. The purpose of this section is to describe the control strategy used by the system to evaluate a restricted class of query PCN. We propose two basic techniques to obtain an efficient execution and to guarantee a deterministic execution of a query :

(i) To exploit, whenever possible, the set-oriented processing capabilities of the relational DBMS for evaluating a single transition.

(ii) To provide a meta-rule which permits an ordering in the transition's firing and which guarantees a deterministic execution of a large class of rule programs.

4.2 Set oriented rules

As described in section 2, the rule semantics leads to what is usually called a "tuple-at-a-time" computation. In order to compute efficiently a PCN with a RDBMS, we wish to transform this "tuple-at-a-time" computation into a set at a time one using a relational algebra program. This means that given a transition t, our objective is to transform the transition condition of t into a relational algebra expression which is executed using all the tokens of the input places of t. In this case, we say that we performed a *relational computation* of the rule modeled by t. Then, a rule R is said to be *set oriented* iff a relational computation of R leads to the same result than the computation induced by the semantics of the PCN.
First, let us recall the semantic of a request of the relational calculus:

Definition : A request of the relational calculus is an expression of the following form :

Q = {(t1,t2,...,tn) / <Formula>}

where <Formula> is a well *formed formula* and t1,...,tn are terms. The tuple (t1,...,tn) is called the projection list of the request. The semantics of a request consists to determine all the instanciations of the free variables figuring in <Formula> and then for each instanciation to assign a value to (t1,...,tn).

In the following we assume that all the rules have a fixed-point. In order to compute the rule with the relational algebra, we associate a sequence of requests of the relational algebra with each rule.
Let R : C → A a rule. For simplicity reasons we shall restrict ourselves to rules of the two following forms :

Ri : P1(x1) **and ... and** Pn(xn) **and** C(x1,...xn, y1,..., yq) → + P1 (t)

Rd: P1(x1) **and ... and** Pn(xn) **and** C(x1,...xn, y1,..., yq) → - P1 (t).

where xi, yi are variables and t a tuple over P1.
For a rule Ri, let us consider the following sequence of relational requests :
$Q_A = Q^0 \cdots Q^N$ where Q^0 = {t / P1 (x1)} Q^1 = {t / Q^0 (x1) **and ... and** Pn (xn) **and** C(x1,...xn, y1,..., yq)}, ..., Q^N = {t / Q^{N-1} (x1) **and ... and** Pn (xn) **and** C(x1,...xn, y1,..., yq)}.
For a rule Rd, let consider the following sequence of relational requests :
$Q_A = Q^0 - \ldots - Q^N$.

Note the computation of theses sequences of relational expressions consists in a classical forward chaining computation of a production rule.

Examples :

- Consider the rules which define the classical ANCESTOR relation :

PARENT (x) $\rightarrow$ + ANCESTOR (x)

PARENT (x) **and** ANCESTOR (y) **and** (x.Child = y.Asc) $\rightarrow$+ ANCESTOR (x.Parent, y.Desc)

Then, the relational computation of these rules consists in computing the expression $Q^0 \cup \cdots \cup Q^N$ where for each i, $0 \leq i \leq N$, Q^i is a relation resulting from a relational request.

Q^0 = { t = (x.Parent, x.Child) / PARENT (x) }

Q^1 = { t = (x.Parent, x.Desc) / PARENT (x) **and** Q^0 (y) **and** x.Child = y.Asc }

...

Q^N = { t = (x.Parent, y.Desc) / PARENT (x) **and** Q^{N-1} (y) **and** x.Child = y.Asc }

- Let us consider the BROTHER relation, BROTHER (Name1,Name2) and the following rule : BROTHER (x) $\rightarrow$ - BROTHER (Name1 = x.Name2, Name2 = x.Name1)

The relational computation consists in evaluating the expression $Q^0 - Q^1$ where :

Q^0 = { t = (x.Name1, x.Name2) / BROTHER (x) }

Q^1 = { t = (x.Name2, x.Name1) / BROTHER (x) }

However this execution mode is not always possible. More precisely, a relational computation of a rule does not necessary return the same result than the execution mode induced by the semantics of a PCN. Thus, we need to provide syntactic characterization of set oriented rules.

It is quite easy to prove that a necessary condition for a rule to be set-oriented is to be deterministic. The deterministic property of a rule is the fact that given a transition t, the order in which the tokens of the input places of t are chosen for firing t has no effect on the stable marking of the output places of t. However this property is not sufficient information to decide whether a rule is set-oriented. Consider for instance the serial rule given in the module REDUCTION in section 2. It replaces two serial wires by one resulting wire . It is clear that such a rule is deterministic, but a set-oriented computation of the rule would not give the same result as a tuple-at a-time computation. We show a counter-example *Figure 8*. Let us consider three serial wires (A,B), (B,C) and (C,D). The set oriented computation produces two resulting wires (A,C) and (B,D), and the "tuple-at-a-time" computation produces one wire (A,D). Thus, we need stronger conditions than determinism which take into account the actions performed by a rule.

Consider a transition T. The idea is that the result of firing T for particular tokens in the input-places of T should not prevent previously firable tokens to be fired.

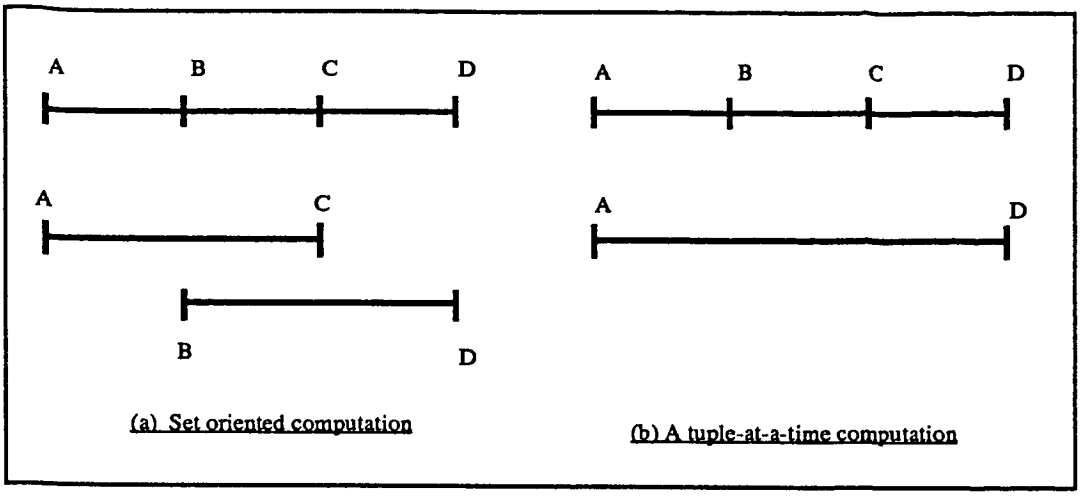

Figure 8 : serial rule computation.

We shall present two lemmas which provide sufficient conditions for a rule to be set oriented. We distinguish two cases depending on whether the actions of a rule contains deletions or insertions. The first lemma only considers an action composed of insertions.

Lemma 4.1 :

Let us consider a general production rule $R : C \rightarrow A$ where C and A are respectively a condition and an action. If the following conditions over a rule R hold :

(i) A only contains insertions,

(ii) No variable ranging over a range predicate figuring in A is universally quantified or bound by an aggregate.

Then R is set oriented.

Sketch of proof :

According to conditions (i) and (ii), the computation of the actions does not modify the truth value of an eventual close formula figuring in the condition part of the rule. For the sake of simplicity, we only consider here a rule R of the following form :

R : RP **and** SF $\rightarrow$ + P(t) where RP is a conjonction of range predicates and SF is a sub-formula of relational calculus in prenex form. Thus, SF is of the following form :

$(Q_1 r_1 \in R_1)\ (Q_2 r_2 \in R_2)\ \ldots\ (Q_n r_n \in R_n)$ (M) where $Q_i = \exists$ or $\forall$, and M is a tuple relational calculus formula whose variables are all either free or bound by an aggregate; $(Q_1 r_1 \in R_1)\ (Q_2 r_2 \in R_2)\ \ldots\ (Q_n r_n \in R_n)$ is called the prefix of SF while M is called the matrix of SF. We assume that all the variables appearing in the prefix of SF which range over the relation P are existentially quantified.

Now, let J_0 be a database state; we consider two convergent series of database states $J_1, J_2, \ldots, J_s$

and $J'_1, J'_2, ..., J'_q$ reachable by successive applications of the rule R from the initial database state J_0. Thus, we have $J_s \supset ... \supset J_1 \supset J_0$ and $J'_q \supset ... \supset J'_1 \supset J_0$.
We have to prove $J_s = J'_q$, i.e $\forall$ k (s-1 $\geq$ k $\geq$ 0) such that, $J_k \cup \{t\}$ is a reachable state from J_k by application of R then there exists p (q-1 $\geq$ p $\geq$ 0) such $J'_p \cup \{t\}$ is reachable from J'_p by application of R. Otherwise stated, for each interpretation $\Im$ of the formula in the left hand side of R, denoted $\Im(C)$, then : $(\forall k, J_k \models \Im(C)) \Rightarrow (\exists p, J'_p \models \Im(C))$ (the proof is done by induction on k)♦.

Let us give some examples to illustrate this lemma :

Examples :

First, consider the ancestor rule :

ANCESTOR (x) **and** PARENT (x) **and** x.desc = y.par → + ANCESTOR (x.asc, y.child)

This rule is detected to be set oriented by application of the lemma.

Consider now a rule containing a universally quantified variable (that is negation). We assume two relational schemas PARENTS_NAME {first-name, parent's family-name}which give for the first-name of a child the familly-name of his parents and PERSON {first-name, family-name}. Let be the following rule which give for a child a family-name :

PARENTS-NAME (x) **and** ($\forall$y $\in$ PERSON, y.first-name $\neq$ x.first-name **or**
y.name $\neq$ x.parent's family-name)
→ + PERSON (first-name = x.first-name, family-name = x.parent's family-name)

This rule is not set-oriented as shown on the following example. For instance, let us consider the following instances of the previous relation :

I (PARENTS_NAME) = {(Lucky, Bidochon), (Lucky, Dalton)}
I (PERSON) = {(Georges, Bidochon), (Averel, Dalton)}

Then the firing of the rule produces one and only one of the two following results :

I (PERSON) = { (Lucky, Bidochon)} or I (PERSON) = { (Lucky, Dalton)}

In the first lemma, we considered rules having insertions in their action part. We now address the case of rules having deletions in their action part.

Lemma 4.2 :

If the following conditions over a rule R : C → A hold :

(i) There is at most one action appearing in A,which is a deletion

(ii) No variable ranging over a relational predicate figuring in the deletion is existentially quantified or bounded by an aggregate.

(iii) The argument of the delete action - P is a variable .

Then R is set-oriented.

Sketch of proof :

According to the conditions (i) to (iii), the computation of any action does not modify the truth value of an eventual close formula figuring in the condition part of the rule. For the sake of simplicity we only consider here a rule R of the following form :

RP **and** SF $\rightarrow$ + P(t) where RP is a conjonction of range predicates and SF is a sub-formula of the relational calculus in prenex form. Thus, SF is of the following form :

$(Q_1 r_1 \in R_1)\ (Q_2 r_2 \in R_2)\ \ldots\ (Q_n r_n \in R_n)\ (M)$ where $Q_i = \exists$ or $\forall$, and M is a tuple relational calculus formula whose variables are all either free or bound by an aggregate. It is assumed that all the variables appearing in the prefix of SF which range over the relation P are universally quantified. Now, let J_0 be a database state; Two convergent series of database states $J_1, J_2, \ldots, J_s$ and $J_1, J'_1, \ldots, J'_q$ are considered, reachable by successive application of the rule R from the initial database J_0. We have $J_0 \supset J_1 \supset \ldots \supset J_s$ and $J_0 \supset J'_1 \supset \ldots \supset J'_q$. We have to prove $J_s = J'_q$, i.e $\forall k$ $(q-1 \geq k \geq 0)$ such that $J_k - \{t_0\}$ is a reachable state from J_k by application of R, there exists m $(q-1 \geq m \geq 0)$ such that $J'_m - \{t_0\}$ is reachable from J'_m by application of R. Otherwise stated, for each interpretation $\Im$ of the formula on the left hand side of R, denoted $\Im(C)$, then

$(\forall k, J_k \models \Im(C)) \Rightarrow (\exists m, J'_m \models \Im(C))$. (The proof is done by induction on k) ♦

Examples :

(1) Let P = {A,B} be a binary relation with the instance I(P) = {(a,b)} and Q = {C,D} a binary relation with the instance I(Q) = {(b,c), (b,d)}.

Then the following rule is not set-oriented:

P(x) **and** Q(y) **and** (x.2 = y.1) $\rightarrow$ + P(x), - Q(C = x.1,D = y.2)

for it leads to the two different results:

I(P) = $\varnothing$, I(Q) = {(a,c), (b,c), (b,d)} and I(P) = $\varnothing$, I(Q) = {(a,d), (b,c), (b,d)}

(2) Let B be the BROTHER = {Name1,Name2} relation with the following instance :

I(B) = {(Pierre,Paul),(Paul,Pierre)}. Then the following rule is not set oriented:

B (x) $\rightarrow$ - B (Name1 = x.Name2, Name2 = x.Name1) for it leads to the different results:

I(B) = {(Pierre,Paul)}, I(B) = {(Paul,Pierre)}.

So at compile time it is possible to determine with lemma 4.1 and lemma 4.2 if a rule can be computed with a relational algebra program.

Lemma 4.3 :

A relational request corresponds to a set-oriented rule.

Proof :

A relational request is represented by a non recursive rule with one insertion in its action part. Thus, the lemma 4.1 applies .

4.3. A Firing Meta rule

As mentionned in section 2, rules are not given by the user in a predefined order. Thus, it is the responsability of the system to decide in which order the rules will be executed. This corresponds to the choice at each step of a transition, among the set of enabled transitions. It is easy to prove that if a rule program is deterministic, (i.e has the Church-Rosser property or is confluent), then there exists a particular way to compute the fixpoint of the program for a given initial marking. This computation consists in successively firing transitions up to saturation. Thus, the problem is to decide whether a rule program is deterministic. It is also clear that if a rule program is deterministic then each rule is deterministic. Notice that the converse is not true, because the fixpoint of the rule program depends on the ordering of rule firing. In this case, the semantics of the PCN is affected by the above computation of rules. In the sequel we will only consider deterministic rules to introduce a method of computation by a *stepwise saturation* of the transitions.

In this section, we introduce a firing meta-rule which permits to proceed by *stepwise saturation* of the transitions. This meta-rule has two important consequences :

(1) it allows an efficient computation of a PCN because the non recursive transitions are fired only once (by using a relational computation).

(2) it provides a deterministic semantics of a large class of rule programs composed of deterministic rules only.

Before introducing the firing meta rule let us introduce a new definition about PCN.

Definition : A positively stable place is a non recursive place whose positive pre-transitions are stable.

Example :

Let us consider the PARENTS = {Father,Child} and G_PARENTS = {G_Father, G_Child} relations. Let be the following rule :

PARENTS (x) **and** PARENTS (y) **and** x.Child = y.Father

→ G_PARENTS (G_Father = x.Father, G_Child =y.Child)

Consider the following instances:

I(PARENTS) = {(Joe, Jules), (Jules, Juliette)}

I(G_PARENTS) = {(Joe, Juliette)}.

Then the place corresponding to G_PARENTS in the PCN is positively stable.

The meta-rule distinguishes two cases. The first one concerns a non recursive transition t. Two conditions are imposed for enabling t which ensure that (i) all the antecedent places of t have been saturated and (ii) all deletions in a place P must be performed after all the possible insertions concerning the place P. In the recursive case, the meta rule ensures that the iterative cycle of these transitions has been completed up to saturation.

Definition : The firing meta-rule consists in the two following axioms:

(i) A non recursive post-transition t of a place P is enabled iff
- all the pre-transitions of P are stable,
- if t is negative then all the post-places of t must be positively stable.

(ii) A recursive transition t of a place P is enabled iff all the non recursive pre-transitions of P are stable.

Example :

Consider the following PCN drawn on *Figure 9* : The meta-rule imposes the following firing sequence : t1 , t2 and then t3. Two observations can be made :

(1) Suppose t2 is set oriented. The action of t2 is interpreted as a set difference. Thus, the meta rule imposes by application of axiom (i) that S is completly defined before performing this set difference. Thus the meta rule is consistent with a set oriented firing of transitions.

(2) Suppose now that t2 is not set oriented. The axiom (ii) imposes that t3 can only be fired when t1 and t2 are stable. Otherwise, tokens in U could be obtained using tokens of S that would be deleted by t2. Thus, the meta rule provides a deterministic execution of a non deterministic PCN.

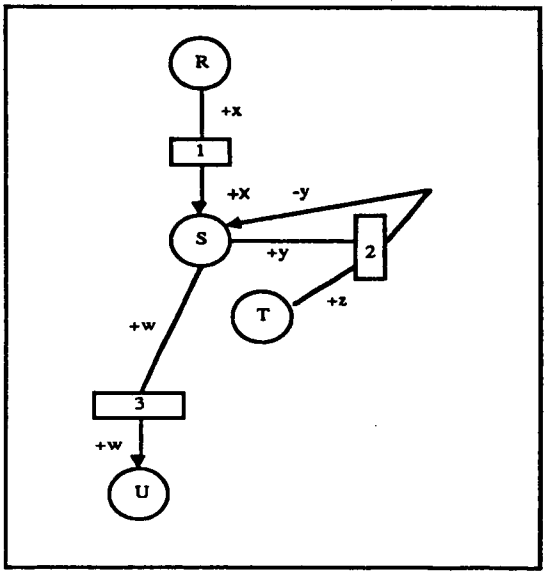

Figure 9 : application of the firing meta-rule

The meta-rule forces the transition corresponding to the query to be the fired last one in the query PCN. The meta rule mentioned above does not resolve all the possible cases of undeterminism in a rule program only composed with deterministic rules.

For instance, on the net represented on *Figure 10*, the meta rule does not induce any order between the negative transition 2 and the negative transition 3, thus the marking in the place U is non deterministic.

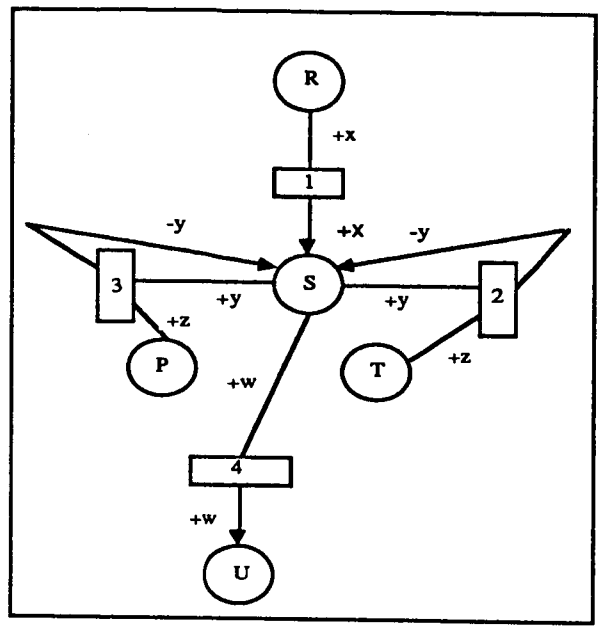

Figure 10 : PCN for which the meta rule does not provide a deterministic execution.

A solution to such a problem should be the possibility for the user to provide additional control information.

5. SET ORIENTED QUERY OPTIMIZATION

5.1 Principles

In the following, we restrict ourselves to set oriented rules. So, as seen above, all the PCN's transitions can be computed with a relational algebra program. The computation of a query can be modeled by the following steps portrayed on *figure 11* :

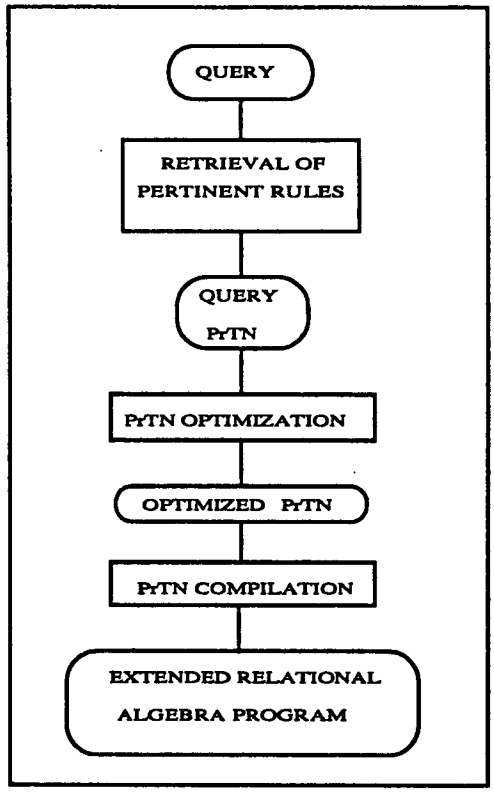

Figure 11 : Query processing steps.

The different steps are :
1) Find the relevant rules for the query
2) Use heuristics to transform the original PCN into an optimized one
3) Execute in forward chaining the PCN with extended relational algebra
We shall develop the second point in the following sections.

A query PCN models a brute force forward chaining algorithm to generate a query answer. An optimization of this process consists in three main steps :
(i) transform the initial PCN with query-modification-like technique to merge the transitions in order to benefit from a global query optimization [Krishnamurthy86],
(ii) check the satisfeasibility of the new generated predicates (using an algorithm derived from [Rosenkrantz80],[Bötcher86]) and in case of non satisfeasibility delete transitions of the PCN,
(iii) move up the selection predicates next to the base relations. This well known heuristic consists in performing selections at first. This problem needs special algorithms, specially for recursive relations, which will be detailled in the following section.

5.2 Merging transitions

The phase of compilation is now described : the compilation algorithm consists in traversing the query PCN. For each transition the corresponding query is generated. Then, these queries are grouped in a unique program. At running time, each query is analysed and optimized by the DBMS query-optimizer. In this section, we propose an optimization of this strategy. The goal of the proposed algorithm is to minimize the number of queries generated in the relational algebra program submited to the DBMS. This algorithm can be seen as an extension of query modification [Stonebraker75] . Note that this technique is semantically valid according to the semantics of the rules, specialy because the ordering of the predicates in a rule has no effect on its semantics (in contrast for instance to Prolog).

The restructuration algorithm consists in crossing the initial PCN and, if possible, for each place P, in grouping its pre-transitions with its post-transitions. The validity of this operation is submitted to the following conditions :

(i) $|Pos_t(P)| = 1$

(ii) $|Pre_t(P)| \neq 0$.

(iii) $|Pos_p(Pre_t(P))| = 1$.

(iii) $\forall$ ti $\in Pre_t(P)$, ti is positive .

The first condition says that the firing of t produce a result in only one place. The second condition says that the place P is not a base relation. The third condition imposes each anterior transition of P to have at most one posterior place. The last one says that each anterior transition of P modelizes an insertion.

Let t1,...,tn be the pre-transitions of P, F1,...,Fn be their associated formulas, t be the post-transition of P with its formula F and P1,...,Ps be the pre-places (distinct of P) of t . The restructuration algorithm is now given.

Restructuration algorithm :

Repeat
Choose a place P in the PCN
If the conditions (i), (ii), (iii) are true **then**
begin
modify all the (Pi,t) arcs into (Pi,t1),...,(Pi,tn) arcs
modify Fi into (Fi and F) and check the validity of (Fi and F)
if (Fi and F) unsatisfeasible **then** delete(ti)
delete the Place P and update the labels of the modified arcs.
end
Until all the choices have been done

The general algorithm is illustrated in the following simplified net :

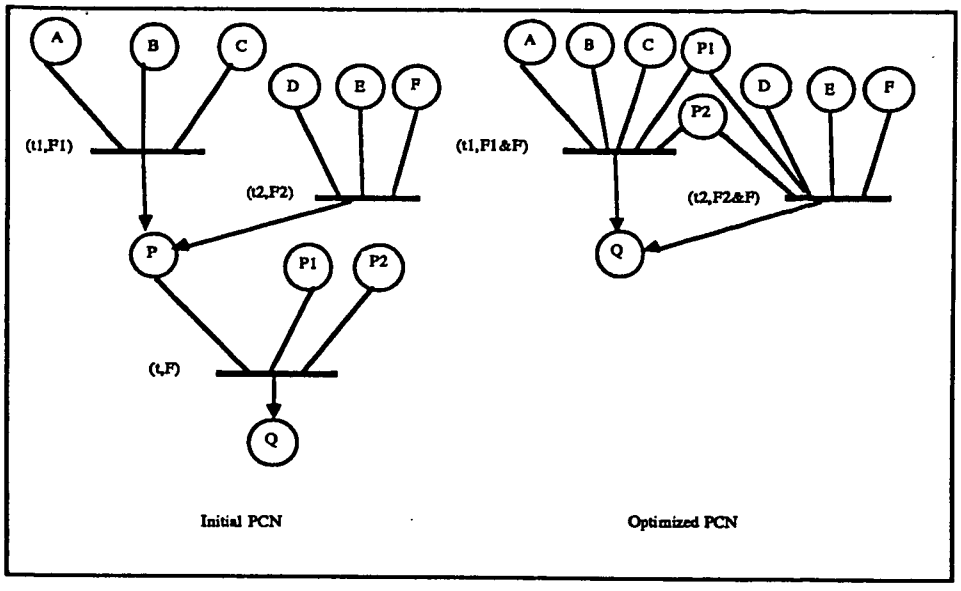

Figure 12 a : Application of the Restructuration algorithm

In the following we detail the deletion of transitions whose inscription is unsatisfeasible. For instance, an inscription such as (x.1 > y.2) **and** (x.1 ≤ 10) **and** (y.2 > 15) would be detected as unsatisfeasible by the algorithm proposed in [Bötcher86]. Let us suppose the formula F2 **and** F unsatisfeasible. The application of the delete function to the transition t2 will produce the PCN portrayed in the following picture :

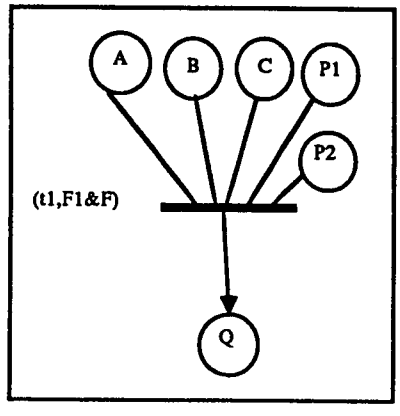

Figure 12 b : Final PCN in case of non satisfaisability of t2

5.3 Selection dominance

In this section we present a framework for transforming a PCN into an optimized one , where the selection operations have been moved up towards the places corresponding to the base

relations. To move up conditions in a PCN, we must be able to compare two transition's formulae. Let R {A1,A2,...,An} be a relational schema, f and g be two functions defined over attributes of R, and OP be a logical comparator chosen among { =, <=, >=, <>, <, > }.
A selection S on R defines a relation as follows :

S(R) = {(A1,...,An) / f (A1,..,An) OP g (A1,...,An) }

Given a relation R and two relations S1(R) and S2(R), we define the *dominance relation* using the set inclusion as follows :

Definition :

A selection S1 dominates S2 (denoted S1 >> S2) iff S2(R) $\supset$ S1(R) .

For example, (R.A1 < 10) dominates (R.A1 =< 20).
We define an atomic selection S as follows :

S(R) = {(A1,...,An) / f(A1,...,An) OP c} where c is a constant.

Then, let $\Re$ be the binary relation between two logical comparators defined as follows :

$< \Re <$ $\quad =< \Re =<$ $\quad < \Re =<$ $\quad > \Re >$ $\quad >= \Re >=$ $\quad > \Re >=$

Lemma 5.1 :

Let R be a relation (A1,...,An) and two atomic selections S1 and S2 such that S1(R) = {(A1,...,An) / f(Ai1,...,Ain) OP1 c1} and S2(R) = { (A1,...,An) / f(Ai1,...,Ain) OP2 c2} then :

(OP1 $\Re$ OP2) and (c1 OP2 c2) $\Rightarrow$ S1 >> S2.

Proof :

OP1 $\Re$ OP2 $\Rightarrow$ { (A1,...,An) / f(Ai1,...,Ain) OP2 c1} $\supset$ { (A1,...,An) / f(Ai1,...,Ain) OP1 c1}
c1 OP2 c2 $\Rightarrow$ { (A1,...,An) / f(Ai1,...,Ain) OP2 c2 } $\supset$ { (A1,...,An) / f(Ai1,...,Ain) OP2 c1}
Thus { (A1,...,An) / f(Ai1,...,Ain) OP2 c2} $\supset$ { (A1,...,An) / f(Ai1,...,Ain) OP1 c1 } $\blacklozenge$

More generally, we can prove in a similar way the following lemma :

Lemma 5.2 :

Let S1 and S2 be two selections such that S1(R1) = {(A1,...,An) / f(Ai1,...) OP1 g(Ai1,...)} and S2 (R) = { (A1,...,An) / f(Ai1,...) OP2 g(Ai1,...)}, then we have :

OP1 $\Re$ OP2 $\Rightarrow$ S1 >> S2 .

We can now compare two conjunctions of atomic selections. Let us denote (S1 **and** S2) the selection which consists in applying the **and** of two atomic selection predicates. Indeed, we demonstrate the following lemma :

Lemma 5.3 :

Let S = S1 **and** S2 **and** .... **and** Si **and** ... Sp,

S' = S'1 **and** S'2 **and** ... **and** S'j **and** ... Sq, be two selections on a relation R;

Then if there exists Si and S'j such that Si >> S'j, we have : S(R) = S(S'j(R)).

Proof:

Si >> S'j ⇒ S'j(R) ⊃ Si(R) . Therefore, S(Si(R)) is included in S(S'(R)). But S(R) = S(Si(R)) because Si appears in S .Thus S(S'j(R)) ⊃ S(R). The converse is obviously true because S'j(R) is included in R. ♦

So for a given place P, if there exists two different post-transitions composed with two selection predicates S and S' such that the hypothesis of the lemma 6.3 are verified we can move up the atomic selection S'j on all the pre-transitions of P because the stable marking in the place P will not be modified.

5.4 Moving up selections in a PCN

An algorithm which applies the previous lemmas will now be presented to move up selections in a predicate transition. We assume that any condition C[ti] in the PCN is in the following form : C[ti] = S1[ti] **and** S2[ti] **and** ... Sq[ti] **and** J1[ti] **and** ... **and** Jn[ti], where Si is a selection predicate and Ji is a join predicate. To be able to apply the previous lemmas we assume that any transition ti in the net does not have more than one positive arc leaving out. The general principle of the algorithm is to move up step by step, in backward chaining, a selection predicate and to check by using the lemma 6.3 that no tuples are lost in the place $ (which corresponds to the final result). The goal of the algorithm for a place P is to move up the atomic selection predicates figuring in the post-transitions of P onto the pre-transitions of P :

Move-up selection algorithm :

Repeat
  Choose an atomic selection predicate Sq[ti] in a post-transition ti of P .
  Move up this atomic predicate onto all the pre-transitions of P .
  If for any post-transition tj of P there exists an selection Sk[tj] such Sk[tj] >> Sq[ti] **then**
  begin
    commit the transformation on the PCN
    remove Sq[ti] from ti
    if the predicate of the transition ti is TRUE **then** merge P with the post-places of ti.
  end
  else
    undo the step 2 on the PCN
until all the choices have been done

This strategy of "do-undo" is induced by the fact that, if a place P is recursive, post-transitions of P can also be pre-transitions of P, as it will be shown using the Ancestor example.

Examples of applications :

(i) Recursive case

Consider a query PCN corresponding to the rules defining the ANCESTOR relation. The application of the move-up selection algorithm is illustrated on the *figure 13*.

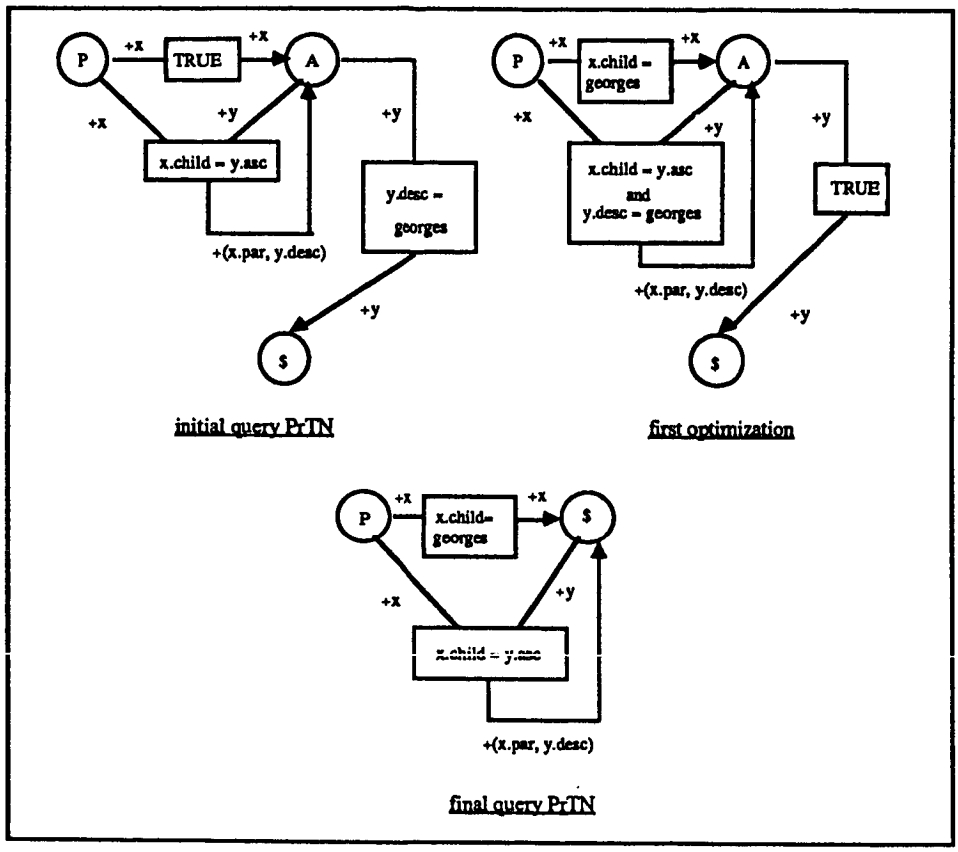

Figure 13 : the different steps of the optimization process for the Ancestor query PCN.

(ii) Non recursive case

The following figure draws the application of the move-up selection algorithm to a PCN with no recursive place.

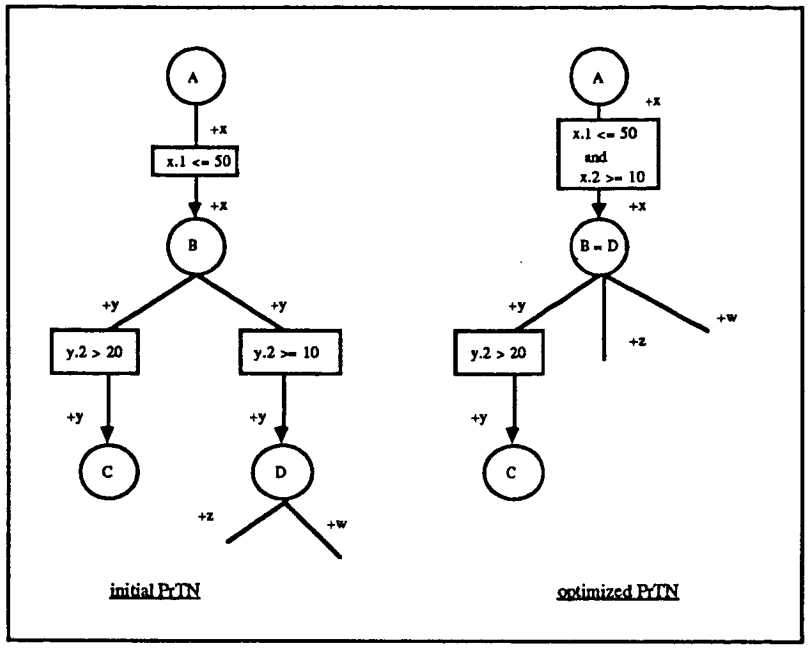

Figure 14 : application of the Move-up selection algorithm to the non recursive case

It is important to note that the given algorithm only performs a move up of the constants given in a recursive query when they do not conflict with the join conditions specified in the query. In the other case, a first transformation of the PCN, using the functional approach described in [Gardarin86], is necessary before applying the above algorithm. Finally, let us point out that the proposed algorithm applies for any kind of restriction and join predicates (with < or > comparators), including functions and mutually recursive relations. It is also easy to implement.

6. QUALITATIVE ANALYSIS

We distinguish two aspects : the generality of both rule language and execution model and the qualitative analysis of the query processing algorithms.

6.1. Generality

The first point is the generality of the proposed production rule language. In fact, RDL1 presents several features.

(1) It supports negation in the condition part of a rule.

(2) It supports complete database update capabilities (in particular, deletions are allowed in the conclusion part of a rule).

(3) Terms of the calculus are typed and we named attributes are used. This makes the writing of rules more concise and easier when the relational predicates have several arguments. Also type checking is possible at compile time when a rule is declared.

(4) The ordering of rules in a rule program is irrelevant as well as the ordering of actions in the action part of a rule.

(5) The language has an operational semantics by opposition to a purely declarative one. This presents, howewer, the advantage of knowing how a rule is used during the program computation. For instance the phrase "All teenagers love Madonna" can be represented by one single logic formula : $(\forall x) \neg$ Teenager (x) v Loves (x, Madonna). This formula can be used either to deduce that someone is not a teenager or that someone loves Madonna. Thus, the same phrase leads to two production rules in our language.

(6) The notion of rule module offers a layered constructive methodology for defining derived relations.

The model of PCN has the following properties :

(1) Descriptive power : PCN have the same capabilities than Predicate Connection Graphs [McKay81] or system graphs [Lozinskii85] for representing inter-rule connections and predicate-rule connections. Furthermore, PCN allows to represent database updates, that is insertions and deletions into the base and derived relations.

(2) PCN offers a good support to capture basic heuristics for query optimization. In particular, the labeling of arcs permit to know how the attribute values of the tokens contained in the input places of a transition t contribute to the pattern of the tokens produced in the output places of t. This information is fundamentally used by the algorithm described in section 5.4. In comparison, System Graphs [Lozinskii85] represent the same kind of information. Rule/Goal Graphs [Ullman85] go further because they capture in addition different control strategies (capture rules) and this permits to design algorithms which can make a choice among these strategies. Rule/Goal Graphs have been used for example in [Bancilhon86a] as a basic tool for recursive query optimization.

(3) Finally, PCN represents a set of rules into a compact compiled form which can be efficiently maintened by incremental updates. Also, PCN constitutes a storage structure which pre-compiles much of the work performed during query processing. This can be opposed to execution models such as And/Or Graphs which are dynamically generated during the proof process.

In fact, PCN appear as a good tool for representing the execution of a set of production rules. The procedurality of the rule language carries along the limitation of PCN. For instance, PCN cannot be used as a suitable tool for rewriting recursive rules as in [Bancilhon86a] because a PCN induces a specific algorithmic way of computation of the rules. Therefore, rewriting rules comes to transform an algorithm into another one which cannot easily be done in the general case. This is why a functional method has been designed in [Gardarin86] to optimize a larger class of recursive rules than those handled by the selection pushing algorithm of section 5.4.

6.2. Query processing algorithms

In this section we analyze the algorithms proposed for optimizing the execution of queries only involving set oriented rules. We follow a query optimization approach of rule programs. That is, we assume an underlying simple strategy which is a semi-naïve evaluation and provide algorithms which make the evaluation of a rule program more efficient. In [Bancilhon86b] basic properties of recursive query processing strategies have been isolated. We relate our method to these properties. First, our method is compiled. During the compilation phase, rules are accessed and a Query PCN is built in main memory. Then a Relational Algebra Program (RAP) is generated. The second phase is the execution of this program by the DBMS. Second, our strategy is iterative. This is clearly shown by the PCN evaluator procedure of section 3.2. Finally, our strategy proceeds in a bottom-up way. This means that we start from the base relations and produce tuples into derived relations until the result of the query is generated.

The two algorithms presented in this paper are now analyzed.

Our selection move up algorithm is an extension of the one described in [Aho79]. We understand that our algorithm works similarly to the algorithm described in [Kifer85]. Indeed, we can make an analogy between the pushing of filters through nodes in a System Graph and our pushing up selection method. The algorithm of [Aho79] is extended as described below.

(1) In a similar way than [Kifer85], our algorithm pushes up a selection through a recursive place defined by more than one recursive rule. For instance, consider the following rules over the relations P = {B,C}, A = {D,E}, R = {F,G}:

P (x) →+ A (x)

P (x) AND A (y) AND x.C = y.D →+ A (D = x.B, E = y.E)

R (x) AND A (y) AND x.G = y.D →+ A (D = x.F, E = y.E)

Let Q be the query {(x.D, x.E) / A (x) and x.E = 'a'}. The move-up selection algorithm rewrites the rules into the optimized rule program :

P (x) AND x.B = 'a' →+ A (x)

P (x) AND A (y) AND x.C = y.D →+ A (D = x.B, E = y.E)

R (x) AND A (y) AND x.G = y.D →+ A (D = x.F, E = y.E)

The following picture illustrates the rewriting of the rules on the PCN :

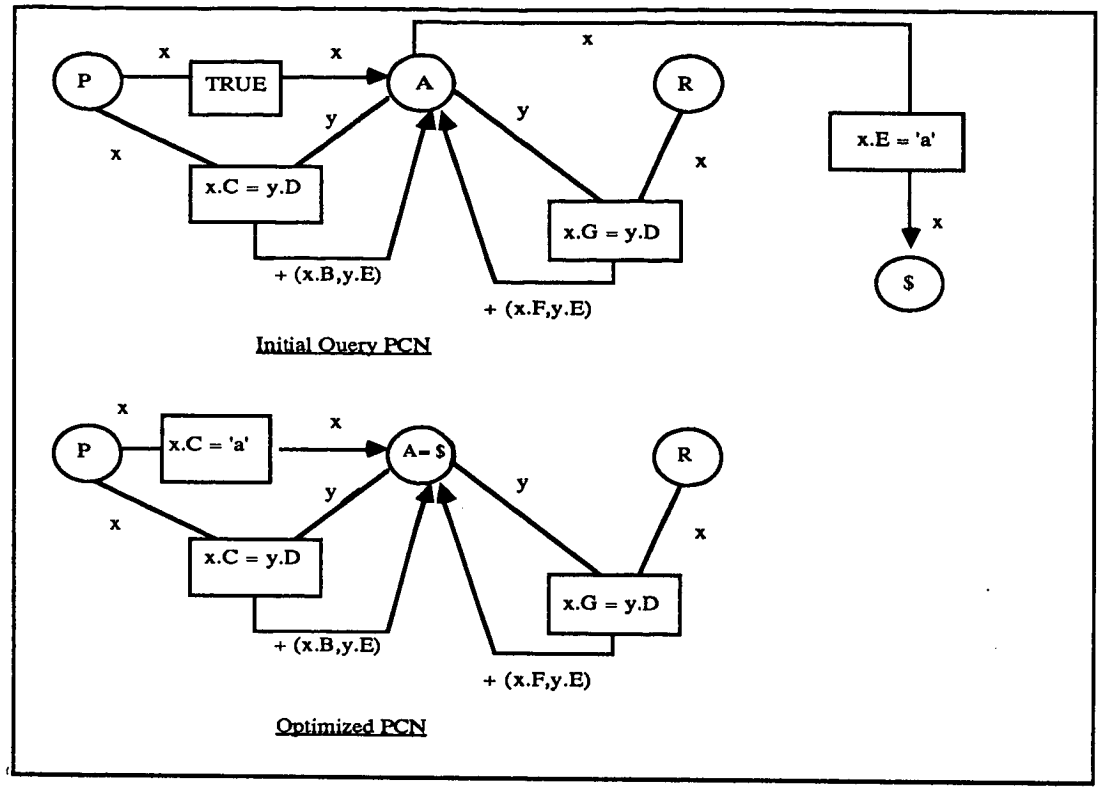

Figure 15

(2) One difference with the Kifer-Lozinskii algorithm is that our algorithm allows to move up selection of the general form Attribute θ Attribute where θ is a logical comparator. For instance, consider the following rules over the relational schemas P {B, C, D} and A {E, F, G}.

P (x) →+ A (x)

P (x) AND A (y) AND x.D = y.G →+ A (E = y.E, F = y.F, G = x.D)

Let Q be the query {(x.E, x.F, x.G) / A (x) and x.E < x.F}. The rules are rewritten into the optimized program :

P (x) AND x.B < x.C →+ A (x)

P (x) AND A (y) AND x.D = y.G →+ A (E = y.E, F = y.F, G = x.D)

The two steps of the optimization algorithm are presented on the *Figure 16* :

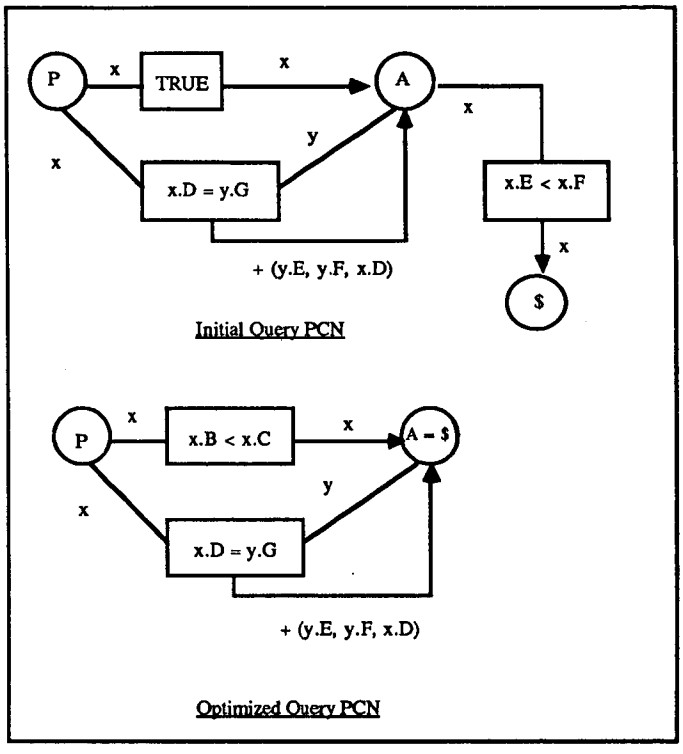

Figure 16

The second optimization is performed by the restructuration algorithm of section 5.2. This algorithm presents two main advantages. First, in case of recursive rules, it allows the push up algorithm to success. For example, consider the following rules over the relational schemas P {B, C}, A {D, E}, R {H, J} and Q {F, G}:

P (x) →+ A (x)

P (x) AND A (y) AND x.C = y.D →+ R (H = x.B,J = y.E)

R (x) AND Q (y) AND x.H = y.F →+ A (x)

Let Q be the query {(x.D, x.E) / A (x) and x.D = 'a'}. The rules are first rewritten into the optimized program :

P (x) →+ A (x)

P (x) AND A (y) AND Q (z) AND x.C = y.D AND x.C = z.F →+ A (D = x.B,E = y.E)

Then, the move-up algorithm applies and rewrites the program into :

P (x) AND x.B = 'a' →+ A (x)

P (x) AND A (y) AND Q (z) AND x.C = y.D AND x.C = z.F →+ A (D = x.B,E = y.E)

The different steps of the optimization process are portrayed on the *Figure 17*

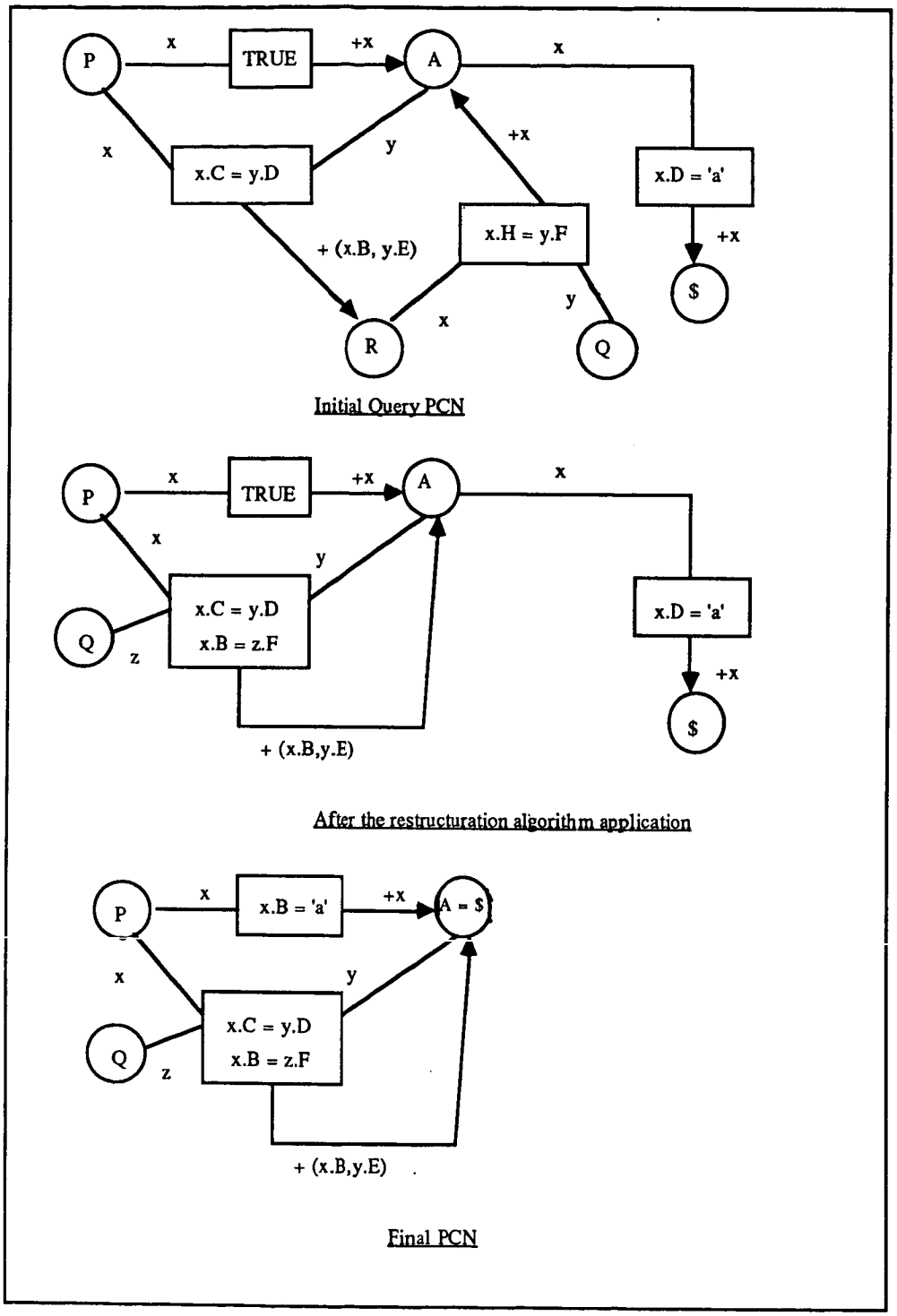

Figure 17

In the non recursive case, the restructuration algorithm permits to group several instructions of the RAP generated during the phase of compilation into a single instruction. The fact that the algebraic transformation is done on the PCN and not on a relational tree presents three improvements :

First, the recursive rules and the rules having more than one insertion or deletion are not easily representable on a relational tree. Second, the complexity of the algorithm is leather than the one of a classical optimizer running on algebraic trees, because the PCN model is more compact than algebraic trees. Third, it allows to manage interactivity with the user with a sufficient level of abstraction (i.e in terms of rules and not in terms of relational expressions).
As a consequence, global optimization techniques, which have been shown to be an important issue in deductive database, can be applied for deciding in which order joins involved in the transitions of the net must be performed. A recent work describing global optimization techniques is reported in [Krishnamurthy86].

Finally, the satisfeasibility check of the conjunction of transition's inscriptions generated by the restructuration algorithm provides an efficient optimization by reducing the number of enabled transitions of the PCN.

7. CONCLUSION

In this paper, we introduced a general compilation technique for transforming production rules program into a new execution model called Productions Compilation Network (PCN). This model is shown to be very general and permits to represent general database updates. Several query optimization techniques have been proposed for a class of programs composed of set-oriented rules. The query processing method follows a compilation approach. During the compilation phase, a PCN is built in main memory. A first optimization algorithm restructurates the net and performs some transition mergings. The resulting net is then modified by a move-up selection algorithm into a net which does not generate useless intermediate results for computing a query. A qualitative analysis compares these two algorithms with other related works in recursive query processing. A specific meta rule was presented for ordering the computation of rules and generating a Relational Algebra Program to the DBMS. The second phase of query processing is achieved through a semi naïve evaluation of the generated RAP.

A first implementation of the method has shown the adequacy of the execution model. Also, it pointed out some research perspectives for improving both usability and performances of the system. Performances pose two main problems : (i) to provide a better syntaxic characterization of set-oriented rules, (ii) to design specialized operators for dealing with non set-oriented rules [Pucheral86] and (iii) to provide global optimization techniques for RAPs. The usability of the sytem could be improved in two ways. A language for specifying different control strategies has to be designed. Such a language would allow the user to specify a partial ordering of rules when our proposed meta-rule fails. Another point is to improve the interactivity with the user through rule

editors and explanation tools. In conclusion, we believe that the techniques described in this paper constitute a basic tool for implementing general production rule languages on top of DBMS.

Acknowledgements :

We wish to thank our project manager, G. Gardarin, for numerous fruitful discussions we had with him and for encouraging us in this work. Special thanks are due to S.Abiteboul for many stimulating discussion on the rule language. We are also grateful to F. Bancilhon for suggesting many corrections and improvements of this paper.

References :

[Abiteboul87] S. Abiteboul, V. Vianu : "*A Transaction Language Complete for Database Update and Specification.*" to appear in ACM PODS, 1987.

[Aho79] A. Aho, J. Ullman : "*Universality of Data Retrieval Languages*" Proc. 6th ACM Symposium on Principles of Programming Languages.

[Bancilhon86a] F. Bancilhon, D. Maier, Y. Sagiv, J. Ullman : *"Magic Sets and Other Strange Ways to Implement Logic Programs"*, Proc. 5th ACM SIGMOD-SIGACT Symposium on principles of Database Systems, 1986.

[Bancilhon86b] F. Bancilhon, R. Ramakrishnan :*"An Amateur's Introduction to Recursive Query Processing Strategies."* Proc of ACM-SIGMOD , Washington, D.C.

[Bocca86] J. Bocca : *" On the Evaluation Strategy of EDUCE."* Proc of ACM-SIGMOD 1986, Washington, D.C.

[Bötcher86] S. Bötcher, M. Jarke, J.W. Schmidt : *"Adaptative Predicate Managers in Database System."* Proc of 12th VLDB 1986, Kyoto.

[Brownston85] L. Brownston, R. Farrell, E. Kant, N. Martin :*"Programming Expert Systems in OPS5 : An Introduction to Rule-Based Programming"*. Ed. Addison-Wesley.

[Bruno86] G. Bruno, A. Elia : *"Operational Specification of Process Control Systems : Execution of PROT Nets using OPS5."* Proc of World IFIP 1986, Dublin.

[Ceri86] S. Ceri, G. Gottlob, L. Lavazza : *"Translation and Optimization of Logic Queries : The Algebraic Approach."* Proc of 12th VLDB 1986, Kyoto.

[Codd71] E.F. Codd : *" A Data Base Sublanguage founded on the relational calculus."* Proc of ACM-SIGFIDET 71.

[Demolombe84] R. Demolombe : *"Syntactical Characterization of a Subset of Domain Independant formulas."* Internal Report, ONERA CERT, Toulouse.

[Gardarin85] G. Gardarin, C. de Maindreville, D. Mermet, E.Simon : *"Extending a Relational DBMS towards a KBMS : A First Approach ."* Proc. Workshop on Knowledge Base Management Systems, Ed. J. Schmidt, C. Thanos, Crete, june 1985.

[Gardarin86] G. Gardarin,C.de Maindreville:*"Evaluation of Database Recursive Logic Programs as Recurrent Function Series."* Proc of ACM-SIGMOD, 1986, Washington, D.C.

[Genrich78] H.J. Genrich, K. Lautenbach : *" Facts in Place : Transition nets."* Lecture notes in Computer Science, 64, Springer-Verlag.

[Genrich81] H.J. Genrich, K. Lautenbach : *"System Modelling with High-Level Petri Nets."* Theoretic.Computer.Science, N°13.

[Giordana85] A. Giordana, L. Saitta : *"Modeling Production Rules by Means of Predicate Transition Networks."* Information Sciences Journal, North Holland Ed. Vol.35,N°1.

[Kifer85] M. Kifer, E. L. Lozinskii : *"Query Optimization in Logic Databases"*, Technical Report, SUNY at Stonybrook, June 85.

[Kowalski72] R. Kowalski : *"And-or Graphs, Theorem-proving Graphs and Bi-directional Search."* In Machine Intelligence, N°7.

[Krishnamurthy86] R. Krishnamurthy, H. Boral, C. Zaniolo : *" Optimization of Non recursive Queries."* Proc of 12th VLDB 1986, Kyoto.

[Lozinskii85] E.L. Lozinskii : *"Evaluating Queries in Deductive Databases by Generating."* Proc of 11th IJCAI.

[Lozinskii86] E.L. Lozinskii : *"A Problem-Oriented Inferential Database System."* ACM Trans. Database Systems, Vol. 11, N° 3, Sept. 1986.

[McKay81] D.P. McKay, S.C. Shapiro : *"Using Active Connection Graphs for Reasonning with Recursive Rules."* Proc of 7th IJCAI, 1981.

[Peterson81] J.L. Peterson : *"Petri Net Theory and the Modelling of systems."* Prentice-Hall.

[Pucheral86] P. Pucheral : *" Recursive Queries Evaluation Using Graph Techniques."* To appear in Modeles et Bases de Données.

[Rosenkrantz80] D.J. Rosenkrantz, H.B.Hunt : *" Processing Conjunctive Predicates and Queries"* Proc of 6th VLDB 1980, Montreal.

[Simon86] E. Simon : *"RDL1 : A Production Rules Language for Deductive Databases."* In preparation.

[Stonebraker75] M. Stonebraker : *"Implementation of Integrity Constraints and Views by Query Modification."* Proc of ACM-SIGMOD 1975, San Jose.

[Ullman85] J.D. Ullman : *"Implementation of Logical Query Languages for Database."* TODS, Vol 10, N°3, 1985.

[Van Gelder86] A. Van Gelder : *"A Message Passing Framework for Logical Query Evaluation."* Proc of ACM-SIGMOD, 1986, Washington, D.C.

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