Modelling Non Deterministic Queries and Updates In Deductive Databases[†]

Christophe de Maindreville and Eric Simon

INRIA Rocquencourt - BP 105
78153 Le Chesnay Cedex, France

Abstract

In this paper, we propose a new formalism to model and implement general, rule-based languages for querying or updating deductive databases. We consider as a target language a production rule language for databases that we introduced in previous papers, namely RDL1. This language can be seen as an extension of a logic-based language for databases to support updates in the head of rules. The model we introduce, named Production Compilation Network (PCN), is derived from Petri-Net based models. A PCN models the two aspects of a rule program: the static aspect which consists of the relationships between the rules and the database predicates and the dynamic aspect which describes the semantics of a rule program. The PCN are shown to have the following features: (i) to provide a formal framework to describe general computation strategies and query optimization algorithms in a clear and concise manner and (ii) to model in a unified way queries and updates in a deductive database context.

1. INTRODUCTION

A deductive database consists of a set of base relations called *extensional database* (EDB), and a set of virtual relations defined using *rules*, called *intentional database* (IDB). In a deductive database, the query language generally consists of a rule based language, named DATALOG, where rules are function-free definite horn clauses. Several researchers studied extensions of DATALOG in order to increase the power of the rule language. The introduction of negative literals in the body of a clause leads to DATALOG^{neg}. Another extension is DATALOG^{set} which augments the terms with sets.

In previous papers [Simon87, Maindreville88], we proposed an original approach to deductive databases based on a production rule language, named RDL1. A production rule, in our language, consists of a conditional part which is a relational calculus expression and a consequent part which is a sequence of insertions and deletions of tuples in database derived or base relations. The main features of this language are the following. First, it captures the operational (or declarative) semantics of stratified DATALOG^{neg} programs but offers more generality and more computational power than a DATALOG^{neg} language since a sequence of database updates can be expressed within a single rule. In fact, RDL1 can be seen as an extension of DATALOG^{neg} to support updates in the head of rules. The second feature is that the operational semantics of the language provides a uniform and clean compromise between declarativity and procedurality. Indeed, the procedural aspects of RDL1 stand upon (i) the use of updates and (ii) the use of an implicit ordering of rules (stratification-like) combined with an explicit (user given) partial ordering of rules. Notice that stratification in logic programs already introduces procedural control in the language. As a consequence of the previous points, RDL1 is not only a query language but is also an update language that can be used to specify complex triggers.

One of the main problems encountered in deductive databases is how to evaluate a rule program, that is a query. Most query processing algorithms for deductive databases strongly use graph models. Many such models have been proposed in the litterature. Predicate Connection Graphs (PCG) and Active Connection Graphs (ACG) [McKay81] are used to model DATALOG programs. The PCG connects the literals that can be unified, i.e., predicate occurrences in the rule body are connected to their occurrences in the heads of the rules. The PCG is static and displays only the structure aspect of the logic program. It allows to retrieve all the formulae that unify with a given formula. ACG provide the dynamic aspect and display the binding propagations through the flow of control. ACGs detect recursion through predicates that have the same bindings. However, both PCG and ACG are not suitable in a database context because they do not make a distinction between base

[†]Research is supported in part by the ESPRIT Project ISIDE (under contract P1133)

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the VLDB copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Very Large Data Base Endowment. To copy otherwise, or to republish, requires a fee and/or special permission from the Endowment.

(i.e., EDB) and derived relations (i.e., IDB). System Graphs (SG) [Lozinskii86] have been proposed as an extension of ACG to compensate for these limitations and to support function symbols. Rule/Goal Graphs (RGG) [Ullman85] are used to statically represent DATALOG programs with function symbols. An RGG is an adorned AND/OR graph where each derived predicate is represented by a number of relation-nodes corresponding to the different possible binding patterns for this predicate. Thus, an RGG can display the propagation of constants but its size can grow exponentially in the maximum number of arguments of a derived predicate. Capture rules [Ullman85] in RGG seem limited to a semi-naive evaluation strategy. Extensions of RGG have been shown to be well suited for studying recursive query optimization algorithms [Bancilhon86b, Beeri87] and they have been enhanced by so called information passing graphs [VanGelder86] in order to restrict the computation process to relevant facts. Nevertheless, both SG and RGG suffer from several limitations. First, they are highly dependent on the query processing algorithms for which they have been designed. They can hardly be used to model general query evaluation strategies (e.g., bottom-up, top-down, ...) as well as the flow of control of several different query optimization algorithms (e.g., different constant move up algorithms, ...). The second limitation is that no execution model has been developed to model more general query languages such as DATALOG^{neg} or updates and triggers' languages, as well as their associated optimization strategies in a deductive database context. For instance, the above models are bound to deal only with query languages such as DATALOG.

Recently, a Petri-Net based model (colored nets) was proposed in [Aly87] to alleviate the first limitation and to model the behavior of DATALOG programs using different strategies and algorithms. In this paper, we propose another Petri-Net based model, called Production Compilation Network (PCN). This model is derived from Predicate Transition Nets (PrTN) defined in [Genrich86]. PCN can model the behavior of any RDL1 program, (and a fortiori any DATALOG^{neg} program), and thus provide a generalization of the approach presented in [Aly87]. In particular, PCN provide a uniform execution model for implementing updates and triggers in a deductive database. Another important feature is that, through a particular language of control over the PCN, it is possible to manage explicit ordering between rules. Therefore, PCN may capture procedural control over the rules.

Apart from this introduction, the paper is organized as follows. Section 2 presents the structural and the dynamic aspect of the PCN formalism. Section 3 introduces a basic language over the PCN which describes in a clear and concise way the different computation strategies of a rule program. This leads to the notion of annotated PCN. Section 4 discusses the problem of generating an efficient annotated PCN and presents a general overview of our query processing strategy as it is currently being implemented. Finally, section 5 is the conclusion.

2. THE PRODUCTION COMPILATION NETWORK MODEL

2.1 Preliminaries

We first recall basic database notions. 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 mapping from R into dom $(A_1) \cup$ dom $(A_2) \cup .... \cup$ dom (A_n) such that for each i in $\{1, ..., n\}$ $c_i \in$ dom (A_i). 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 (also called a database) over a database schema S is a total function from S such that for each R in S, I(R) is an instance over R.

An RDL1 production system (or an RDL1 program) is composed of a set of *if-then* rules called *productions* that make up the rule base, and a relational database called the *fact database.*

The left-hand side of a production, (corresponding to the if part of the rule), is a range restricted expression of the tuple relational calculus [Codd71, Ullman82] which consists of the conjunction of a *range formula* and a *sub-formula*. A range formula is a conjunction of positive (resp. negative) range predicates which indicate that a tuple variable ranges (resp. does not range) over a relation. A positive range predicate is denoted R(x) while a negative range predicate is denoted $\neg$ R (x) where x is a tuple variable and R is a relation name. A sub-formula is a condition over the variables that appear in the range formula part.

The right-hand side of a production, (corresponding to the then part of the rule), is a set of actions. There are two *elementary actions*, denoted "+" and "-". The update action "+"

takes a ground fact (i.e., a constant tuple) and maps a database state into another state which contains this fact. Thus, the action "+" inserts a tuple into a derived relation. For instance, + Parent (sam, jeremie) inserts the tuple (sam, jeremie) into the Parent relation. On the contrary, the action "-" takes a fact and delete it from a relation. In a *parametrized action*, variables are allowed in the argument of the action (e.g., + Parent (Father = x, Child = y)). For safety reasons, all the variables that appear in the action part of a rule also appear positively in the condition part.

Traditionally,a production system *interpreter* [Brownston85] is the underlying mechanism that determines the set of satisfied productions and controls the execution of the production rule program. The interpreter executes a production rule program by performing the following *recognize-act* cycle :

- **Match** : in this first phase, the left hand side of all productions are matched against the database. As a result, a conflict set is obtained which consists of instantiations of all satisfied productions. An instantiation of a production is a rule where all free variables are substituted to constant tuples.
- **Conflict-resolution** : In this second phase, one of the productions in the conflict set is chosen for execution. If the conflict set is empty, the interpreter halts.
- Act : In this last phase, the actions of the production selected in the previous phase are executed. These actions may change the content of the database. At the end of this phase, the first phase is executed to restart the cycle.

This cyclic interpretation scheme forms the basic control structure for RDL1 production rule programs. In [Simon88, Simon88b], we provided the formal semantics of RDL1 which is in the spirit of Kripke semantics for modal logic [Kripke63]. As a result, we show that this semantics captures the declarative semantics of a stratified DATALOG^{neg} language but offers more generality and more computational power than a DATALOG^{neg} language. Indeed, RDL1 can be seen as an attempt to extend DATALOG^{neg} to support database updates. A notable feature of the language is that it enables to specify non-deterministic updates or queries. Recently a very attractive formal approach has been proposed in [Abiteboul87a] to study database update languages. We believe that the basic techniques developed in this paper can also be used to implement their languages.

In the next sub-section we present the formal graphical tool called Production Compilation Network (PCN) to model the behavior of RDL1 programs. This model has two aspects : the structure aspect and the execution aspect.

2.2. The structure of a Production Compilation Network

The PCN model is a Petri Net based model. More precisely, it derives from Predicate Transition Nets (PrTN). A formal definition of PrTN can be found in [Genrich86]. It is convenient to recall that a PrTN consists of [Genrich 86] :

(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 their extensions, and transitions represent classes of elementary changes of extensions.

(2) a *labeling* of the arcs with formal sums (noted "+") 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. Formulae built up in Σ can be used as *inscriptions* inside some transitions.

(4) A function K, from the set of places 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 structural aspect of a PCN represents the relationships between rules and relational predicates as specified by a rule program. The following associations can be made between rules and the PCN structure. We represent each rule by a transition and each relational predicate involved in the rule by a place. The relational predicates that occur in the condition part of a rule are input places to the transition representing the rule and the relational predicates that occur in the action part of the rule are the output places of the transition. The condition itself of a rule is represented in the transition's inscription. More formally, we are able to set up an isomorphism between an RDL1 program and a PCN structure which is summarized by the table below.

RDL1	PCN
RULE	TRANSITION T
RELATIONAL PREDICATE P	PLACE P
Free tuple variables $X_1,\dots, X_n$ ranging over P	ARC (P, T) labelled by : $+X_1 + \dots +X_n$
Bound variables ranging over P or free variables ranging over $\neg$P	ARC (P,T) labelled by : (.)
ACTION + P (t)	ARC (T, P) labelled by + t
ACTION - P (t)	ARC (T, P) labelled by - t
Condition and negative range predicates	Transition's inscription

We now give a first example of a PCN built from a set of rules. Let us consider the following rule module ANCESTOR which defines a derived relation ANCESTOR as the transitive closure of the base relation PARENT {Par, Child}.

```
MODULE  : ANCESTOR ;
target  : ANCESTOR {Asc; Desc};
rules   :
PARENT (x) → + ANCESTOR (x) ;
PARENT (x) and ANCESTOR (y) and x.Child = y.Asc
  → + ANCESTOR (Asc = x.Par, Desc = y.Desc);
end.
```

In this example, x and y denote tuple variables ranging respectively over the PARENT and ANCESTOR relations.

Two relational predicate names appear in the rules. They lead to the places PARENT and ANCESTOR (represented by the circles P and A on the net). The first rule is modelled by the transition T1 whose inscription is the formula TRUE. The second rule is modelled by the transition T2, whose inscription is the formula "x.Child = y.Asc". On the net, transitions are represented by boxes. The arcs outgoing from PARENT are labelled by + (x.Par, x.Child). The arc outgoing from ANCESTOR is labelled by + (y.Asc., y.Desc.). Finally, the arc going from T2 to ANCESTOR is labelled by + (x.Par, y.Desc.). To simplify the graphical representation of the PCN, tuples are replaced by tuple variables in the labels of the input arcs of the transitions.

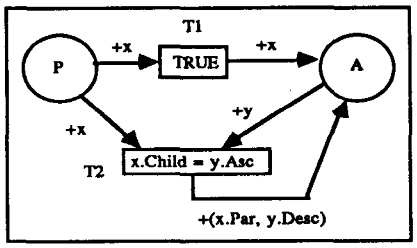

Figure 2. 2 : *PCN for the Ancestor rules*

The second example is the PCN (figure 2.3) associated to the rule Module WINGS. The base relations are PENGUINS, CROWS having for schema {Name, Color, Sex, Country, Predator}.

```
MODULE : WINGS  ;
target :   FLY {Name} ;
rules  :
PENGUINS (x) → + BIRDS (Name = x.Name) ;
CROWS (x)  → + BIRDS (Name = x.Name) ;
BIRDS (x) →  + FLY (Name = x.Name) ;
PENGUINS (x)  → - FLY (Name = x.Name) ;
end.
```

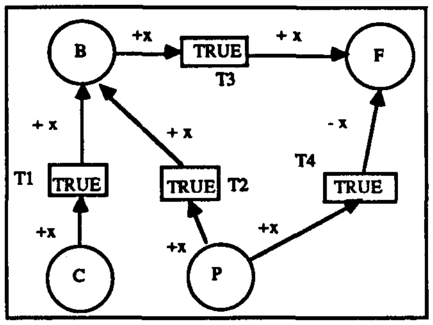

Figure 2.3 : *the PCN for the WINGS Module*

A query is a production rule represented as a single transition, noted T$, several input places and a single output place, noted $, representing the result of the query. Thus, a query can be represented as the combination of two PCNs. The first one represents all the rules needed to define the content of the input places of the query. The second net represents the query itself. The resulting PCN is called a *query PCN*. An example of a query PCN is given figure 2.4.

The query expressed in the SQL language is :

```
SELECT Asc, Desc FROM ANCESTOR
WHERE Desc = Georges;
```

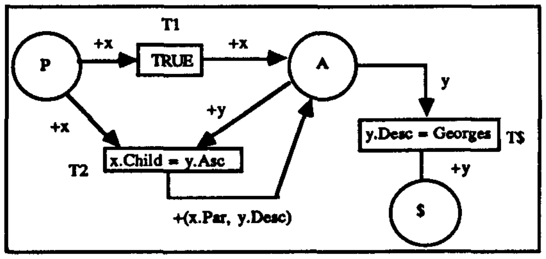

Figure 2.4 : *Query PCN*

2.3. Semantics of a PCN

In this section, we define the semantics of a PCN. The dynamic aspect consists of the semantics associated to the firing of the PCN transitions. This semantics corresponds precisely to the semantics of RDL1 as shown in [Simon88, Maindreville88].

A *marking* is a distribution of tokens over the places of a PCN. The token is a basic concept in all Petri Net based models. In a PCN, a token represents a database tuple. Traditionally, a Petri Net based model executes by firing transitions. A transition can be fired if it is enabled.

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 occurring in the input places of T have values satisfying the transition inscription.
(iii)The marking of at least one output place of T is modified by the transition firing.

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 a duplication free PCN. 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 only produce already existing tokens is not enabled. This corresponds to assign a fixpoint semantics to the "$\rightarrow$" symbol in the rule language.

At a given time, several transitions can be enabled. A *transition's occurrence* is a couple (T, L) where T is a transition name and where L is a list of variable substitutions of the form $<x_1/a_1, ..., x_n/a_n>$ where the x_i are all the free variables figuring on the labels of the input arcs of transition T and the a_i are tokens issued from the input places of transition T. In the remainder of this paper L is called a *substitution list*. The set of all transition's occurrences, for a given marking of the net, is called the *transition's conflict set*. Note that for a single transition T, different substitution lists can be used to fire the transition. Each one of these lists leads to an element (T, L_i) in the transition's conflict set. We note Rel (T) (for : relevant set of tokens for T), the set of all the substitution lists that appear in the transition's conflict set together with the transition T.

Example :

Consider the transition T1 of the PCN represented in Figure 2.2. Assume that the marking of the place A is empty and that the marking of P is {(Bill, Paul), (Paul, Sam)}. Then the transition's conflict set is the set : {(T1, <x / (Bill, Paul)>), (T1, <x / (Paul, Sam)>)} and Rel (T1) = {<x / (Bill, Paul)>, <x / (Paul, Sam)>}. Note that by definition, Rel (T1) only contains tokens that satisfy the inscription of T1.

When a transition T is enabled, it can be *fired*. Firing a transition's occurrence (T, L) produces several actions. First, it duplicates from each input place P of T a number of tokens equal to the number of positive symbols labeling the arcs (P, T). Indeed, we use conservative nets which means that firing a transition T will only change the state of the output places of T. Then, for each output place P of T, let $+ t_1 + ... + t_n$ be the positive label of (T, P) and let $- s_1 - ... - s_q$ be the negative label of (T, P). Given the substitution list L in Rel (T), we define $L(t_i)$ as the result of applying the variable substitutions of L to the variables of t_i. Two sets of tokens S_+ and S_- are then defined as follows:

$$S_+ = \{L(t_i)\} \text{ for } i = 1 \text{ to } n;$$
$$S_- = \{L(s_i)\} \text{ for } i = 1 \text{ to } q;$$

Let M(P) be the marking of P; the firing of (T, L) produces a new marking in P:

$$M'(P) = (M(P) \cup (S_+ - S_-)) - (S_- - S_+).$$

This result is portrayed in the figure 2.5. The dashed part represents the marking M'(P).

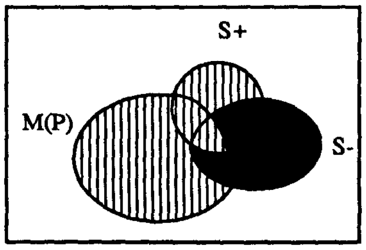

Figure 2.5 : *result of the firing of a transition's occurrence.*

With these definitions, a multi-labelled arc is interpreted as *an atomic database update*. This property has several consequences. First, it ensures that the order of insertions and deletions in the action part of a rule is *irrelevant*. Second it *nullifies* an action which inserts and deletes the same constant tuple. For instance, the instantiated label of an arc (P, T): $L(t) = + (a, a) - (a, a) - (a, a) \equiv - (a, a) + (a, a) - (a, a)$ is a null action over P. However, $L(t) = + (a, b) - (a, c) \equiv - (a, c) + (a,b)$ deletes the tuple (a, c) and adds the tuple (a, b).

Example :

Consider the following PCN :

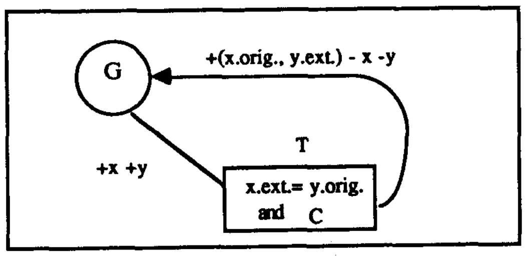

Figure 2.6

On this net, the relation G {Ext., Origin} stores oriented arcs in a graph. The transition T computes the reduction of the graph by replacing two serial edges by one resulting edge. The condition C is : $(\forall z \in G)[(z = y) \vee ((z.Origin \neq x.Ext) \wedge (z.Ext \neq x.Ext))]$. It expresses that x.Ext. is a single node in G.

Let M_0 (G) be the following initial marking in G :

$$M_0(G) = \{t1 = (a, b), t2 = (b, c), t3 = (c, d)\};$$

Then, Rel (T) has two elements : Rel (T) = {L1 = <x / (a, b), y/(b, c)>, L2= <x / (b, c), y / (c, d>}; The sets S_+ and S_- are defined as follows :

For (T, L1) : $S_+ = \{(a, c)\}, S_- =\{(a, b), (b, c)\}$

For (T, L2) : $S_+ = \{(b, d)\}, S_- = \{(b, c), (c, d)\}$

The firing of (T, L1) produces the following marking :

$$M_1(G) = (M_0(G) \cup (S_+ - S_-)) - (S_- - S_+)$$
$$= \{(a, c), (c, d)\}$$

Then, Rel (T) = {L'1 = <x / (a, c), y / (c, d)>}

For (T, L'1) : $S_+ = \{(a, d)\}, S_- =\{(a, c), (c, d)\}$

The firing of (T, L'1) produces the following stable marking :

$$M_2(G) = (M_1(G) \cup (S_+ - S_-)) - (S_- - S_+) = \{(a, d)\}.$$

For a given initial marking, a place P has a *stable marking* iff none of its input transitions is enabled. This notion is used to define the stability of a transition. A transition T is said to be *stable* for a given 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 stable.

When the PCN reaches a stable state, no transition can be enabled and the PCN evaluation stops. This corresponds to a fixpoint for the set of rules modelled by the net. We are now able to state the definition of a deterministic PCN. A transition T is *deterministic* iff, for any initial marking M_0, the output places of T have a unique stable marking depending on M_0. A PCN is deterministic iff for any initial marking M_0, the net has a unique stable marking depending on M_0.

Example :

In the example of figure 2.2, each transition is clearly deterministic and the PCN is also deterministic. Indeed, this PCN describes a monotone increasing function (in the sense of set inclusion partial order) over the marking of the net.

In the example of figure 2.6, the transition T does not describe a monotone function over the markings of the net. However, it is quite easy to demonstrate that such a transition is deterministic.

Finally, it is important to note that the net portrayed on figure 2.3 has no stable state.

A PCN evaluator has to execute the following procedure portrayed in figure 2.7. This procedure, starting from an initial marking, successively computes each reachable marking.

```
procedure evaluate (P: PCN, M0: Initial
      marking)
   M ← M0;
   repeat
     compute the conflict set;
     select a transition's occurrence (T, L)
     in the conflict set;
     compute M := reachable marking from M
     by firing (T, L);
   until a stable marking is reached.
```

Figure 2.7 : *A PCN evaluator.*

This procedure is not deterministic. The undeterminism stands in the choice of a transition occurrence in the conflict set, that is, (i) the choice in the conflict set of a transition T and (ii) the choice of the tokens used to fire T. In a deductive database context, the problem is to determine a sequence of transition's occurrences that leads to a *correct* and *efficient* computation.

Correctness refers to the declarative semantics of the corresponding program and to the stable marking associated with it. Indeed, the non monotony of the PCN evaluation denies the existence of a unique stable marking for all nets. When several stable markings are reachable, using a *meta-semantics* similar in essence to the stratification of logic programs [Apt86], a deterministic semantics can (under certain conditions) be assigned to the PCN. When it does, the meta semantics yields a partial ordering over the transitions, and thus constrains the choice of the firing of a transition's occurrence in the conflict set. Therefore, a *correct* computation must return the unique stable marking of the net when it exists under the previous meta-semantics or one of the possible stable markings when the net remains undeterministic. Efficiency refers to both the computation mode of a single transition (remark that evaluating a transition leads to evaluate a relational calculus expression) and the control of the inherent parallelism between rules. We shall see later two different computation modes for a transition (a set oriented mode and a tuple at a time mode).

The *flow of control* in a PCN is defined as the description of both the sequence of transition's occurrences firings, and the computation mode of each transition. In our framework, this comes to say that query optimization techniques are captured either by the syntactic transformation of the net or by describing a particular flow of control. Therefore, the problems that immediately arise are (i) to provide a formal way of describing the flow of control in a PCN and (ii) to define how to generate a correct and efficient flow of control. The former is tackled by the next section while section 4 discusses some aspects of the second problem.

3. FLOW OF CONTROL IN A PCN

3.1. A control language over the PCN

The previous section described the PCN structure and its semantics. In this section, we present a language used to describe the computation of a PCN.

As we have seen in section 2, a PCN is a non deterministic machine that describes sets of sequences over states of a database, and thus determines a relation on these states. This scheme was illustrated by the PCN evaluator procedure portrayed on figure 2.7. The implementation of such a procedure on a deterministic machine necessitates specifying the order in which enabled transitions are selected for firing (the *conflict resolution strategy*) and the order in which states of the database are expanded (the *backtracking strategy*). For instance, a possible conflict resolution strategy can be the stratified execution of a program. All these strategies are called the *evaluation scheme* of the PCN.

Remark that if backtracking is eliminated, for non deterministic PCN only one solution path can be generated for any given initial marking and query. In such cases, the solution set depends not only on the transitions and the marking of the net but also on the particular evaluation scheme used. Nevertheless, we do not consider any backtracking strategy in the evaluation scheme presented in this paper.

Our objective here is to define an explicit control component that is specified independently of the PCN evaluation scheme. Informally, this control can be represented by the set of all allowable sequences of transition firings; that is by specifying a language over the set of transitions. We will call such a language a *control language* . At each step of execution, the control language restricts the set of transitions that may be considered for firing and only a subset of the transitions in the transition's conflict set is *active*. We call a PCN together with a control language an *annotated PCN.*

All the procedural control is thus contained in the control language and is quite independent of the specification of the evaluation scheme. Note that when the control language places no constraint on transition's firings, (i.e., when the control language allows all possible transition sequences), an annotated PCN reduces to a non deterministic system whose execution is completely governed by the evaluation scheme. At the other extreme, where the control language makes the PCN deterministic, the evaluation scheme is inhibited.

We now define formally what is a control language over a PCN. First, recall that a PCN is defined as a triple (P, T, F). We define Σ as the set of all transition names of T. A control language over a PCN is any subset of Σ^* where Σ^* is the set of all strings over Σ. A word in Σ^* is called an *annotation.*

We are now able to define the execution of an annotated PCN = (P, T, F, A) where A is a set of annotations (or a control language over Σ). We first define a *state of execution* to be a pair <u, M> where u is a prefix of some word in A and M is a marking over the net. Now, let u and uT be prefixes for some word in A where $u \in \Sigma^*$ and $T \in \Sigma$. Then, we say a state of execution $<uT, M_2>$ is directly reachable from a state $<u, M_1>$ (noted $<u, M_1> \rightarrow <uT, M_2>$), if and only if one of the two following conditions holds :

(i) The transition T is enabled in M_1 and M_2 is reachable from M_1 by executing T.

(ii) the transition T is not enabled in M_1 and $M_2 = M_1$.

The last condition means that if a transition in the annotation is not enabled in the current marking, then we move on to the next symbol in the annotation.

Let $\rightarrow^*$ denote the reflexive transitive closure of $\rightarrow$. Then the reachability relation computed by an annotated PCN, (P, T, F, A), is a subset of M x M where M is the set of all possible markings over the net :

$$\{<M_1, M_2> : <\lambda, M_1> \rightarrow^* <u, M_2> \text{ for some u in A}\}$$

The set A of annotations, (or the control language), can be specified using a context-free language. We give below the context-free grammar associated with the language that describes all the possible annotations over a PCN.

The terminal alphabet is composed of Σ, +, (,), *, σ. The non terminal alphabet is: T, Q and the start symbol is S. Now, the productions are :

$$S \rightarrow (T + Q) \mid TQ \mid T$$
$$T \rightarrow (T + t) \mid Tt \mid t \mid (T)^* \mid (T)^\sigma$$
$$Q \rightarrow (Q + t) \mid Qt \mid t \mid (Q)^* \mid (Q)^\sigma$$

In this grammar, the symbol + denotes a disjunction (i.e., (T + Q) means that T or Q can be fired); the symbol σ stands for saturation and means that a sequence is repeated up to saturation; the symbol * means that a sequence is repeated 0 or more times. Finally, TQ means composition of T and Q.

Example :

Suppose one wants to express that T1 is always tried to be fired before T2 which in turn must always be tried before T3. The sequence $((T1)^\sigma T2)^\sigma$ expresses the precedence between T1 and T2. Now, the sequence $(((T1)^\sigma T2)^\sigma T3)^\sigma$ is the desired result. Thus, T1 is always evaluated first. If T1 is not enabled then T2 is evaluated. If T2 is fired using a given transition's occurrence then T1 is again evaluated. When a stable state is reached for T1 and T2 then T3 can be evaluated and so on. []

Remark that due to the presence of parenthesis, the above grammar is of the type a^nb^n and is not a regular grammar. Another point is that due to the disjunction "+", the set of annotations can be described by only one word in the above context-free language.

We are now able to present the modified evaluator procedure that executes an annotated PCN = (P, T, F, A). Initially, the state of execution is $<\lambda, M_0>$ for some initial marking M_0. Suppose that a state of execution <u, M> has been reached. If u is an element of A and M is a stable state (i.e., answers the query), then execution may terminate in absence of backtracking strategy. Otherwise, we consider all transitions T in the conflict set such that uT is a prefix of some word in A. If this set (called *active set*) is empty then execution terminates unsuccessfully; this means that the annotation is erroneous. Otherwise, a transition T is selected from the active set and T is executed giving raise to a new state of execution <uT, M'>. This is illustrated by the following procedure.

```
procedure evaluate annotated (P:annotated PCN,
   M0: Initial marking)
   M ← M0;
   u ← λ;
   repeat
      compute active set:= set of all transitions
      T in the conflict set such that uT is a
      prefix of some word in A;
       if active set = Ø and uT is not a word in A
       then return error;
      select a transition's occurrence (T, L) in
      the active set;
      compute M := reachable marking from M by
      firing (T, L);
      u ← uT ;
       if u is a word in A and M is not a stable
       marking then return error;
   until a state <u, M> is reached where M is a
      stable marking and u ∈ A.
```

3.2 Extended annotations

In the previous section we described a control language without indicating the computation mode of each transition. The purpose of the present section is to give a detailed description of these modes. Each of them yields a new basic symbol associated with a transition. The previous language is then extended to deal with these symbols instead of the transition names. We call *extended annotations* the words recognized by this new control language.

The first basic symbol is $F_{T, L}$. It means that a transition's occurrence (T, L') is fired, where L' belongs to the rèlevant set of T and L' matches the pattern list L. A *pattern list* is a list of the form $<x_1 / p_1, ..., x_n / p_n>$ where the x_i are all the free variables figuring on the labels of the input arcs of transition T and the p_i are pattern tokens. A ***pattern-token*** is simply a partially instantiated tuple. A non instantiated attribute value in such a token is represented by "?". A substitution of the form x_i / p_i where p_i only contains "?" values is named an ***empty substitution.*** A substitution list L' matches a pattern list L if each pattern token of L respectively matches each token of L'. A non instantiated value "?" matches any constant value. The procedure associated with this symbol is :

```
procedure compute_trans.(T:trans., L:pattlist,
            M:marking, C: boolean);
begin
  if C = false then compute Rel (T);
  Choose a list L' that matches pattern list L
  (if any) in Rel (T);
  compute the sets $S_+$ and $S_-$ ;
  for each place P such that
   there exists an arc (T, P) do
   compute M := $(M \cup (S_+ - S_-)) - (S_- - S_+)$;
end.
```

A particular case arises when L is such that all the variable substitutions are empty substitutions as defined above. In this case the symbol $F_{T, L}$ is abbreviated as F_T and its meaning is to fire a transition's occurrence of T chosen at random in the transition's conflict set.

The second basic symbol does not really belong to the semantics of a PCN. Rather, this symbol, noted $R_{T, L}$, defines a particular computation mode of a transition. The meaning of $R_{T, L}$ is first to compute rel (T) and then to fire T with all the occurrences of rel (T) without observing the intermediate modifications of T on the conflict set. Roughly speaking, it means that T is fired in a *set oriented way*. More formally, let T be a transition and (T, P) be an output arc of T. Let $+ t_1 ... + t_n$ be the positive label of (T, P) and $- s_1 ... - s_q$ be the negative label of (T, P). Two sets S_{R+} and S_{R-} are now defined as follows :

$$S_{R+} = \bigcup_{L \in Rel(T)} (\{L(t_i)\}) = \bigcup_{L \in Rel(T)} S_+$$

$$S_{R-} = \bigcup_{L \in Rel(T)} (\{L(s_j)\}) = \bigcup_{L \in Rel(T)} S_-$$

The procedure associated with the symbol $R_{T, L}$ is then :

```
procedure compute_rel_trans (T:trans.,
                    L:patternlist, M:marking);
begin
compute Rel(T);
for each place P such that there
            exists an arc (T, P) do
   begin
   compute $S_{R+}$ and $S_{R-}$ for all lists
   that satisfy pattern L;
   compute
   M (P) := $(M(P) \cup (S_{R+} - S_{R-})) - (S_{R-} - S_{R+})$;
   compute Rel(T);
   end
end.
```

The following remarks can be made : (i) such a procedure is deterministic, and (ii) the "for each" statement of the procedure exactly corresponds to compute t as a *Relational Algebra Program* (RAP) where the relations are the marking of the input places of T and the result of the program is the marking of the output places of T. Rel (T) is computed only once: thus, the procedure is equivalent to a usual relational algebra query. This is why such a transition firing is interesting in a DBMS context.

An extended annotation is then any word computed by the previous context-free grammar where Σ is now the set of symbols built from the two above basic symbols, the set of transition names in T, and the set of all pattern lists.

A particular case is the annotation $(F_{T, L})^{\sigma}$. It means that the transition T is fired up to saturation using the pattern list L as a filtering pattern for tokens. In a similar way as above, $(F_T)^{\sigma}$ means that T is fired up to saturation without any pattern. The associated procedure is :

```
procedure compute_transition_sat (T:trans.,
  L:patternlist, M:marking);
  begin
     compute Rel (T);
     C := false;
     while  Rel (T) ≠ Ø do %until a stable
                      marking M is reached%
     begin
        compute_trans. (T , L, M, C);
        compute Rel (T);
        C := true;
     end
  end.
```

Example :

On the net portrayed on figure 2.2, the extended annotation $(F_{T1,L1})^{\sigma} (F_{T2, L2})^{\sigma}$ where : L1 = <x / (?, sam)> and L2 = <(x / (?, ?), y / (?, sam)> computes in the ANCESTOR place all sam's ascendants. Indeed, $(F_{T1,L1})^{\sigma}$ successively copies all the PARENT's tokens that satisfy the pattern of L1 into ANCESTOR. Then $(F_{T2, L2})^{\sigma}$ computes the ascendants of sam. Equivalent annotations are either $(F_{T1,L1^*} F_{T2, L2^*})^{\sigma}$ or $((F_{T1,L1})^{\sigma} F_{T2, L2^*})^{\sigma}$. But $F_{T1,L1^*} F_{T2, L2^*}$ is not a *correct* annotation for the PCN. Remark that all these annotations have not the same computation cost.

An interesting problem is to detect the cases where the symbol $(F_{T, L})^{\sigma}$ can be rewritten into $(R_{T, L})^{\sigma}$ and yields the same result. When this is possible, the transition T is said to be *relational*. Note that, when the language is DATALOG^{neg}, all the rules are relational computable. A complete discussion of this problem can be found in [Simon88]. In the following, we only give some examples to illustrate the notion of relational computable transition.

Examples :

(i) Consider the PCN portrayed in Figure 2.2. The transitions T1 and T2 are relational. This means that the firing of T1 corresponds to a usual query in relational algebra, and that the repeated firing of T2 can be computed by a relational algebra program which is precisely a loop of joins onto the relation PARENT and ANCESTOR. In this particular example the word $(F_{T1})^{\sigma} (F_{T2})^{\sigma}$ can be rewriten into $(R_{T1})^{\sigma} (R_{T2})^{\sigma}$ which is a more efficient annotation.

(ii) Consider now the PCN portrayed in Figure 2.6. As presented in section 2, the computation of the word $(F_T)^{\sigma}$ leads G to the following stable marking : $M_2(G) = \{(a, d)\}$. We show in the following that a relational computation of this PCN does not provide the same marking in G.

The initial marking in G is $M_0(G) = \{t1 = (a, b), t2 = (b, c), t3 = (c, d)\}$; Then, we have :

$$S_{R+} = \bigcup_{L \in Rel(T)} S_+ = \{(a, c)\} \cup \{(b, d)\}$$

$$S_{R-} = \bigcup_{L \in Rel(T)} S_- = \{(a, b)\} \cup \{(b, c)\} \cup \{(b, c)\} \cup \{(c,d)\}$$
$$= \{(a, b), (b, c), (c, d)\}$$

The computation of the procedure `compute_rel_trans` leads to:

$$M_2(G) := (M_0(G) \cup (S_{R+} - S_{R-})) - (S_{R-} - S_{R+})$$
$$= \{(a, c), (b, d)\}.$$

Hence, the transition T is not relational, i.e., $(F_T)^{\sigma} \neq (R_T)^{\sigma}$.

4. GENERATION AND USE OF AN ANNOTATED PCN

In this section we present a general query processing overview of the system we are being implementing using PCN. We first discuss the evaluation scheme. Then, we present how the previous model is used to implement the production rule language RDL1.

4.1. The evaluation scheme

The evaluation scheme presented here only incorporates a conflict resolution strategy and proceeds in a bottom-up evaluation. The two basic techniques used for conflict resolution are : *firing by stepwise saturation* and *implicit ordering* of rules.

Implicit ordering corresponds to what is called *stratification* for logic programs as described in [Apt86]. A detailed discussion of what this concept becomes in our framework where we not only deal with negation but also with updates is given in [Simon88b]. Roughly speaking, it corresponds to traverse the net in the order specified by the oriented arcs of the net. This is called the *chaining* property.

We now discuss the choice of computation of a transition using a saturation mode (σ mode). Several cases arise according to the nature of the program. First, assume that the PCN is deterministic. Therefore the σ computation mode for all the transitions defines one correct computation among others. Is it an efficient one ?. In fact, two basic optimizations are made possible: (i) If a given transition is relational computable then $(F_T)^{\sigma}$ can be replaced by $(R_T)^{\sigma}$ in the annotation, (ii) it avoids the PCN evaluator to maintain a main memory environment for several rules at the same time; in particular, a transition that has already been fired and that does not appear in a repetitive sequence in the annotation is eliminated from the evaluator's environment.

Now, consider an a priori non deterministic PCN; two sub-cases are distinguished. First, the net has an underlying deterministic semantics that can be assigned using the implicit partial ordering mentioned above, or because an explicit meta-semantics was provided by the user (for instance, an annotation can be given using a particular grammar). If explicit information is given then the choice of the computation mode is also explicitly derived. Furthermore, if the net is only composed of deterministic transitions, then the explicit annotation may only concern the ordering between transitions. Thus, in this case, a σ computation mode can be assigned to all transitions and we get a correct computation.

The second sub-case that remains is the case of a non deterministic PCN for which a deterministic semantics cannot be infered. Here, there is no notion of correct computation and any mode (excepted to the relational one) can be a priori chosen.

Thus, in summary, a σ computation mode can be used when the PCN is assigned a deterministic canonical semantics. We call such a mode a firing by ***stepwise saturation*** mode. Obviously, a different mode arises when explicit annotations are provided by the user.

Example :

Consider the PCN represented Figure 4.1. This PCN admits a unique fixpoint on the contrary to the PCN portayed on Figure 2.3 which loops for ever.

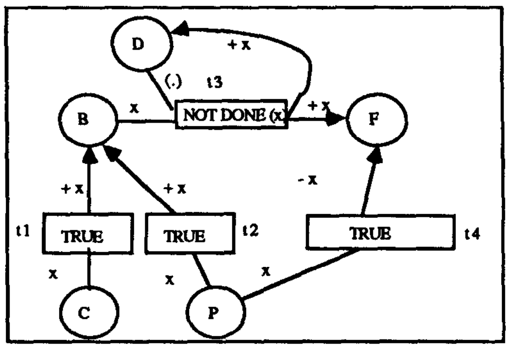

Figure 4.1

Thus, this PCN is deterministic and a correct extended annotation is for example : $w = (R_{T1})^{\sigma} (R_{T2})^{\sigma} (R_{T3})^{\sigma} (R_{T4})^{\sigma}$. Remark that in this case, this annotation can be generated by the system itself.

4.2 A general query processing overview.

In this section, we briefly present the different steps of the evaluation of a query by generation of an extended annotated PCN.

When a query enters the system, the first step consists of retrieving the pertinent PCN, i.e., the minimal PCN which is able to produce the desired marking in the result place. Then, this minimal Query PCN is build in main memory. The second step consists of an optimization and translation phase. The optimization techniques being currently implemented in our system are : the rewriting of the $(F_T)^{\sigma}$ words into $(R_T)^{\sigma}$, a generalization of query modification in order to minimize the number of transitions appearing in the query PCN, the transformation of a set of database updates into an optimized one as [Sellis85] does, and the "push-up" of the selections. These algorithms are described in [Maindreville87].

Let us review the different uses of annotations. First, the control language enables to model a procedural control over the rule language that is not hidden in the rules [Georgeff82]. Whatever the meta-language is for expressing this procedural control, annotations provide some means to capture it in a compiled form. A second point is the use of annotations for capturing evaluation algorithms. For instance, the "push-up" of constants can be expressed in the extended control language in the same spirit as [Aly87]. If an annotation was already specified by the user, query optimization techniques correspond to transform the specified annotation into an optimized and an extended one.

The last step is the evaluation of the annotated PCN. As a result of this step, a sequential (i.e., without use of disjunction) extended annotation is produced. It is then transformed into an access plan composed of relational algebra programs, including specialized operators in order to compute efficiently the non relational rules. The different steps of the query processing are summarized on the following figure:

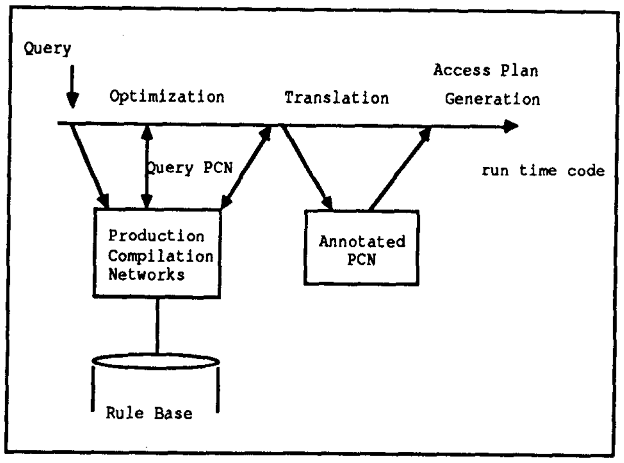

Figure 7.1 : *query processing overview*

5. CONCLUSION

This paper presented a new execution model called a Production Compilation Network (PCN). This model is an extension of a Petri-Net based model, namely Predicate Transition Nets. We showed how to model the static aspect and the dynamic aspect of RDL1, a production rule language for databases. RDL1 can be seen as an extension towards the support of updates for DATALOG^{neg}. Hence, the PCN model provides a uniform framework to model the evaluation of queries and updates in a deductive database context.

We introduced a control language over the PCN to describe different evaluation schemes and algorithms in a clear and concise manner. In particular, we showed how to describe the relational computation of a production rule. The set of basic symbols of the PCN language as presented in this paper permits only to capture a bottom-up evaluation strategy. We defined elsewhere [Maindreville88b] other symbols that capture pure top-down evaluation strategies or mixed strategies (as in the query/subquery approach). Using the full PCN language, we are able to describe the Prolog evaluation strategy for DATALOG-like rules. Finally, the structure aspect of a PCN enables to capture a partial ordering over the set of transitions. This ordering can be considered as an extension of the notion of stratification for a production rule language, [Simon88b].

Our main contribution is to provide a good intermediate model that can be used as a control structure for implementing rule based languages for deductive databases. For instance, it is possible to describe the computation of a rule program using either a tuple at a time computation mode or an extended relational algebra program or both. Moreover, our claim is that the implementation of DATALOG^{neg} programs can benefit from using PCN because (i) the efficient evaluation of a DATALOG^{neg} program requires implementing a variety of strategies and algorithms, and (ii) procedural control enables to describe more meaningful programs. Thus, we need to describe control over rules. To illustrate the first reason, consider the recursive query evaluation problem. No universal optimization algorithm is available and different algorithms need to be supported according to the structure of the data in the database relations (cyclic or acyclic, tree, cylinder, ...), the structure of the rules (chain rules, linear rules, ...), and the operators supported by the system (e.g., transitive closure). PCN offer a uniform way to describe these algorithms with their application area.

A first implementation of the method has shown the adequacy of the execution model. It also pointed out some research perspectives :

•to extend the model to support : (i) integrity constraints over the derived relations, (ii) access methods information in order to include physical optimization in the query processing process,

•to map parallelism onto the model,

•to use annotations for parameterizing various evaluation schemes

Acknowledgments :

The authors would like to thank Serge Abiteboul for the numerous fruitful discussions, Nathalie Lefebvre for her comments on a previous version of this paper and Marie Aude Portier for her participation in the implementation.

References :

[Abiteboul87a] S. Abiteboul, V. Vianu : *"A Transaction Language Complete for Database Update and Specification."* , in ACM PODS, 1987.

[Aly87] H. Aly , Z. M. Ozsoyoglu : *"Non-deterministic Modelling of Logical Queries in Deductive Databases "* Proc of ACM-SIGMOD, Los Angeles, 1987.

[Apt86] K.R. Apt, H. Blair, A.Walker : *" Towards a Theory of DeclarativeKnowledge"* . IBM Report RC 11681, April 1986.

[Bancilhon86] F. Bancilhon, R. Ramakrishnan : *"An amateurs's introduction to recursive query processing strategy"*, Proc of ACM SIGMOD, 1986.

[Bancilhon86b] F. Bancilhon, D. Maier, Y. Sagiv, J. Ullman : *"Magic sets and other strange ways to implement logic programs"*, 5th ACM Symp. of Princ. on Database Systems.

[Beeri87] C. Beeri et al., :*"Sets and Negation in a Logic Database Language (LDL1)"*, MCC Tech. Report, Nov. 1986.

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

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

[Gardarin85] G. Gardarin, C. de Maindreville, D. Mermet, E. Simon : *"Extending a Relational DBMS towards a KBMS : A First Approach ."* Springer Verlag, Ed Schmidt, Thanos, 88.

[Genrich86] H. J. Genrich : *"Predicate / Transition Nets "* , in Advances in Petri Nets' 86. Springer Verlag, 1987.

[Georgeff82] M. P. Georgeff : *"Procedural Control in Production Systems"*, Artificial Intelligence 18 (1982).

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

[Kripke63] S. Kripke : *"Semantical consideration on Modal Logic"* , Acta Philiosophica Fennica, Helsinki, 1963.

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

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

[Maindreville87] C. de Maindreville, E. Simon :*"A Predicate Transition Net for Evaluating Queries against Rules in a DBMS."* INRIA Research Report, N° 604, Feb. 1987. abstract in Proc of 3th JBDA, Port-Camargue, France, 1987.

[Maindreville88] C. de Maindreville, E. Simon : *"A Production Rule Based Approach To Deductive Databases"*, Proc of 4th Int. Conference on Data Engineering, Los Angeles, Feb. 88.

[Sellis85] T.K. Sellis, L. Shapiro : *"Optimization of extended database query languages."* ACM SIGMOD, Austin, 1985.

[Simon88] E. Simon, C. de Maindreville : *"Deciding whether a production rule is relational computable"* Proc. of International Conference on Database Theory, Bruges, Belgium, Sept. 88.

[Ullman82] J.D. Ullman : *"Principles of Databases Systems"*, Computer Science Press, 1982.

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

[VanGelder86] A. Van Gelder : *"A Message Passing Framework for Logical Query Evaluation"*, ACM SIGMOD 86.