Analyzing Repetitive Evaluations of Active Rules Within a Transaction

Françoise Fabret, François Llirbat, Eric Simon

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Analyzing Repetitive Evaluations of Active Rules Within a Transaction

Françoise Fabret, François Llirbat, Eric Simon

N° 2816

March 1996

PROGRAMME 1

Rapport de recherche

ISSN 0249-6399

Analyzing Repetitive Evaluations of Active Rules Within a Transaction

Françoise Fabret *, François Llirbat *, Eric Simon *

Programme 1 — Architectures parallèles, bases de données, réseaux et systèmes distribués
Projet Rodin

Rapport de recherche n°2816 — March 1996 — 26 pages

Abstract: An active database system automatically triggers rules in response to certain events occuring. Events are issued by transactions or action parts of rules. Repeated executions of rules can be caused by the structure of the initial triggering transaction program and by the structure and execution semantics of rules. Repeated calculations of rules may incur costly redundant computations in rule conditions or actions. The central contribution of this paper is to propose technics for analyzing the behaviour of a transaction and a set of rules triggered by this transaction in order to derive: (i) if a given rule is processed more than once, and (ii) a fine indication of the database changes that may occur between two consecutive executions of the rule. Knowing these changes, it is possible to use existing algorithms that compute useful intermediate expressions in a rule that can be cached and incrementally maintained in order to avoid redundant computations. A notable property of our analysis technics is that they are parametrized by a few essential semantics parameters that define the execution semantics of an active rule language. Thus, our analysis apply to a large class of existing active rule systems.

Key-words: Active databases, transactions

(Résumé : tsvp)

* E-mail: {Firstname.Lastname}@inria.fr

Unité de recherche INRIA Rocquencourt
Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex (France)
Téléphone : (33 1) 39 63 55 11 – Télécopie : (33 1) 39 63 53 30

Analyse de l'évaluation répétitive des règles actives dans une transaction

Résumé : Un système de base de données active déclenche automatiquement des règles en réponse à l'occurence de certains événements. Les événements sont générés par la transaction ou par les actions des règles. Les exécutions répétitives des règles peuvent être causées par la structure du programme de transaction provoquant le déclenchement initial et par la sémantique d'exécution des règles. L'évaluation répétitive des règles peut conduire à effectuer des calculs redondants coûteux tant dans la condition que dans l'action des règles. La contribution principale de ce papier est de proposer des techniques pour analyser le comportement d'une transaction et d'un ensemble de règles déclenchées au cours de cette transaction dans le but d'en déduire: d'une part si une règle donnée peut être exécutée plusieurs fois et, d'autre part, une indication précise sur les changements de l'état de la base pouvant se produire entre des exécutions consécutives de la règle. A partir de la connaissance de ces changements, il est alors possible d'utiliser des algorithmes existants pour isoler des expressions intermédiaires dont la mémorisation et la maintenance par calcul incrémentiel permet d'éviter des calculs redondants. Il est à noter que nos techniques d'analyse sont paramétrées par les paramètres sémantiques essentiels qui définissent la sémantique d'exécution d'un langage de règles actives. De ce fait, notre méthode d'analyse est applicable à un large éventail de systèmes actifs existants.

Mots-clé : Bases de données actives, transactions

1 Introduction

An active database system automatically triggers *Event-Condition-Action* (ECA) rules in response to certain events occuring. Events are issued by transactions or action parts of rules. The points at which rules may be triggered and executed is determined by the rule processing granularity of the active database system. For instance, in a relational active system, using an "SQL statement" rule processing granularity (e.g., SQL3), rules can be triggered after or before every SQL data modification command issued by the transaction or the rules. Depending on the granularity of rule processing and other parameters that characterize the rule execution semantics, a given rule can be triggered and executed several times as the following concrete examples show.

1.1 Motivating Example

We consider an information system representing the activity of an industry which must manage, sell, and distribute a product worldwide in the flavor of [TPC95]. We assume that the industry holds a set of widely distributed stores, and each store has a fleet of trucks for deliveries. Orders are registered in the database using two relations $Order$, and $Lineitem$. The supplier in $Lineitem$ is not known when the order is entered. A relation $Shipment$ records all shipments to customers, and a relation $Fleet$ records all supplier's delivery trucks. Finally, a relation Av_truck records all available trucks which are not yet assigned any delivery for the next 8 days. The schema for these relations is given below.

```
Order    (orderkey, custkey, orderdate, cust_area, ...)
Lineitem (orderkey, linenumber, partkey, suppkey, space_occ, ...)
Shipment (orderkey, linenumber, shipdate, truckkey, area, space_occ, ...)
Fleet    (suppkey, truckkey, size, ...)
Av_truck (truckkey, date)
```

In a first scenario, assume a transaction program first selects all tuples from $Lineitem$ of a given $orderdate$. For each such tuple, the appropriate supplier (i.e., a store) is determined and $suppkey$ is assigned a value. The pattern of this embedded SQL transaction is sketched on Figure 1.

```
begin-trans
declare cursor for
  select Lineitem.* from ...
open cursor
do-while
  ....
  /* find a supplier */
  update Lineitem set suppkey=...
od
end-trans
```

```
create trigger R1
after update of suppkey on Lineitem
   begin
   select ... into Total_deliv ...;
   if ... /* code1 */
   then insert into shipment ...;
   else
     ... /* code2 */;
     insert into shipment ...;
   end
```

Figure 1: first scenario

Suppose an active rule is defined on $Lineitem$ and triggered after an update of $suppkey$. The pattern of the rule is sketched on Figure 1 using a concrete syntax inspired from SQL3. The rule action first issues a query that returns in a temporary table $Total\_deliv$ (truckkey, deliv_day, space_left, visits), for each supplier's truck and delivery day, the space left in the truck, and the number of customers visited. The query only selects deliveries planned for the area of the customer (we assume a truck visits one area per day). The text of the query is given on Figure 2. Then "code1" searches for one tuple of $Total\_deliv$ having the minimal $deliv\_day$ and such that $space\_left$ is less than the space occupied by the lineitem (attribute $space\_occ$), and $visits$ does not equal a maximal value. If it exists, this tuple is used to compute the tuple inserted into $Shipment$. Otherwise in "code2", an available truck is picked (and removed) from Av_truck and used to build the tuple inserted into $Shipment$.

```
select f.truckkey, s.shipdate, f.size - sum(s.space_occ),
       count(s.orderkey)
into   Total_deliv (truckkey, deliv_day, space_left, visits)
from   Fleet f, Shipment s, Order o
where  f.truckkey = s.truckkey and new.orderkey = o.orderkey
       and o.cust_area = s.area and new.suppkey = f.suppkey
groupby f.truckkey, s.shipdate
order by s.shipdate
```

Figure 2: The Total_deliv SQL query

If we assume that the active system uses an "SQL statement" rule processing granularity then the rule will be executed after every update statement in the while-loop of the transaction, and so will the $Total\_deliv$ query. However, the result of the queries corresponding to two consecutive triggering updates of $Lineitem$ with a same supplier are quite the same. In fact, only one tuple of $Total\_deliv$ will be changed from one result to the other: $space\_left$ will be reduced and $visits$ will be incremented. This can be deduced from the analysis of the $Total\_deliv$ query and the fact that each time the trigger executes a single tuple is inserted into $Shipment$, which takes part in this query.

Because the $Total\_deliv$ query is quite complex, at each computation many costly redundant operations are performed. Thus, a much better implementation strategy would be for instance to cache the result of $Totals\_deliv$ after the first execution of the rule, and to incrementally maintain it using extra operations placed just before the end of R1's action.

A slightly different scenario would generate the same repeated execution. Suppose that a transaction program inserts a tuple into $Order$ and a set of tuples into $Lineitem$ for which the supplier is already given. Suppose a rule is defined on $Lineitem$ and triggered by an insert. The rule executes the same action as rule R1. Suppose the rule is defined with an instance-oriented execution granularity, which means that its action is executed "for each row" inserted in $Lineitem$, and we use a "delayed" rule processing granularity, which means that the rule is only triggered at the end of the transaction. The rule and the transaction are sketched in Figure 3. In this case, since a set of tuples is inserted by the transaction, the execution of R2 can be repeated and hence redundant computations of $Total\_deliv$ may occur.

```
begin-trans
insert into Order values ...;
insert into Lineitem values ...;
...
insert into Lineitem values ...;
end-trans
```

```
create trigger R2
after insert on Lineitem
  for each row
  begin
  /* same action as R1 */
end
```

Figure 3: second scenario

As a last scenario, consider the same transaction program as before except that the supplier is not provided in $Lineitem$. Assume we have two rules. The first one is defined on $Lineitem$ and triggered by an insert, and its action determines the appropriate supplier according to the city of the customer (i.e., a single supplier is selected for the entire order entry). The action of the rule is executed once "for each statement" that triggers the rule. The second rule is the same as rule R1 except that we specify that the action is executed "for each row" updated in $Lineitem$. The rules are sketched on Figure 4. Rule R4 is executed as many times as there are updated tuples in $Lineitem$ by the action part of R3.

```
create trigger R3
after insert on Lineitem
  for each statement
  update Lineitem set suppkey=...
  where ...
```

```
create trigger R4
after update of suppkey
on Lineitem
  for each row
  begin
  /* same as R1 */
  end
```

Figure 4: third scenario

The above example hopefully makes two points. First, repeated executions of rules can occur in subttle ways. In the first scenario, repetition is caused by the structure of the transaction (while loop) and the SQL statement rule processing granularity, whereas in the second and third scenarios, it is caused by the instance-oriented rule execution granularity. These scenarios accurately reflect the situation of real applications because existing products only offer an SQL statement or even a tuple rule processing granularity, and users tend to prefer to use instance-oriented rules [SKD95], [Coc96]. In the three scenarios, we showed that a realistic analysis of possible rule repetitions requires to take into account the structure of the triggering transaction because (i) it plays a direct role in the occurence of repetitions, and (ii) it determines a maximal potential set of rules that can be triggered (usually, a small set) and that needs to be analyzed.

Second, these repeated calculations of rules may incur redundant computations in rule conditions or actions (e.g., $Total\_deliv$). However, if we know that a rule executes several times and the database changes that may occur between two consecutive executions, then it is possible to use existing algorithms such as [FRS93] and [RSS96] to derive which useful intermediate expressions in a rule condition or action can be cached or materialized.

1.2 Research Contribution

The central contribution of this paper is to propose techniques for analyzing the behaviour of a transaction and a set of rules triggered by this transaction in order to derive: (i) if a given rule is processed more than once, and (ii) the *relevant* database changes that may occur between two consecutive executions of the rule. For instance, in the first scenario, although R1 may perform an insert into $Shipment$ and a delete to Av_truck each time it executes, only the insert is relevant because $Shipment$ takes part in the $Total\_deliv$ query.

A first difficulty is to propose analysis techniques that apply to a wide variety of active rule languages which indeed differ considerably in their execution semantics. To address this problem, our analysis techniques are parametrized by a few essential parameters that define the execution semantics of an active rule language. Thus, an important feature of our analysis techniques is to apply to a large class of existing active rule systems.

A second difficulty is that a single active system may offer various possibilities of execution semantics, e.g., different rule processing granularities. Indeed, the three above scenarios could easily happen within a single active system. A second major feature of our rule analysis is to be general enough to cope with the many combinations of rule execution semantics that can be offered by a given system.

1.3 Outline of the Paper

Section 2 presents the semantic parameters of rule execution semantics retained by our analysis techniques. Section 3 introduces an abstract representation of transactions and rules. Useful data structures for carrying the rule analysis are defined in Section 4. Section 5 contains the detailed analysis of rule executions. The global analysis of a transaction and the triggered rules are given in Section 6. In Section 7, we compare our results with other work. Section 8 concludes the paper.

2 Semantics of Rule Execution

Troughout this paper, we consider relational databases and we assume that the active rule base is defined as a set of ECA rules that consist of an event that causes the rule to be triggered, a condition that is checked when the rule is triggered, and an action that is executed when the rule is triggered and its condition is true. The triggering event is a data modification operation, i.e., an insertion, deletion or update, applied to a given relation. Thus, we only consider simple events. The condition is an SQL search condition over the database, and the action is an atomic procedure that may contain SQL statements combined with other procedural constructs.

There exists a large variety of ECA rule languages, in both the research and commercial arenas, that considerably vary in their syntax and semantics [WC96]. A few recent papers have proposed to describe and classify rule execution semantics according to different dimensions and parameters [FT95] [WC96]. We characterize the range of rule execution semantics to which the analysis techniques specified in this paper apply, using the semantics parameters introduced in these classification frameworks. We first present the (fixed) parameters, which have a pre-determined value in our analysis, then the (variable) parameters for which our analysis techniques allow different values. Other parameters such as the net effect policy [WC96] are irrelevant with respect to our analysis techniques.

2.1 Fixed Parameters

C-A coupling mode: when a rule is triggered, its condition is first evaluated and then if it is true the corresponding action is immediately executed within the same transaction than the transaction that triggered the rule. This is referred to as immediate C-A coupling mode.

Event consumption mode: It specifies if an operation that triggered a given rule can retain its capability of triggering rules after the rule is processed. We restrict ourselves to local consumption at evaluation time [FT95]. This means that each triggering operation of a rule r is always consumed whatever is the result of condition evaluation and can no longer trigger r. Nevertheless, it can trigger other rules.

2.2 Variable Parameters

Rule execution granularity: It indicates if the rule is instance-oriented (noted for-each-row granularity), or set-oriented (noted for-each-statement granularity). An instance-oriented rule is executed once for each instance of a database operation triggering the rule, whereas a set oriented-rule is executed once for all instances of a database operation triggering the rule [WC96]. For instance, a rule whose event is an insert is triggered once for each tuple in the set of tuples inserted by an insert operation if it is instance-oriented and only once for the entire set of inserted tuples if it is set-oriented.

Rule processing granularity: It describes how often the points (henceforth, called *rule processing points*) occur at which rules may be processed. This granularity is chosen once for all in a given system. Rules can be processed after or before every SQL data modification command issued during the transaction. This is referred to as SQL-statement granularity. Using a finer granularity, referred to as tuple granularity, rules can be processed after each occurence of an insert, delete, or update of a single tuple. Finally, at a coarser granularity, referred to as delayed granularity, the execution of rules can be delayed until a specific point placed by the user into the transaction or until commit time. With SQL-statement (resp. tuple granularity) we distinguish two kinds of rules: before rules are processed just before the triggering SQL statement (resp. tuple operation) while after rules are processed just after the triggering SQL statement (resp. tuple operation).

Rule processing behaviour: It specifies how the rules are executed at rule processing points. In particular, the operations of a rule action may trigger other rules. As in [WC96], we distinguish two kinds of behaviours[1]:

[1]We do not consider a parallel execution of rules as in Hipac or Sentinel.

1. With a recursive behaviour, the execution of a rule recursively invokes the processing of the rules triggered by its action part. If rules are noninterruptable, the recursive invocation is made at the end of the action part. If rules are interruptable, the recursive invocation is made at processing points within the action. In most systems having interruptable rules, these points usually occur before or after each SQL statement for SQL-Statement granularity and before or after each tuple operation for tuple granularity. Moreover, if several rules are triggered at the same time, active systems allow to statically specify the order in which they must be considered. We shall consider both recursive interruptable and recursive noninterruptable behaviours and take into account the static order relationship (if any) between the rules, noted $\prec$.

2. With iterative behaviour, one triggered rule is successively selected and processed until there are no triggered rules. The criteria used to select a rule at each step may be deterministic (e.g., a static total priority ordering, or a total dynamic ordering such as breath-first evaluation of rules), or non-deterministic (e.g., a static partial ordering). Note that an iterative behaviour implicitly assumes that rules are noninterruptable. We shall consider iterative behaviour and take into account the static order relationship (if any) between the rules, noted $\prec$.

2.3 Active Systems Captured

Most relational active systems use a recursive interruptable rule processing behaviour [WC96, FT95]. They usually propose both for-each-row rules and for-each-statement rules. In $Ingres$, and $Postgres$, the rule processing granularity is tuple, while it is SQL-statement in $Sybase$ and $DB2\ Common\ Server$. Other systems as $Oracle$ and $Informix$ mix the two granularities: for-each-row rules are executed with tuple rule processing granularity, while for-each-statement rules are executed with SQL-statement granularity. In the $SQL3$ standard [Coc96, ISO95], both SQL-statement and delayed rule processing granularities are proposed.

The iterative rule processing behaviour is used in several research prototypes. For instance, $Starburst$ provides for-each-statement rules with delayed rule processing granularity.

In this paper, we focus on sets of rules where the rules are executed with the same rule processing granularity and the same rule processing behaviour. We claim that our results can be easily adapted for systems that mix such rule sets.

3 Representation of Transactions and Rules

3.1 Abstract Programs

We assume that programs specified in transactions and action parts of rules interact with the database using SQL statements (e.g., embedded-SQL transaction programs, stored procedure programs).

We shall use abstract programs which essentially represent the flow of atomic database operations denoted by their intentions, together with the programming control structures embedding these statements. We restrict the programming control structures used in a program to sequential compositions, conditionals (noted *ifthen else*) and while-loops (noted *whiledo*). We intentionally omit to represent the conditions in conditional and whiledo statements since they will not be used in our analysis.

In abstract programs, an SQL statement is represented by the set of its operations denoted by their intentions as follows: $T.c$ denotes a read operation on the column c of a table T, $+T$ (resp. $-T$) denotes an insert (resp. a delete) in a table T, and $\pm T.c$ denotes an update of the column c of a table T. This set of operations can be easily deduced from the syntactic analysis of an SQL statement.

Example 3.1 Suppose we have three relations A, B, and C, and attributes in these relations are respectively denoted a_i, b_i, and c_i. The following SQL statement

```
update A  set A.a1 = A.a1+ 10
where A.a2 = B.b1 and B.b2 = C.c1
```

is represented by the set of operations denoted by their intentions: $\{ \pm A.a1, A.a1, A.a2, B.b1, B.b2, C.c1\}$

Abstract programs capture the rule processing granularity adopted in an active database system. With tuple granularity, each SQL statement s in a program is mapped into a *whiledo* s *od* statement. With delayed granularity, a specific checkpoint statements noted *chk*, specifies each rule processing point in a transaction program.

We depict an abstract program using a reducible flow graph [ASU86]. Statements have a unique label to distinguish them in the graph. Two specific nodes, *bop* and *eop*, respectively indicate the beginning and the end of the program. We shall say that a statement s_1 *precedes* a statement s_2 if there is a path from s_1 to s_2 in the flow graph. Intuitively, this means that there exists a possible execution of the program where s_1 executes before s_2.

Example 3.2 Using the same relations as before, let $P0$ be the following program:

```
bop; s1 ; s2; whiledo s3; s4 od; eop;
```

where s_1 to s_4 are SQL statements such that: $s_1 = \{+A, A.a_1, B.b_1\}$, $s_2 = \{+D, A.a_1, B.b_1\}$, $s_3 = \{+B, A.a_1\}$, and $s_4 = \{+C, A.a_1, B.b_1\}$. The flow graph of Figure 5(a) depicts the abstract program for $P0$ with SQL-statement granularity having four SQL statements represented by nodes s_1 to s_4. Figure 5(b) represents the same program with tuple granularity.

3.2 Abstract Representation of Rules

We characterize a rule r by the following functions:

- *Triggered_by* takes a rule r and returns the operation that triggers r.
- *Action* takes a rule r and returns the flow graph associated with its action part.
- *Condition* takes a rule r and returns the set of operations occuring in the condition part.
- *Performs* takes a rule r and returns the set of operations occurring in the action of the rule.

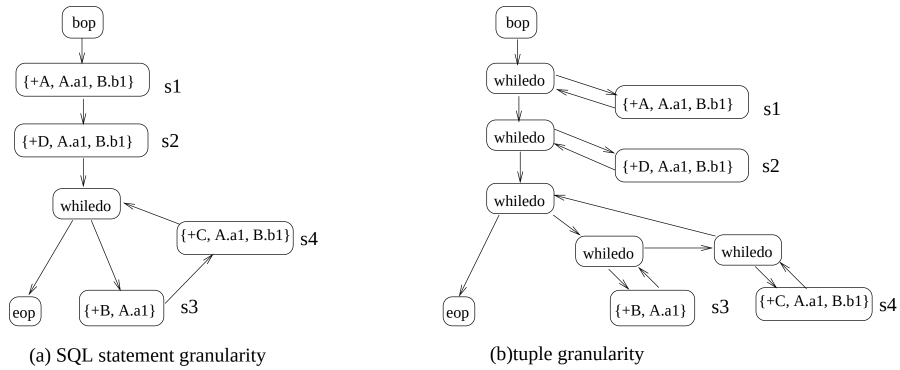


Figure 5: flow graphs for $P0$

- *Conflict* takes a rule r and returns a set of operations. An operation *op* is in *Conflict(r)* if :
  1. *op* $\in \{+T, -T\}$ and $T.c \in Condition(r) \cup Performs(r)$ for some attribute c of a relation T.
  2. *op* $= \pm T.c$ and $T.c \in Condition(r) \cup Performs(r)$ for some attribute c of a relation T.

Intuitively, $Conflict(r)$ gives the data modification operations which, if executed after r, can affect the result of the select operations in r's condition and action.

Example 3.3 let r be an ECA rule defined as follows:

```
r : Triggered_by(r) = {+A}
    Condition(r)    = {A.a2,B.b1}
    Action(r)       = P0
```

where $P0$ is the program of Example 3.2. Then, $+A$, $-A$, $\pm A.a1$, $+B$, $-B$ and $\pm B.b1$ are in $Conflict(r)$ since B.b1 and A.a1 are in $P0$. And $\pm A.a2$ is also in $Conflict(r)$ since $A.a2$ is in $Condition(r)$.

4 Data Structures

4.1 Simplified Flow Graph

Given a program $\mathcal{P}$ we contruct a simplified flow graph, noted Φ_P, which contains simple nodes and loop nodes. A simple node corresponds to an SQL statement or a chk statement which is not embedded in a whiledo control statement. A loop node corresponds to an outermost whiledo control statement. It is characterized by the set of all SQL statements and chk statements involved in the loop. The root of the graph is the bop statement and the exit node is the eop statement. There is an arc between two nodes n and n' of Φ_P iff n contains an SQL statement that precedes in the flow graph an SQL statement in n'.

Example 4.1 Figure 6(a) shows the simplified flow graph of $P0$ (see example 3.2) in case of SQL-statement granularity. It is derived from the flow graph in Figure 5(a). Figure 6(b) shows the simplified flow graph for $P0$ in case of tuple granularity. It is derived from the flow graph in Figure 5(b). Figure 6(c) depicts the simplified flow graph of $P0$ in case of delayed granularity and if the chk statements are placed in $P0$ as follows:

```
bop; s1 ; chk; s2  whiledo s3; chk; s4 od; eop;
```

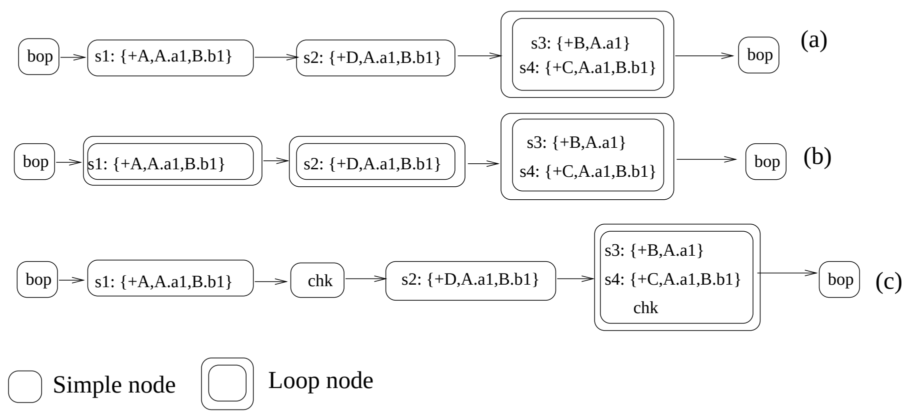


Figure 6: Simplified flow graphs for $P0$

By reducing the loops of the flow graph into loop nodes in our simplified graph, we adopt a pessimistic approach. Indeed, we consider that the loop will be executed several times and consequently that the execution of each statement in the loop both precedes and follows the execution of the other statements in the loop.

The simplified flow graphs exhibit the processing granularity. SQL statements which are not embedded into a loop in the original program are represented by simple nodes if the granularity is SQL-statement and by loop nodes if the granularity is tuple granularity. Finally, for delayed granularity, the chk statements of the original program are put in either simple or loop nodes.

In the following, without a loss of generality we focus on *simple programs*, that consist of a traversal path of the activation graph (i.e., programs that do not contain conditional branching outside of a loop). General programs that result in multiple traversal paths can be handled by either analysing separately each path, or translating (pessimistically) each conditional of the form $<$ifthen path1 else path2 $>$ into $<$ path1; path2 $>$. However, this issue will not be covered in this paper.

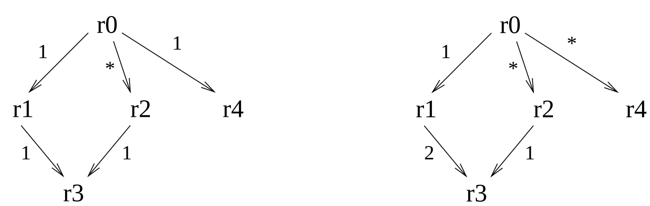
4.2 Triggering Graph

Given a set of rules Γ and their semantics, we represent the interactions beetwen the rules by means of a labelled directed graph, called a triggering graph, where labels are parametrized by the execution granularity of the rules and the fact that the rules are interruptable or not. There is one node per rule in Γ. There is an arc from rule r to rule r' if firing r can trigger r', i.e., $Triggered\_by(r) \cap Performs(r) \neq \emptyset$. There are three kinds of labels on the arcs : the *Trigger*, *Forward* and *Backward* labels.

The *Trigger* label on arc (r, r') indicates how many triggering of r' are caused by a single execution of r. The label is "$*$" if r' is for-each-row or if r is interruptable and the triggering operation of r' is embodied in a loop in the action part of r (the label "$*$" means 0 or more times). The label on (r, r') is "1" if r' is for-each-statement and r is noninterruptable. The label is "k" ≥ 1 if r' is for-each-statement, r is interruptable and the triggering operation of r' occurs k times in the action part of r.

The *Forward* label on arc (r, r') gives the maximal set of data modification operations in $Action(r)$ that may be executed after any complete execution of r'. If r is noninterruptable, $Forward(r, r') = \emptyset$. In the case where r is interruptable, an operation *op* is in $Forward(r, r')$ if $Action(r)$ contains two nodes n and n' (not necessarily distinct) such that n precedes n', *op* is in n', and $Triggered\_by(r)$ is in n. If r is a before rule, every operation of n is in $Forward(r, r')$.

The *Backward* label on arc (r, r') gives the set of data modification operations in $Action(r)$ that may execute before the begining of an execution of r'. If r is noninterruptable, $Backward(r, r')$ = $Performs(r)$. In the case where r is interruptable, an operation *op* is in $Backward(r, r')$ if $Action(r)$ contains two nodes n and n' (not necessarily distinct) such that n' precedes n, *op* is in n', and $Triggered\_by(r)$ is in n. If r is an after rule, every operation of n is in $Backward(r, r')$.



(a) : noninterruptable rules

(b) : interruptable rules

Forward labels:	Backward labels:	Forward labels:	Backward labels:
(r0,r1) : {}	(r0,r1) : {+A, +D, +B, +C}	(r0,r1) : {+D, +B, +C}	(r0,r1) : {+A}
(r0,r2) : {}	(r0,r2) : {+A, +D, +B, +C}	(r0,r2) : {+B, +C}	(r0,r2) : {+A, +D}
(r0,r4) : {}	(r0,r4) : {+A, +D, +B, +C}	(r0,r4) : {+B, +C}	(r0,r4) : {+A, +D, +B, +C}
(r1,r3) : {}	(r1,r3) : {-D, -C}	(r1,r3) : {-C}	(r1,r3) : {-D, -C}
(r2,r3) : {}	(r2,r3) : {-C}	(r2,r3) : {}	(r2,r3) : {-C}

Figure 7: Triggering Graphs

Example 4.2 Let $P0$ be the program defined in Example 3.2 and let $r0$, $r1$, $r2$, $r3$, $r4$ be five rules characterized as follows:

```
r0: Triggered_by(r0)= {-B}      r1: Triggered_by(r1)= {+A}
    Condition(r0)= {A.a1}           Condition(r1)= {A.a1,B.b1}
    Action(r) = P0                  Action(r1)= {-D};{-C,A.a1};{-C,B.b1};

r2: Triggered_by(r2)= {+D}      r3: Triggered_by(r3) = {-C}
    Condition(r2)= {A.a1,D.d1}      Condition(r3) = {D.d1,C.c1}
    Action(r2)= {-C,A.a1};          Action(r3) = {-A};

r4: Triggered_by(r4)= {+C}
    Condition(r4)= {D.d1,C.c1}
    Action(r4)= {+B};
```

We assume that the rule processing granularity is SQL-statement. We also assume that $r0$, $r1$, $r3$, $r4$, are for-each-statement rules and $r2$ is a for-each-row rule. The simplified flow graphs of the action parts of $r1$, $r2$, $r3$, $r4$ are easily derived: they consist of a sequence of simple nodes (one simple node per SQL-statement).

Figure 7(a) shows the resulting triggering graph when all rules are noninterruptable. The arcs show that $r0$ triggers $r1$, $r2$, and $r4$, while $r3$ is triggered by $r2$ and $r1$. The $trigger$ label on arc ($r0$,$r2$) is "$*$" since $r2$ is a for-each-row rule. The $trigger$ labels on the remaining arcs are "1" since the other rules are for-each-statement and noninterruptable. $Forward$ labels are all empty since the rules are noninterruptable, while $Backward$ labels contain all the operations of the triggering rule's action.

Figure 7(b) shows the triggering graph when all rules are interruptable and after rules. The $trigger$ label on arc ($r0$,$r2$) is "$*$" since $r2$ is for-each-row. The $trigger$ label on arc ($r0$,$r4$) is "$*$" since the triggering operation of $r4$ (i.e., $+C$) is embodied in a loop node of $P0$. The $trigger$ label on arc ($r1$,$r3$) is "2" since the triggering operation of $r3$ (i.e., $-C$) occurs twice in $Action(r1)$. The $Trigger$ labels on the remaining arcs are "1" since triggering operations occur only once. The $Forward$ label on arc ($r0$, $r2$) contains $+B$ and $+C$ since the triggering operation of $r2$ (i.e., $+D$) is executed in $s2$, $+B$ $+C$ are respectively executed in $s3$ and $s4$, and finally, $s2$ precedes $s3$ and $s4$ in $P0$. It does not contain $+D$ because $r2$ is an after rule. Symetrically, the $Backward$ label on arc ($r0$, $r2$) contains $+D$ and $+A$. Note that $Forward$ and $Backward$ labels on arc ($r1$, $r3$) contain both $-C$ because $-C$ is executed in two distincts SQL statements of $Action(r1)$. Also note that $Forward$ and $Backward$ labels on arc ($r0$, $r4$) contain both $+B$ and $+C$ since they are executed in a loop node.

4.3 Execution Graph

Let Γ be a set of rules, and R a subset of Γ. Suppose that R represents the initial set of triggered rules at a given rule processing point. Then, we model the entire rule processing initiated by R using an execution graph noted $\mathcal{G}_\Gamma^R$. This graph is composed of two parts. The first part is the subgraph of

$\mathcal{G}_\Gamma$ that contains every path starting at any rule of R. In the second part there is a root node, noted *root*, and, for each rule r of R, an arc connects *root* to r. The $Trigger$ label on this arc is "1" if r is for-each-statement and "$*$" otherwise. The $Backward$ and $Forward$ labels of the arcs starting at the root node are irrelevant.

Example 4.3 Take the set of rules of Example 4.2 and the set $R = \{\ r1\}$. Then the corresponding execution graph contains rules $r1$ and $r3$ plus the labelled arc $(r1, r3)$ and a root node $root$ connected to $r1$. As $r1$ is for-each-statement, the $Trigger$ label on arc $(root, r1)$ is "1".

We shall only consider acyclic executions graphs. Cyclic execution graphs are first reduced to their strongly connected components. Then priorities, and labelled triggering arcs between connected components need to be constructed using specific construction rules. Due to space limitation, we do not present this construction in this paper.

The next definitions will be useful. Given an execution graph $\mathcal{G}_\Gamma^R$, and a node r:

- An *execution path for* r is a path starting at the root node and ending at r.
- $Ancestor_r$ is the set that contains all the nodes occuring in the execution paths for r excepted r.
- $Reachable_r$ is the set of nodes r' such that r in $Ancestor_{r'}$.
- $Predecessor_r$ is the set of nodes r' such that (r', r) is an arc of $\mathcal{G}_\Gamma^R$.
- $Successor_r$ is the set of nodes r' such that (r, r') is an arc of $\mathcal{G}_\Gamma^R$.

5 Rule Execution Analysis

The rule analysis is parametrized by the rule processing behaviour (iterative or recursive), and takes into account a static order between rules. Given an initial set of triggered rules R, our rule analysis aims to deduce for any rule r:

1. how many times r can be executed during the rule processing initiated by R.
2. if r is processed more than once, which database operations appearing in $Conflict(r)$ may occur between two consecutive executions of r.

Example 5.1 Take rules $r1$, $r2$,$r3$, $r4$ of Example 4.2. We illustrate two cases:
Iterative case: All rules are noninterruptable and the rule processing behaviour is iterative. We assume a partial static order between the rules: $r0 \prec r4$, $r4 \prec r1$, $r4 \prec r2$ and $r2 \prec r3$. If the rule processing is initiated by $R = \{r0,\ r4\}$, then the corresponding execution graph is given in Figure 8(a).
Recursive case: All rules are interruptable and the rule processing behaviour is recursive. Suppose that the rule processing is initiated by $R = \{r0\}$ and there is no static ordering between rules. Then the corresponding execution graph is given in Figure 8(b).

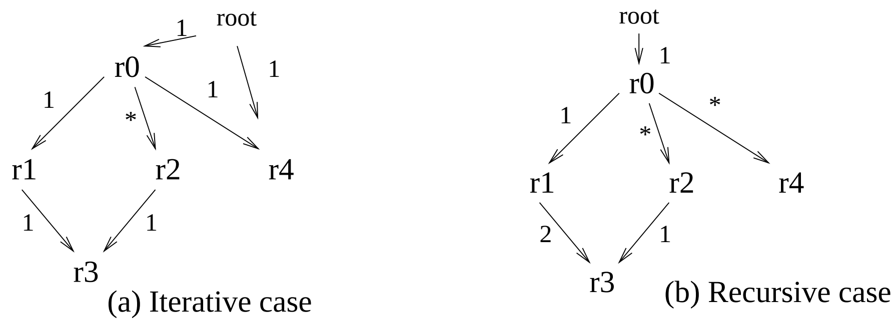


Figure 8: Execution graphs

5.1 Auxiliary Information

We consider a set of rules Γ, an initial set of triggered rules R, a rule processing behaviour (iterative or recursive), and a static ordering beetwen rules.

Maximal set of triggered rules : $Trigger\_set_R$ is the set that contains all the rules in the corresponding execution graph $\mathcal{G}_\Gamma^R$.

$$Trigger\_set_R = Reachable_{root}$$

Maximal set of executed operations : $Can\_perform_R$ is the set of data modification operations in all the rules in $\mathcal{G}_\Gamma^R$.

$$Can\_perform_R = \{op \mid \exists r \in Trigger\_set_R, op \in Performs(r)\}$$

Proposition 5.1 Let Γ be a set of rules, R an initial set of triggered rules, r a rule of Γ and op an operation. If $r \notin Trigger\_set_R$ (resp. $op \notin Can\_perform_R$) then r (resp. op) can never be executed during a rule processing when R is the initial set of triggered rules.

Example 5.2 Take the iterative and recursive cases of Example 5.1 and their corresponding execution graphs. In both cases, $Trigger\_set_R$ = $\{$ $r0$, $r1$, $r2$, $r3$, $r4\}$ and $Can\_perform_R$ = $\{+A$, $+D$, $+B$, $+C$, $-D$, $-C$, $-A\}$.

$Trigger\_set_R$ and $Can\_perform_R$ enable to prune rules from the triggering graph. However, for a finer analysis of the interactions beetwen the rules, we need to separately handle the iterative and recursive rule processing behaviours.

5.2 Iterative Rule Processing Behaviour

We first define the preceding rule set of a rule r, which intuitively represents the set of the triggered rules that are necessarily executed before r.

Preceding rule set of a rule: Given a rule r of $\mathcal{G}_\Gamma^R$, the preceding rule set of r is the set $P_R(r)$ recursively defined as follows:

1. The root node of $\mathcal{G}_\Gamma^R$ is in $P_R(r)$.

2. if $\exists r', r' \prec r$ and $\forall r'' \in Predecessor_{r'}$ $(r'' \in P_R(r))$ then $r' \in P_R(r)$.

3. if $\exists r', \forall r'' \in Predecessor_r - \{r'\}$ $(r' \in P_R(r''))$ then $r' \in P_R(r)$.

4. if $\exists r', r'$ dominates[2] r and $\forall r'' \in predecessor_{r'}$ ($r'' \in P_R(r')$ and $Trigger$ label of (r'', r') = "$*$" implies $r' \prec r$) then $r' \in P_R(r)$.

5. If $\exists r' \in Ancestor_r$ and $\exists r''$, r'' dominates r, $r'' \in P_R(r)$, and $\forall r''' \in Reachable_{r''} \cap Ancestor_r$ $(r''' \prec r)$ then $r' \in P_R(r)$.

6. if $\exists r'$ and $\exists r''$, $r' \in P_R(r'')$ and $r'' \in P_R(r)$ then $r' \in P_R(r)$

7. only rules satisfying items 1, 2, 3, 4, 5, or 6 are in $P_R(r)$.

Example 5.3 Consider the iterative case of Example 5.1. Using item 2, $r0 \in P_R(r4)$. Using item 4, $r0 \in P_R(r1)$, $r0 \in P_R(r2)$ and $r0 \in P_R(r3)$. Then, using item 2, $r_4 \in P_R(r2)$ and $r_4 \in P_R(r1)$. Next, using item 3, $r_4 \in P_R(r3)$. Finally, using item 2, $r2 \in P_R(r3)$.

Preceding operation set of a rule: Given a rule r of $\mathcal{G}_\Gamma^R$, the preceding operation set of r is the set $OP_R(r)$ defined as follows. An operation *op* is in $OP_R(r)$ iff:

1. *op* is in $Conflict(r) \cap Can\_perform_R$ and,

2. if $\forall r' \in Trigger\_set_R$, *op* $\notin Performs(r')$ or $r' \in P_R(r)$.

Succeeding operation set of a rule: Given a rule r of $\mathcal{G}_\Gamma^R$, the succeeding operation set of r is the set $OS_R(r)$ defined as follows. An operation *op* is in $OS_R(r)$ iff:

1. *op* is in $Conflict(r) \cap Can\_perform_R$ and,

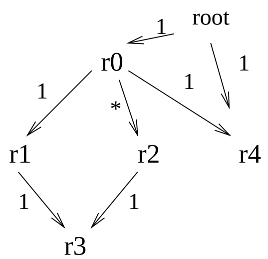
2. if $\forall r' \in Trigger\_set_R$, *op* $\notin Performs(r')$ or $r \in P_R(r')$.

Proposition 5.2 Let a set of rules be defined with an iterative rule processing behaviour, and R an initial set of triggered rules. Let r be a rule of $Trigger\_set_R$, and $OP_R(r)$ (resp. $OS_R(r)$) its preceding (resp. succeeding) operation set. If *op* is in $OP_R(r)$ (resp. in $OS_R(r)$), there is no possible execution of rules initiated by R such that *op* executes after (resp. before) or during an execution of r.

Corollary 5.1 Given a rule r, the maximal set of operations in $Conflict(r)$ which may execute after an execution of r is a subset of $Conflict(r) \cap (Can\_perform_R - OP_R(r))$, and the maximal set of operations in $Conflict(r)$ which may execute before an execution of r is a subset of $Conflict(r) \cap (Can\_perform_R - OS_R(r))$.

[2] A node n dominates a node $n' \neq n$ if all execution paths for n' contain n [ASU86] (chapter 10)

Example 5.4 Figure 9 shows the resulting $OP_R(r)$ and $OS_R(r)$ sets for each rule r of Example 5.1 in the iterative case. For instance, $+A$ and $+D$ are in in $OP_R(r2)$ since they are only executed by $r0$ which is in $P_R(r2)$. $OS_R(r2)$ = $\{-A\}$ since $-A$ is only executed by $r3$ and $r2$ is in $P_R(r3)$. We also give for each rule r, the maximal set of rules that may precede (resp. may follow) r. They are respectively called $Max\_precede(r)$ and $Max\_follow(r)$. For instance, $Max\_follow(r2)$ is { $+A$, $+D$, $-D$, $-A\}$ - $OP_R(r2)$= $\{-D, -A\}$, and $Max\_precede(r2)$ is { $+A$, $+D$, $-D$, $-A\}$ - $OS_R(r2)$ = $\{+A, +D, -D\}$.



(a) Execution graph

	OP	OS	Max_follow	Max_precede
r0	{ }	{-A}	{+A, -A, +B}	{+A, +B}
r1	{+A, +B}	{}	{-A}	{+A, +B, -A}
r2	{+A, +D}	{-A}	{-D, -A}	{-D, +D, +A}
r3	{+D, +C}	{}	{-D, -C}	{-D, -C, +C, +D}
r4	{+D, +C}	{-C, -D}	{-D, -C}	{+C, +D}

(b) computed information

Figure 9: Iterative rule processing

Maximal execution number for a rule: Given a rule r , its maximal execution number $N_R(r)$ in $\mathcal{G}^R_\Gamma$ is an element of $\boldsymbol{N} \cup \{*\}$ recursively defined as follows:

1. if r is the root node then $N_R(r)$ = 1.
2. if there is an arc (r', r) with a $Trigger$ label "$*$" then $N_R(r)$ = "$*$"[3],
3. let $Q_1 = Predecessor_r \cap P_R(r)$ and $Q_2 = Predecessor_r - Q_1$, if $\exists r' \in Q_2$ s.t $N_R(r')$ = "$*$" then $N_R(r)$ = "$*$". Otherwise[4]
$$N_R(r) = q_1 + \sum_{r' \in Q_2} N_R(r')$$
with $q_1 = 0$ if $Q_1 = \emptyset$ and $q_1 = 1$ otherwise.

$N_R(r)$ = "$*$" indicates that r can be executed zero or more times.

Example 5.5 Take the iterative case of Example 5.1 and use the precedence rule sets obtained in Example 5.3. Using item 3, we obtain $N_R(r0)$ = 1, $N_R(r1)$ = 1, $N_R(r4) = 1$ Using item 2, $N_R(r2)$= "$*$'. Finally, using item 3, $Q1$ = { $r2$} and $Q2$ ={ $r1$}; thus, $q1$ =1 and $N_R(r3)$ = $q1 + N_R(r1)$ = 2.

Proposition 5.3 Let a set of rules be defined with an iterative rule processing behaviour, and R an initial set of triggered rules. Let r be a rule of $Trigger\_set_R$, if $N_R(r) \neq *$ then no possible execution of the rules initiated by R is such that r is executed more than $N_R(r)$ times.

[3] $N_R(r)$ = "$*$" indicates that r can be executed zero or more times.
[4] if one of the $N(r')$ is "$*$", the sum is "$*$"

5.3 Recursive Rule Processing Behaviour

When a rule r executes, the order in which the rules triggered by r are executed is dependent on the relative order of execution of their triggering operations in the action of r. This order is inferred from the *Forward* and *Backward* labels on the arcs of the triggering graph. It is complemented by taking into account the static order between rules.

More formally, given a rule r, and two rules r' and r'' ($r' \neq r''$) in $Successor_r$, then r' precedes r'' wrt r (noted $r' <_r r''$) iff one of the following items holds:

1. r is not interruptable and $r' \prec r''$.
2. r is interruptable, $Triggered\_by(r') \in Backward(r, r'')$, and $Triggered\_by(r'') \notin Backward(r, r')$.
3. r is interruptable, r' and r'' are both before (resp. after) rules, the $Trigger$ labels of both (r, r') and (r, r'') are "1", $Triggered\_by(r') = Triggered\_by(r'')$, and $r' \prec r''$.

Example 5.6 Take rules $r0, r1, r2$ and $r4$ of Example 5.1 in the interruptable case (see Figure 7(b)). The order relationship $<_{r0}$ is deduced from the $Backward$ and $Forward$ labels as follows: by item 2, $r1 <_{r0} r2$ since the triggering operation of $r1$ (i.e., $+A$) is in $Backward(r0, r2)$ and the triggering operation of $r2$ (i.e., $+D$) is not in $Backward(r0, r1)$. Similarly, $r1 <_{r0} r4$ and $r2 <_{r0} r4$.

Maximal execution number of a rule: Given a rule r , its maximal execution number $N_R(r)$ in $\mathcal{G}_\Gamma^R$ is recursively defined as follows:

1. if r is the root node then $N_R(r)$ = 1
2. if $\exists r' \in Predecessor_r$ s.t $N_R(r') =$ "$*$" or the trigger label of (r', r) is "$*$" then $N_R(r) =$ "$*$". Otherwise

$$N_R(r) = \sum_{r' \in Predecessor_r} (N_R(r') * \text{trigger\_label\_of}(r', r))$$

Proposition 5.3 also holds with the above definition of $N_R(r)$.

Example 5.7 Consider the recursive case of Example 5.1(see Figure 8(b)). By item 2, $N_R(r0)$ =1, $N_R(r1)$ =1, $N_R(r4)$ = "$*$" $N_R(r2) = N_R(r3)$ = "$*$".

Preceding rule set of a rule: Given a rule r of $\mathcal{G}_\Gamma^R$, the preceding rule set of r is the set $P_R(r)$ recursively defined as follows: let r' be a rule of $\mathcal{G}_\Gamma^R$

1. if $\exists r''$, $Predecessor_{r'}$ = $Predecessor_r$ = $\{r''\}$, $N_R(r'')$ = 1 and $r' <_{r''} r$ then $r' \in P_R(r)$.
2. if $\forall r'' \in Predecessor_{r'}, \forall r''' \in Predecessor_r$ ($r'' = r'''$ and $N_R(r'')$ = 1 and $r' <_{r''} r$) or ($r'' \neq r'''$ and $r'' \in P_R(r'''))$ then $r' \in P_R(r)$.
3. if $\forall r'' \in Predecessor_{r'}, r'' \in P_R(r)$ then $r' \in P_R(r)$.

4. if $\exists r''$ such that $r' \in P_R(r'')$ and $r'' \in P_R(r)$ then $r' \in P_R(r)$.
5. only rules satisfying items 1, 2, 3, or 4 are in $P_R(r)$.

Example 5.8 Consider the recursive case of Example 5.1(see Figure 10(a)). By item 1 and using the order relationship $<_{r0}$ between $r1$, $r2$ and $r4$ obtained in Example 5.6, we have $r1 \in P_R(r2)$, $r1 \in P_R(r4)$, and $r2 \in P_R(r4)$. By item 3, $r3 \in P_R(r4)$.

Preceding operation set of a rule: Let r be a rule of $\mathcal{G}_\Gamma^R$. Let G_1, G_2 be two subgraphs of $\mathcal{G}_\Gamma^R$ defined as follows: for each traversal path p of $\mathcal{G}_\Gamma^R$, if p contains some rule r' of $P_R(r)$, then p is in G_1 else, if p contains r then p is in G_2. Let G_3 be the set of rules that are neither in G_1 nor in G_2. Then, the preceding operation set of r is the set $OP_R(r)$ defined as follows. An operation *op* in $Conflict(r) \cap Can\_perform_R$, is in $OP_R(r)$ iff:

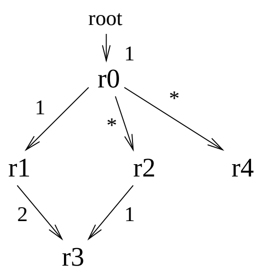
1. $\forall r'' \in G_3$, *op* $\notin Performs(r'')$ and,
2. $\forall (r'', r''') \in G_2$, *op* $\notin Forward(r'', r''')$ and,
3. $\forall r'' \in Reachable_r \cup \{r\}$, *op* $\notin Performs(r'')$ and,
4. if $\exists r'' \in G_2$ such that *op* $\in Performs(r'')$ then G_2 contains a distinguished traversal path p = $root(= \rho_0), \cdots, \rho_n, r(= \rho_{n+1}), r_1, \cdots, r_m$, $n \geq 0$ and $m \geq 0$, such that :
   (a) $\forall i \in 1 \cdots n$, if *op* $\in Backward(\rho_i, \rho_{i+1})$ then $\forall j \in 0 \cdots i$, $N_R(\rho_j) = 1$,
   (b) for each traversal $p' \neq p$ in G_2, there exists $i \in 1 \cdots n$ such that $p' = root(= \rho_0), \cdots, \rho_i$, $s_1, \cdots, s_{k-1}, r(= s_k)$, ..., with $(1 \leq k)$, and for $j \in 0 \cdots k$, *op* $\notin Backward(s_j, s_{j+1})$, and $\rho_{i+1} \in P_R(s_1)$.

Example 5.9 Consider the recursive case of Example 5.1(see Figure 10(a) and the precedence rule sets obtained in Example 5.8), we compute $OP_R(r1)$. First, $Conflict(r1) \cap Can\_perform_R$ = { +A, -A, +B}. $G1 = \emptyset$ since no rules are in $P_R(r1)$ (see Example 5.8), $G2$ contains the path $r0, r1, r3$, and $G3$ contains $r4$. $-A$ is not in $OP_R(r1)$ because item 3 does not hold. Indeed, $-A$ is executed by $r3$ which is triggered by $r1$, thus, $-A$ can be executed during an execution of $r1$. $+B$ is not in $OP_R(r1)$ because item 2 does not hold: $+B$ is in $Forward(r0, r1)$, thus, $+B$ can be executed after an execution of $r1$. $+A$ is only executed by $r0$ and satisfies items 1, 2, 3 and 4. Thus, $OP_R(r1)$ = $\{+A\}$. $-C$ is not in $OP_R(r3)$ because it is executed by $r1$ and $r2$ and both $r1$ and $r2$ trigger $r3$. Thus, $r3$ is executed between $r1$ and $r2$ and $-C$ cannot precede $r3$. By item 4, $G2$ contains two paths $(r0, r1, r3)$ and $(r0, r2, r3)$ and $-C$ is both in $Backward(r1, r3)$ and $Backward(r2, r3)$. Thus, item 4.b can never apply.

Succeeding operation set of a rule: Let r be a rule of $\mathcal{G}_\Gamma^R$. Let G_1, G_2 be two subgraphs of $\mathcal{G}_\Gamma^R$ such that: for each traversal path p of $\mathcal{G}_\Gamma^R$, if p contains some rule r' such that $r \in P_R(r')$, then p is in G_1 else if p contains r then p is in G_2. Let G_3 be the set of rules that are neither in G_1 nor in G_2. Then, the succeeding operation set of r is the set $OS_R(r)$ defined as follows. An operation *op* in $Conflict(r) \cap Can\_perform_R$ is in $OS_R(r)$ iff:

1. $\forall r'' \in G_3$, $op \notin Performs(r'')$ and,
2. $\forall (r'', r''') \in G_2$, $op \notin Backward(r'', r''')$ and,
3. $\forall r'' \in Reachable_r \cup \{r\}$, $op \notin Performs(r'')$ and,
4. if $\exists r''$ in G_2 such that $op \in Performs(r'')$ then G_2 contains a distinguished traversal path p = $root(=\rho_0), \cdots, \rho_n, r(=\rho_{n+1}), r_1, \cdots, r_m$, $n \geq 0$ and $m \geq 0$, such that :
   (a) $\forall i \in 1 \cdots n$, if $op \in Forward(\rho_i, \rho_{i+1})$ then $\forall j \in 0 \cdots i$, $N_R(\rho_j)$ = 1,
   (b) for each traversal $p' \neq p$ in G_2, there exists $i \in 1 \cdots n$ such that p' = $root(=\rho_0), \cdots, \rho_i$, $s_1, \cdots, s_{k-1}, r(=s_k)$, ..., with ($1 \leq k$), and for $j \in 0 \cdots k$, $op \notin Forward(s_j, s_{j+1})$, and $s_1 \in P_R(\rho_{i+1})$.

Proposition 5.2 and Corollary 5.1 also hold with the definitions of $OP_R(r)$ and $OS_R(r)$ in the recursive case.



(a) Execution graph

	OP	OS	Max_follow	Max_precede
r0	{}	{}	{+A, -A, +B}	{+A, -A, +B}
r1	{+A}	{+B}	{+B, -A}	{+A, -A}
r2	{+A, +D, -D}	{}	{-A}	{-D, +D, +A, -A}
r3	{-D}	{+C}	{-C, +C, +D}	{-D, -C, +D}
r4	{-D, +D, -C}	{}	{+C}	{-D, -C, +C, +D}

(b) computed information

Figure 10: Recursive rule processing

Example 5.10 Figure 10(b) shows the resulting $OP_R(r)$ and $OS_R(r)$ sets for each rule r in the execution graph of Figure 10(a). We also give for each rule r, the maximal set of rules that may precede (resp. may follow) r. Given these sets, we can derive for instance that $-A$ may execute between two executions of $r2$ ($\{-A\}$ = $Max\_precede(r2) \cap Max\_follow(r2)$). This result is quite different from the iterative case where only $-D$ can execute between two executions of $r2$. Moreover, in the recursive case, $-A$ is performed by $r3$ which may execute several times ($N_R(r3)$ = $*$), while in the iterative case, $-D$ is executed only once by $r1$.

6 Global Analysis of a Transaction and Rules

Given a transaction program $\mathcal{T}$ and a set of rules Γ, our goal is to compute the following indications :

$Triggered\_rules$ is the subset of Γ which omits rules of Γ that will never be triggered by $\mathcal{T}$.

$Conflicting\_operations(r)$ is a subset of the operations in $Conflict(r)$ which omits operations that can never occur between two consecutive executions of r within $\mathcal{T}$.

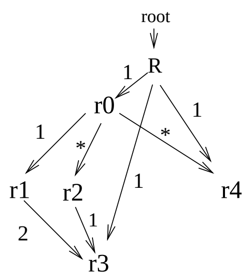
$Executions\_number(r)$ gives an upperbound on the number of possible execution of r within $\mathcal{T}$.

We distinguish two cases in the analysis:

6.1 SQL-statement, tuple + recursive

The first one consists of programs with an SQL-statement or tuple rule processing granularity. In this case, a transaction program can be regarded as the action of a specific rule, noted R, which initiates the rule processing. This specific rule is interruptable. Thus, we can model $\mathcal{T}$ and Γ using an execution graph in which the label of the arc $(root, R)$ is "1". Using the analysis of Section 5, we have:

$$\begin{array}{lcl} \text{Triggered\_rules} & = & Trigger\_set_R \\ \text{Executions\_number(r)} & = & N_R(r) \\ \text{Conflicting\_operations(r)} & = & Conflict(r) \cap (Can\_perform_R - (OP_R(r) \cup OS_R(r))) \end{array}$$



(a) execution graph

	OP	OS	Exec_number	Conflicting_operations
r0	{-B}	{}	1	{+A, -A, +B}
r1	{+A, -B}	{+B}	1	{-A}
r2	{+A, +D, -D}	{}	*	{-A}
r3	{-D}	{}	*	{+D, -C, +C}
r4	{-D, +D}	{}	*	{-C, +C}

(b) computed information

Figure 11: Global Analysis of T in the recursive case

Example 6.1 Take the interruptable rules of Example 5.1 and assume a recursive behaviour. Let T be the following transaction program $-B$;$-C$; $+C$;. The resulting execution graph is shown in Figure 11. The resulting OP, OS, $Executions\_number$, $Conflicting\_operations$ informations for each rules are shown in Figure 11.

6.2 SQL-statement, tuple, delayed + iterative

This case consists of programs with an SQL-statement, tuple or delayed rule processing granularity, and an iterative rule processing behaviour. Unlike the previous case, a transaction program first needs to be translated into a rule processing sequence.

Rule processing sequence: Given $\Phi_{\mathcal{T}}$ the simplified flow graph for $\mathcal{T}$, the rule processing sequence associated with $\Phi_{\mathcal{T}}$, noted Φ', is defined as follows. If the rule processing granularity is SQL-statement or tuple, Φ' is derived from $\Phi_{\mathcal{T}}$ by decomposing each simple node n of $\Phi_{\mathcal{T}}$ into a *before* node and an *after* node that both contain the statement in n. The $before$ node represents the processing point for before rules triggered at n, while the $after$ node represents the processing point for after rules. The $before$ node just precedes the $after$ node in Φ'.

If the rule processing granularity is delayed, Φ' is the sequence derived from $\Phi_{\mathcal{T}}$ in two steps. First, each loop node containing a *chk* statement is completed with a *before* node. The *before* node derived from a node n in $\Phi_{\mathcal{T}}$ is a simple node that contains the statements of n. This additive node is placed just before n in the sequence. It represents the processing point initiated by the operations occuring in n plus those that precede n in $\Phi_{\mathcal{T}}$ and succeed to the previous chk statement in $\Phi_{\mathcal{T}}$. Second, the nodes which contain no chk statement are discarded.

We shall use the function $Operation$ which takes a node in Φ' and returns the set of data modification operations for this node. Given an element n of Φ', n' its predecessor a data modification operation *op*, is in $Operation(n)$ iff either

1. the rule processing granularity is SQL-statement or tuple and *op* is in some statement contained in n, or
2. the rule processing granularity is delayed, n is a simple node, *op* is in some statement s s.t. $s \in n$, or there exists some node m in $\Phi_{\mathcal{T}}$ such that $s \in m$ and m is beetwen n' and n in $\Phi_{\mathcal{T}}$ and n' just precedes n in Φ', or
3. the rule processing granularity is delayed, n is a loop node, and *op* is in n.

Example 6.2 Take the program $P0$ in Figure 6(c). The resulting rule processing sequence is $n1$,$n2$,$n3$ where $n1$ and $n2$ are simple nodes, $n3$ is a loop node, $Operation(n1)$ = $\{+A\}$, $Operation(n2)$ = $\{+D, +B, +C\}$ and $Operation(n3)$ = { +B, +C}.

Interactions between transaction and rules:

We define $R(n)$, $n \in \Phi'$, as the initial set of triggered rules for the simple node n.

- If the rule processing granularity is SQL-statement or tuple, and n is an $after$ node(resp. a $before$ node), $R(n)$ = $\{r \mid Triggered\_by(r) \in Operation(n)$, and r is an after rule (resp. a before rule)$\}$
- If the rule processing granularity is delayed, $R(n) = \{r \mid Triggered\_by(r) \in Operation(n)\}$

Then, we define $Triggered(n)$, $n \in \Phi'$, as follows:

1. If the rule processing granularity is delayed, $Triggered(n) = Trigger\_set_{R(n)}$
2. If the rule processing granularity is SQL-statement or tuple and n is a loop node,

$$Triggered(n) = \bigcup_{op \in n} (Trigger\_set_{RA} \cup Trigger\_set_{RB})$$

   where RA (resp. RB) is the set of rules given by: $\{r \mid Triggered\_by(r) = op$ and r is a before rule (resp. an after rule)$\}$.

3. If the rule processing granularity is SQL-statement and n is an $after$ or $before$ node, $Triggered(n) = Trigger\_set_{R(n)}$

We redefine $Max\_follow(n, r)$ and $Max\_precede(n, r)$, n in Φ' and r in Γ, as:
$Max\_follow(n, r) = (Conflict(r) \cap Can\_perform_{R(n)}) - OP_{R(n)}(r)$
$Max\_precede(n, r) = (Conflict(r) \cap Can\_perform_{R(n)}) - OS_{R(n)}(r)$

We are now able to compute our final indications.

Triggered_rules $= \bigcup_{n \in \Phi'} Triggered(n)$

Execution_number(r), r in Γ, is defined as follows:

1. if $r \notin Triggered\_rules$ then $Execution\_number(r) = 0$
2. if $\exists n \in \Phi'$ s.t. $r \in Triggered(n)$ and n is a loop node or $N_{R(n)}(r) =$ "$*$" then $Execution\_number(r) =$"$*$".
3. and otherwise, $Execution\_number(r) = \sum_{n \in \Phi'} N_{R(n)}(r)$

Conflicting_operation(r), r in Γ, is defined as follows: Given $\Phi' = n_1, ..., n_k$ be the processing sequence for $\mathcal{T}$, an operation op is in $Conflicting\_operation(r)$ iff $op \in Conflict(r)$, $Execution\_number(r) \notin \{0, 1\}$, and one of the following assertions holds:

1. $\exists i \in 1..k, n_i$ is a loop node, $r \in Triggered(n_i)$ and $op \in Can\_perform(n_i) \cup Operation(n_i)$, or
2. $\exists i \in 1..k$, n_i is a simple node, and $N_{R(n_i)}(r) \notin \{0, 1\}$, and $op \in Can\_perform(n_i)$ - $(OP_{R(n_i)}(r) \cup OS_{R(n_i)}(r))$, or
3. $\exists i, j \in 1..k$, $r \in Triggered(n_i) \cap Triggered(n_j)$ and
$op \in Max\_follow(n_i, r) \cup Max\_precede(n_j, r)$
$\cup \left(\bigcup_{i' \leq l \leq j'} Operation(n_l) \bigcup_{i \leq l \leq j} Can\_perform(n_l) \right)$
with $i' = i$ if n_i is a $before$ node, else $i' = i + 1$ and $j' = j - 1$ if n_j is a $before$ node, else $j' = j$.

6.3 Analysis complexity

Given a transaction program and a set of rules Γ, a naive algorithm that computes the subset of (possible) triggered rules and, for each of them, its maximal execution number and its conflicting operation set, runs in $O(n^3)$ time where n is the number of rules in Γ. Indeed, for each rule r, a naive algorithm builds all paths containing r and tests each node in these paths in $O(n^2)$ time. In the case of SQL-statement, or tuple, or delayed granularity with iterative rule processing behaviour, the algorithm is applied to each node of Φ'.

7 Related Work

The analysis of rules for discovering repeatitive evaluations was only previously proposed in [FRS93]. This analysis was done in the framework of deductive rules whose execution corresponds to for-each-row and/or for each statement noninterruptable rules.

Other papers have proposed static rule analysis techniques for predicting the behaviour of rules in order to determine if a rule set satisfies the termination and/or the confluence properties. These techniques analyse rules independantly from the triggering transaction. The method presented in [AHW95] is developed in the context of the Starburst system [WC96] which only considers for-each-statement rules and executes them with a delayed rule processing granularity and an iterative behaviour.

In [vdVS93] the analysis of rule behaviour is performed in the context of active object oriented databases. Rule actions are restricted to perform data modification operations on data items returned by the conditions of the rules. Deletions and insertions seem to be disallowed. This analysis essentially focuses on for-each-row and for-each-statement rules.

In [BCP95], the rule analysis combines the information provided by a triggering graph and an activation graph that represents the effect of each rule action on conditions of other rules. In [BW94], a "propagation" algorithm is proposed to generate such activation graph. All these methods are restricted to noninterruptable, for-each-statement rules with an iterative rule processing behaviour. Our analysis tool does not consider activation graphs, but we expect it can be complemented by taking such information into account.

A rather different approach to rule analysis is used by [KU94] where ECA rules are first translated into a *term rewriting systems*, and then existing analysis techniques for termination and confluence of these systems are applied. However, this rule analysis does not take into account neither the rule processing behaviour nor the rule execution granularity.

8 Conclusion

We have presented algorithms that analyze the behaviour of a transaction and a set of rules triggered by this transaction in order to derive: (i) if a given rule is processed more than once, and (ii) the relevant database changes that may occur between two consecutive executions of the rule. Such analysis techniques are essential for optimizing the processing of active database transactions by eliminating costly redundant computations using either caching techniques [FRS93] or materialized views [RSS96]. Redundant computations of rules are potentially frequent in active database applications because existing products use an SQL statement rule processing granularity and users tend to prefer to use instance-oriented rules.

Although the problem studied in this paper has not been studied before for ECA rules, our rule analysis techniques have in themselves several salient features. First, they take into account fundamentals parameters of ECA rule execution semantics: rule execution granularity (for-each-row, for-each-statement), rule processing granularity (tuple, SQL-statement, and delayed), and rule processing behaviour (iterative noninterruptable, recursive interruptable and noninterruptable), which taken together yield several combinations of rule execution semantics. By comparison, other existing ECA rule analysis techniques essentially developed for studying the termination and confluence properties of a set of rules, only consider an iterative rule processing behaviour. However, the recursive case is important since this behaviour is adopted by all commercial active relational systems and the forthcoming SQL3 standard. Our generality produces an increased complication of the rule analysis. However, our rule analysis is inexpensive, since algorithms are running at worst in $O(n^3)$, using a very naive implementation where n is the number of rules triggered by a transaction.

As a second feature, an original aspect of our techniques is to analyze the behaviour of rules with respect to the structure of a triggering transaction. This is a major decision because the structure of the triggering transaction (i) plays a direct role in the occurence of repeated rule executions, and (ii) determines a maximal set of rules that can possibly be triggered (usually, a small set). Furthermore, since redundant calculations can be detected on a transaction basis, transactions can be separately optimized using caching techniques. Similarly, we also analyze the structure of rule actions which can be structured programs (a frequent situation in our experience).

Last, the rule execution semantics framework considered in this paper, which also includes a local consumption of events at evaluation time and an immediate C-A coupling mode, enables to capture a very large class of existing active relational systems.

As a future work, we first envision to generalize our rule analysis technique to cope with multiple rule granularities and multiple rule processing behaviours in the same system. Next, we believe that our rule analysis technique can provide useful insight to the study of termination and confluence properties of active transactions.

Acknowledgements

We wish to thank Bobbie Cochrane and Ravi Raj Mahendru for the fruitful discussions we had during the preparation of this paper. We also thank Maja Matulovic and Dimitri Tombroff for their helpful comments on earlier versions of the paper.

References

[AHW95] A. Aiken, J. M. Hellerstein, and J. Widom. Static Analysis Techniques for Predicting the Behavior of Active Database Rules. *ACM-TODS Transactions on Database Systems*, pages 3–41, March 1995.

[ASU86] A. Aho, R. Sethi, and J. Ullman. *Compilers Principles, Techniques, and Tools*. Addison-Wesley Publisher Company, 1986.

[BCP95] E. Baralis, S. Ceri, and S. Paraboschi. Improved Rule Analysis by Means of Triggering and Activation Graphs. In *Second International Workshop, RIDS' 95*, pages 165–181, Athens, September 1995.

[BW94] E. Baralis and J. Widom. An Algebraic Approach to Rule Analysis in Expert Database Systems. In *20 th Conf. on Very Large Data Bases*, Santiago- Chile, September 1994. LNCS.

[Coc96] B. Cochrane. Private Communication, Jan. 1996.

[FRS93] F. Fabret, M. Régnier, and E. Simon. An Adaptative Algorithm for Incremental Evaluation of Production Rules. In *Proc. International Conference on Very Large Databases*, Dublin, Ireland, Aug. 1993.

[FT95] P. Fraternali and L. Tanca. A Structured Approach for the Definition of the Semantics of Active Databases. *ACM Transactions On Database Systems*, December 1995.

[ISO95] ISO-ANSI Working Draft: Database Language SQL (SQL3) DBL:YOW-006 and X3H2-95-086, 1995.

[KU94] A.P. Karadimce and S.D Urban. Conditional Term Rewriting as a Formal Analysis of Active Database Rules. In *4th Int'l Workshop on Research Issues in Data Enginnering*, Houston, Feb. 1994.

[RSS96] K. A. Ross, D. Srivastava, and S. Sudarshan. Materialized View Maintenance and Integrity Constraint Checking: Trading Space for Time. In *In Proc. of the ACM SIGMOD International Conference*, june 1996.

[SKD95] E. Simon and A. Kotz-Dittrich. Promise and Realities of Active Database Systems. In *Proc. International Conference on Very Large Databases*, Zurich, Switzerland, Sept. 1995.

[TPC95] TPC benchmark D , Standard Specification, May 1995.

[vdVS93] L. van der Voort and A. Siebes. Termination and Confluence of Rule Execution. In *Proc. 2nd Int'l Conf. on Inflormation and Knowledge Management*, Nov. 1993.

[WC96] J. Widom and S. Ceri. *Active Database Systems: Triggers and Rules for Advanced Database Processing*. Morgan-Kaufmann, San Francisco, California, 1996.

Unité de recherche INRIA Lorraine, Technopôle de Nancy-Brabois, Campus scientifique,
615 rue du Jardin Botanique, BP 101, 54600 VILLERS LÈS NANCY
Unité de recherche INRIA Rennes, Irisa, Campus universitaire de Beaulieu, 35042 RENNES Cedex
Unité de recherche INRIA Rhône-Alpes, 46 avenue Félix Viallet, 38031 GRENOBLE Cedex 1
Unité de recherche INRIA Rocquencourt, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex
Unité de recherche INRIA Sophia-Antipolis, 2004 route des Lucioles, BP 93, 06902 SOPHIA-ANTIPOLIS Cedex

Éditeur
INRIA, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex (France)
ISSN 0249-6399