Optimizing Repetitive Computations of Database Triggers Within a Transaction

Françoise Fabret, Eric Simon

INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Optimizing Repetitive Computations of Database Triggers Within a Transaction

Françoise Fabret, Eric Simon

N° 2533

Avril 1995

PROGRAMME 1

ISSN 0249-6399

Optimizing Repetitive Computations of Database Triggers Within a Transaction

Françoise Fabret*, Eric Simon*

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

Rapport de recherche n° 2533 — Avril 1995 — 21 pages

Abstract: We study the problem of optimizing costly repetitive evaluations of database triggers within a transaction. We first show that well known incremental rule evaluation algorithms such as RETE or TREAT are inappropriate for that because they do not consider how repetitive triggerings of rules can be caused by the structure of transaction programs. Therefore, their decision of precomputing and caching some expressions in rule conditions for a later reuse can be erroneous. We assume that transaction programs are represented by their flow graph. We then propose an algorithm that, given a transaction's flow graph, and a set of triggers, constructs a compact data structure called a triggering graph. First, for each possible transaction execution, this graph indicates which rules may be triggered. Second, for every rule r capable of being triggered and fired several times, the graph represents the real "influence" of both the transaction and the rules on r. This provides the necessary information for deciding which subexpressions of r are most profitable to cache for the considered transaction.

Key-words: Active Databases, Transactions, Differential computation, Caching, Rule computation optimization.

(Résumé : tsvp)

* {Françoise.Fabret}{Eric.Simon}@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

Optimisation de l'Evaluation Répétitive des Règles Actives dans une Transaction

Résumé : Nous étudions le problème de l'optimisation de l'évaluation répétitive des règles au cours d'une transaction. Tout d'abord, nous montrons que les algorithmes classiques d'évaluation incrémentale des règles, tels RETE ou TREAT ne fournissent pas de solution appropriée car ils ne prennent pas en compte comment le déclenchement répétitif des règles peut être causé par la structure de la transaction. De ce fait, leurs décisions de précompiler et de mémoriser certaines expressions pour accélérer les évaluations ultérieures de la partie condition des règles peuvent être erronées. Nous supposons que les programmes des transactions sont représentés par leurs grapphe de flôt de données. Dans un deeuxième temps, nous proposons un algorithme qui, étant donnés le graphe d'une transaction et un ensemble de règles actives, construit un structure de données compacte appelée graphe de déclenchement. Tout d'abord, pour chaque exécution possible du programme d'une transaction, ce graphe indique quelles règles peuvent être déclenchées. De plus, il indique les règles qui peuvent être déclenchées plusieurs fois, et, pour chacune d'entre elles, il représente l'influence de la transaction et des autres règles. Ce qui fournit l'information nécessaire pour décider quelles expressions est-il profitable de mémoriser pour la transaction considérée.

Mots-clé : Bases de données actives, transactions, calcul différentiel, mémorisation, optimisation du calcul des règles.

1 Introduction

A production rule consists of an action that must be executed whenever a condition over the database holds. Usually, the action is a set of operations such as insertions, deletions, and updates. Executing a set of production rules proceeds by (i) evaluating rule's conditions against the database, (ii) choosing one rule whose condition is satisfied, (iii) executing the action of the selected rule, and repeating the cycle until a fixed-point is reached (if any). A rule instantiation whose condition holds in a given database state is called a *satisfying rule instantiation*.

A critical part of rule evaluation is the *match phase* (phase (i) above) because a rule condition can be evaluated more than once against the entire database, thereby causing costly redundant computations. There are two typical situations where a rule needs to be evaluated more than once. First, if a rule has an instance-oriented semantics (i.e., the rule is fired for one satisfying rule instantiation) then the condition of the rule is evaluated as many times as there are satisfying rule instantiations. Second, the graph of causal dependencies between rules may have cycles. A rule has a causal dependency with another rule if firing the first rule may trigger the other rule. A recursive rule is a special case where the cycle of causal dependencies is of length one (the rule has a causal dependency with itself).

To overcome this problem, incremental rule evaluation algorithms maintain state information across the execution of a rule program. More precisely, the idea is to precompute and cache subexpressions occuring in a rule's condition, and then incrementally maintain them when their operand data are updated by subsequent rule firings. The choice of the data to be cached and maintained depends on the *caching strategy*.

In most algorithms, the caching strategy is based on information that is purely local to a rule (we call them *local* strategies). For instance, RETE [For82] and TREAT [Mir87] use local caching strategies that only consider the pattern of rule conditions independently: RETE maintains the result of every selection and join involved in the condition of the rule whereas TREAT only maintains the result of selections and the result of rule conditions. As explained in [FRS93], these strategies are quite blind because they do not examine if caching an expression is profitable or not.

A *global* caching strategy has been proposed in [FRS93]. For any given rule r in a program, a heuristic-based algorithm decides which data to cache in terms of the kinds of dependencies that r has with the entire rule program. More precisely, the strategy, in the tradition of [PK82], [Pai86], and [PH87], consists of extracting subexpressions in a rule that (i) can be efficiently differentiated with respect to the changes induced by the entire rule program, and (ii) whose caching, most probably, avoids repeated calculations.

The important point is that all algorithms, whether they use a local or global strategy, optimize rules in a similar way: each rule of the rule base is analyzed and a single optimized, possibly compiled, version of the rule is generated (the optimized version makes use of the cached expressions). Then, any processing of the rule base uses the optimized versions of the rules. This scheme works well for production systems[1] where one seeks to optimize *one* processing of a rule program at a time,

[1] Almost all production rule systems developped in AI implement such a scheme with variants of RETE or TREAT

and several implementations of this scheme have been proposed for database rule systems [SLR88], [DWE89], [SZ91], [Han92].

In this paper, we argue that this scheme is however not appropriate for optimizing rules in active database systems. In these systems (see [WCD95] for a survey), a rule is triggered by the occurence of some specific *triggering event* associated with the rule. Events are initially generated by transaction executions. When an event occurs (generally, a database change), a set of *immediate* rules may be triggered. One rule is selected and processed, possibly causing new database changes which may trigger additional immediate rules. This process continues until all triggered rules have been processed. At the end of the transaction execution, a set of *deferred* rules may be triggered and executed. Thus, to each transaction execution is associated a *set* of rule program executions: a set of immediate rules at each event, and a set of deferred rules at the end of the transaction. We claim that one should optimize repetitive calculations over this global set of rule program executions instead of optimizing separately each rule program execution. Following this line, our goal is to develop a formal tool that describes how repetitive triggerings of rules can be caused by the structure of a transaction program. Using such a tool, a global caching strategy would then be able to take appropriate caching optimization decisions regarding the repetitive evaluation of triggers within a transaction.

In this paper, we assume that transaction programs are represented by their *flow graph* [ASU86] representing the flow of events in the program together with the programming control structures (conditional, loop, sequence) embedding these events. Quite different transaction programs may have the same flow graph. Our major contribution is to propose an algorithm that, given a transaction's flow graph, constructs a compact data structure called a triggering graph. First, for each possible transaction execution, this graph indicates which rules may be triggered. Second, for every rule r capable of being triggered *and* fired several times, the graph represents the influence of both the transaction and the rules on r. The later information is essential for a caching strategy in order to decide which subexpressions of r are most profitable to cache for the considered transaction. The information vehicled by a triggering graph is kept minimal by the use of optimization techniques that perform a detailed code analysis of the transaction and the rules.

The paper is organized as follows. Section 2 presents the active rule language considered throughout this paper. Section 3 sets the problems addressed by our work and give the basic requirements for our algorithm. Section 4 introduces our abstract notation for transactions, and formalizes the notion of triggering event. Section 5 presents our central data structure, the triggering graph, and optimization techniques that enable to simplify it. Section 6 describes the general algorithm that constructs a simplified triggering graph. Section 7 discusses how a triggering graph can be used by a global caching strategy, and how to adapt our construction of triggering graphs to ECA rule languages. Section 8 concludes.

2 The Active Rule Language

There is a large variety of active database rule languages that considerably vary both in their syntax and semantics [WCD95]. Most of the features and semantic details found in each language are not relevant for the purpose of this paper. Furthermore, it is not realistic to specify our algorithm in terms of all existing active rule languages. Therefore, we introduce a simple and concise notation for active rules with a semantics that both retains the essential notions used by our algorithm and facilitates the presentation of our results.

2.1 Syntax of Rules

An active rule is an expression of the form: if *condition* then *action*. For simplicity, we assume that the triggering events are implicit from the rule condition and action (i.e., there is no event part in a

rule definition). The condition part of the rule corresponds to a query over the database that produces a set of tuples. The action may consist of either (i) a rollback statement (in which case, the rule is said to be a *rollback* rule), or (ii) a sequence of data modification statements (the rule is called a *productive* rule).

In the action part of a productive rule, each element of the sequence specifies either that the tuples produced by the condition part must be inserted in a specified relation or that the tuples produced by the condition part must be deleted from a specified relation.

The condition part of any rule may refer to *delta relations* in addition to normal (also called *extensional*) database relations. There are two delta relations, ins_T and del_T, associated with each extensional relation T. These relations contain the tuples that have been inserted, and deleted between the beginning of the transaction and the current state. Thus, if I_k is the current database state reached by a transaction, and I_0 is the initial database state, for any relation T, we have:

$$\begin{array}{lcl} I_k[T] - I_0[T] & = & I_k[ins\_T] \\ I_0[T] - I_k[T] & = & I_k[del\_T] \\ I_k[ins\_T] \cap I_k[del\_T] & = & \emptyset \end{array}$$

where $I_k[T]$ denotes the instance of relation T in state I_k.

More formally, a rule is an expression of the form:

$$\begin{array}{lll} \text{if } B_1, ..., B_n & \text{then} & A_1, ..., A_k, \text{ or} \\ \text{if } B_1, ..., B_n & \text{then} & rollback \end{array}$$

where $k \geq 1, n \geq 0$. Each A_j is an expression, called a *literal*, of the form $(\neg)Q(x_1, ..., x_m)$ where Q is an extensional relation name (we sometimes call Q a predicate), and the x_i's are variables or constants. Each B_i is also a literal of the form $(\neg)Q(x_1, ..., x_m)$ where Q can be a delta relation name (we sometimes say a delta predicate). We denote $(\neg)Q(\vec{x})$ a literal where $\vec{x}$ is a tuple of variables.

We assume that there may exist a partial user-defined ordering between rules noted $\leq$. The notation $r \leq r'$ means that if r and r' are both firable at the same time, then rule r has priority over rule r'. This ordering is assumed to be available at any time during program execution.

2.2 Semantics of Rules

We shall only consider transactional and data modification events. There are three transactional events: begin of transaction (noted bot), end of transaction (noted eot), and checkpoint (noted chk). They are essentially used to synchronize the execution of rules with a transaction. Data modification events include the usual insert, delete, and update of a set of tuples of a specific relation. An *event type* $+T$ denotes an insertion into relation T and $-T$ denotes a deletion from relation T. A modification is represented by a deletion followed by an insertion.

Rules are executed at specific points, called *rule processing* points, during a transaction's execution. In our language, rules are processed when a checkpoint or the end of the transaction is encountered. The later case corresponds to the usual notion of *deferred* processing. The checkpoint command is a facility offered under different names (e.g., savepoint, process rules) by several systems (e.g, Ariel [Han95], A-RDL [SK95], Starburst [Wid95]). It enables a user to ask for the processing of deferred rules at an arbitrary point in the transaction. Since we do not distinguish immediate from deferred rules in our language, the same set of rules is considered at each rule processing point.

The case of immediate rules that can be processed before or after a data modification event does not jeopardize our algorithm. It just makes the presentation of our results heavier since two sets of rules must be considered everywhere (the immediate and the deferred rules). Note that an immediate

processing is easily simulated in our framework by putting a checkpoint after every data modification event in the transaction.

The events issued by a transaction are processed sequentially. When a data modification event occurs, its net effect on the current database state is computed and delta relations are changed accordingly. When an eot or a checkpoint occurs, all active rules are executed until no more rule is applicable.

The execution of rules at a rule processing point is described by the following procedure[2].

S := initial state;
repeat until steps 1-2 can have no effect or transaction is rollbacked:
1. find a rule r whose condition part produces tuples according to S
2. for each element in r's action part, perform the specified action (insert or delete) using the set of tuples produced by the condition part

Processing stops when no rule produces tuples in step 1 or no execution in step 2 can change a relation. In Step 2, each element of the action is a set-oriented delete or insert statement to the database, i.e., our rules are set-oriented rules.

We formally characterize the (consistent) net effect of a rule. First, some terminology will be useful. A *fact* over a relation Q of arity n is an expression $Q(a_1, ..., a_n)$ where each a_i is a constant. A *ground literal* is a literal in which all variables have been replaced by constants.

Now, let r be a rule: if ... then $A_1, ..., A_k$, and I a database instance. Let r' be an instantiation of r such that each variable is valuated to some constant, each positive literal in the if-part is a fact in I and each negative literal in the if-part holds, then r' is a *satisfying instantiation* of r in I. The set of ground literals in the then-part for all the satisfying instantiations of r in I is called the *effect* of r on I, noted $effect_r(I)$.

In fact, as we shall see later, we are not sensitive to the definition used for the effect of a rule. For instance, almost everything said in the sequel remains valid if each element in the action part is executed in its specified order (A_1 first, then A_2, ...). Our only assumption is that $effect_r(I)$ is consistent in the sense that it cannot contain both A and $\neg A$ for some fact A. To this aim, we define $cons\_effect_r(I)$ as the maximal consistent subset of $effect_r(I)$. Note that if the A_i's are executed on order, $effect_r(I)$ is always consistent.

We now define specific relations, called *event relations*, that play a key role in the computation of a rule program. Intuitively, given a rule r and a literal $l = (\neg)P(\vec{x})$ in its action part, the event relation associated with r and l records the net effect of r on the instance of P for a particular database state. Such relations are essential because they enable to know if a rule is firable: at least one of its event relations must be not empty.

Definition 2.1 Let r be a rule, $l = (\neg)P(\vec{x})$ a literal in its action part, and I a database state. The *event relation*, noted $Ev(r, l)$, associated with r and l is the set of ground instances of P in $cons\_effect_r(I)$ that represent a net change to $I[P]$.

Remark that a rollback rule has no event relation.

[2] As shown in [Wid93], the structure of this procedure is quite generic and can be used to describe rule processing in several relational active rule languages.

3 Problem Analysis

In this section, we introduce and motivate the main requirements for our data structure.

Example 3.1 Consider the three following active rules:

r_1: if $ins\_A(x,y), B(y,z), C(t,x)$ then $D(y,t), F(t,x)$
r_2: if $del\_E(x,y), B(y,z), F(z,t)$ then $E(x,y)$
r_3: if $D(x,y), E(y,z), G(z,t)$ then $H(x,y)$

We assume that the user-defined ordering specifies that r_1 precedes r_2 which precedes r_3.

Triggering events are implicit in these rules. Rules r_1 and r_2 are respectively triggered by insertions into A (ins_A must be non empty) and deletions from E (del_E must be non empty). Rule r_3 does not use any delta relation and is triggered by insertions into D, E, or G, and deletions from H. Intuitively, if D has more tuples then new tuples can be produced by the condition of r_3, and if H has fewer tuples then r_3 may become productive.

The local caching strategy of RETE suggests to cache and maintain every join occuring in the condition part of every rule. The strategy of TREAT leads to cache the relation defined by the condition of each rule. Finally, since there is no recursion in this set of rules, the global strategy of [FRS93] infers that each rule can be fired at most once during an execution and thus no expression needs to be cached.

Let us now study how the pattern of transactions impacts on the quality of the decisions taken by these caching strategies. Take transaction $\mathcal{T}_1$ that consists of a while loop containing an insertion into A, and a checkpoint. At each checkpoint, the event issued by the transaction triggers rule r_1. Firing r_1 generates insertions into D which trigger r_3, and insertions into F which are not capable of triggering r_2 (since del_E is empty, r_2 cannot be fired). Now, firing r_3 generates insertions into H and the processing of the rules stops. Within r_1, subexpression $R_1 = B(y,z) \bowtie C(t,x)$ is invariant throughout the transaction with respect to the changes issued by both $\mathcal{T}_1$ and the rules. Similarly, within rule r_3 subexpression $R_2 = E(y,z) \bowtie G(z,t)$ is also invariant. Thus, caching both R_1 and R_2 may save redundant computations. Caching all other subexpressions is an overhead: these expressions will not be useful and their maintenance will add an extra processing time. This example teaches the following tenet.

Tenet 1 Given a transaction $\mathcal{T}$, the repeated execution of events within $\mathcal{T}$ must be considered in order to perform a global optimization of the rules triggered by $\mathcal{T}$.

Take transaction $\mathcal{T}_2$ that consists of a while loop containing an insertion into A, a deletion from E, and a checkpoint. At each checkpoint, event $+A$ triggers both r_1 and r_3 as we have seen before, and event $-E$ triggers r_2. Within r_1, expression R_1 above is still an invariant. Within r_2, there is no invariant since F is possibly changing at each iteration (through firings of r_1). The interesting point is that R_2 is not anymore an invariant because firing r_2 may cause a change to E at each iteration. Thus, R_1 is the only expression whose caching is worthwhile. Here again, none of the previous strategies give good caching results. We derive the following tenet.

Tenet 2 Given a transaction $\mathcal{T}$ and a set of rules, only the subset of rules that can be triggered must be considered for a global optimization.

The next example points out the importance of the order in which events are issued by a transaction. Consider a transaction $\mathcal{T}_3$ that inserts tuples into A, sets a checkpoint, inserts tuples into G, and then commits. Transaction $\mathcal{T}_4$ inserts tuples into G, sets a checkpoint, inserts tuples into A, and then commits. The ordering of events is reversed in $\mathcal{T}_4$ with respect to $\mathcal{T}_3$. Rule r_3 is computed

twice in each transaction. However, in $\mathcal{T}_3$, r_3 is re-triggered by the insertions into G issued by the transaction, whereas in $\mathcal{T}_4$, r_3 is re-triggered by the insertions into D issued by rule r_1. In the first case, the invariant for r_3 is $R_3 = D(x,y) \bowtie E(y,z)$ whereas in the second case the invariant is R_2. Therefore, the caching decision for r_3 should not be same in $\mathcal{T}_3$ and $\mathcal{T}_4$. This shows the third tenet.

Tenet 3 Given a transaction $\mathcal{T}$ and a set of rules, the order in which events trigger rules must be considered to perform a global optimization of the rules triggered by $\mathcal{T}$.

In the sequel, these tenets are used as basic requirements for an algorithm that computes into a compact structure the influence of a transaction program on the triggering of rules. Using such a data structure, we show in section 7 that a global caching strategy such as the one described in [FRS93] can take appropriate caching decisions.

4 Analysis of Rules and Transactions

4.1 Abstract Representation of Transactions

We use an abstract notation for transactions[3]. Control structures are limited to sequential composition, conditionals and while loops. Within control structures, we intentionally omit the conditions which are not used by our algorithm. Data modifications are represented as event types (i.e., $+T$ or $-T$). A modification is represented as a deletion followed by an insertion. We use a BNF-like notation to describe the syntax for abstract transactions in Table 1.

Table 1: Syntax for Abstract Representation of Transactions

<transaction>	:	**bot** <statement> **;** [{,<statement> **;**} . . .] **eot**
<statement>	:	<data-modification> | <control-statement> | **chk**
<data-modification>	:	**+** relation-name | **-** relation-name
<control-statement>	:	**ifthen** <statement> [**else** <statement>] **endif**
	:	| **whiledo** <statement> [{,<statement>} . . .] **od**

Non terminal symbols are enclosed in angle brackets <>; terminal symbols that are key words are in boldface; alternative productions are introduced with |; $[a]$ means that a is optional; and $\{a\}\ldots$ means that a is repeated one or more times.

Example 4.1 The following is an abstract transaction.

```
bot;
+A;
ifthen whiledo +F; chk; od;
else -D; endif;
eot;
```

4.2 Triggering Events

We introduce the notion of triggering event, which provides sufficient conditions for deciding that a rule cannot be fired in a state I' resulting from the application of a sequence of data modifications to a state I, whatever is I.

Definition 4.1 Let r be a rule and E a set of event types, then E is said to be a *triggering set* for r, noted $Trig\_r$, if

[3]We shall use the word transaction to denote a transaction program's text, and we use the expression transaction execution to denote a running instance of the transaction

- for any state I such that r is not firable in I, if there exists a sequence σ of events that maps I into a state I' where r is firable, then E contains the event type of some event in σ,
- E is minimal.

Each element of E is called a *triggering event*

For instance, rule r_3 of Example 3.1 can be fired if new tuples are inserted into at least one of the three relations D, E, G or tuples are deleted from relation H since the last time r_3 was fired, or since the beginning of the transaction if r_3 has not been fired before. Thus, the triggering set for r_3 is $\{+D, +E, +G, -H\}$. The following proposition shows how to obtain the triggering set of a rule.

Proposition 4.1 Let r be a rule and T a relation

1. $+T$ is a triggering event for r if
   - T occurs positively in the condition part of r or negatively in the action part of r, or
   - ins_T or $\neg del\_T$ occurs in r, or
   - there is a relation T' that occurs both positively and negatively in the action part of r and del_T, $\neg ins\_T$ or $\neg T$ occurs in the condition part of r.
2. $-T$ is a triggering event for r if
   - T occurs negatively in the condition part of r or positively in the action part of r, or
   - $\neg ins\_T$ or del_T occurs in r, or
   - there is a relation T' that occurs both positively and negatively in the action part of r and ins_T, $\neg del\_T$ or T occurs in the condition part of r.

In Example 3.1, the first execution of rule r_1 within a transaction necessarily results from an insertion into ins_A, i.e., a net insertion into A. Thus, its triggering set is $\{+A\}$. But subsequent firings of r_1 may result from insertions into either A, B or C, or deletions from either D or F. Thus, the triggering set becomes $\{+A, +B, +C, -D, -F\}$. This observation leads to introduce the notion of an *initial triggering set* defined as a triggering set for the first firing of a rule.

Proposition 4.2 If r contains a delta relation then the initial triggering set for r, noted $Init\_r$, contains exactly one triggering event $+P$ for every delta relation ins_P occuring positively in r and one triggering event $-P$ for every delta relation del_P occuring positively in r.

Example 4.2 Take the rule program of Example 3.1. By Proposition 4.2, the initial triggering sets for r_1 and r_2 are respectively $\{+A\}$ and $\{-E\}$. r_3 has no initial triggering set because no delta relation occurs in the rule. By Proposition 4.1, the triggering sets for r_1, r_2 and r_3 are respectively $\{+A,+B,+C, -D, -F\}$, $\{-E, +B, +F\}$, and $\{+D, +E, +G, -H\}$.

Triggering sets can be used to determine which rules are capable of being fired after issuing some events. Consider the following transaction:

```
T:  bot;  -E;  eot;
```

Let I_1 be the state following event $-E$, rules are processed in I_1. Neither r_1 and r_3 are firable in I_1 since $-E$ is not in their initial triggering sets, but rule r_2 may be firable. If r_2 fires, it generates event $+E$, yielding a new state I_2. Rule r_1 is still not firable. But rule r_3 may be firable. Thus, r_1 will never fire, and r_2 and r_3 may fire at most once.

5 Interactions Between Transactions and Rules

The corner stone of our algorithm is the formal description of what rules can be triggered by a given transaction and recursively what rules can be triggered by rule firings. This is the subject of this section.

5.1 Compensative Rules

Going back to the previous example, we saw that the first firing of r_2 yields insertions into E, and since $+E$ is a triggering event for r_3, we concluded that r_3 could be firable. In fact, in rule r_2, the variables (x, y) occuring in del_E and E are the same. Thus, the only possible effect of literal $E(x, y)$ in the action part of r_2 is to delete tuples from del_E. Thus, no new tuple can be inserted into E and rule r_3 should not be firable. This idea that r_2 "compensates" or annihilates some previous effect is formalized below. These rules are quite useful because they enable to "repair" data modifications issued by transactions.

Definition 5.1 Let r be a rule and $l = P(\vec{x})$ (resp. $\neg P(\vec{x})$) a literal in its action part. If for any state I_k where r is firable, firing r cannot produce insertions into $I_k[ins\_P]$ (resp. $I_k[del\_P]$), then r is said to be *compensative* with respect to l.

Fact: Let r be a rule, and $l = P(\vec{x})$ (resp. $\neg P(\vec{x})$) a literal in its action part, r is *compensative* wrt l if a literal $del\_P(\vec{x})$ (resp. $ins\_P(\vec{x})$) occurs in the condition of r.

5.2 Triggering Graph

We now present our central data structure, called a *triggering graph*, that represents the flow of events from a given transaction towards the set of rules and within the rules. Given a transaction $\mathcal{T}$ and a set of rules $\grave{}$, the triggering graph for $\mathcal{T}$ and $\grave{}$ is a labelled directed graph noted $\mathcal{G}_{\grave{},\mathcal{T}}$.

$\mathcal{G}_{\grave{},\mathcal{T}}$ has two kinds of nodes: event nodes and rule nodes. There is one *event node* per event type occuring in $\mathcal{T}$ and one *rule node* per rule of $\grave{}$. The event nodes are the entries of the graph. There is a directed arc from rule r to rule r' if firing r may produce an event, $+P$ or $-P$, and either P, ins_P, or del_P occurs in r'. The label associated with (r, r') is a set of expressions that depends on r'. If r' is not a rollback rule, an expression in the label is of the form (l, θ, l'), where l (resp. l') is a literal in the action part of r (resp. r') and θ is in {+, -, ?}. Expression (l, θ, l') means that firing r may change the instance of a relation of r', and in turn change the instance of event relation $Ev(r', l')$. If the change is an insertion, θ is in $\{+, ?\}$, otherwise $\theta = -$. An expression of the form $(l, ?, l')$ means that r is compensative wrt l. If r' is a rollback rule, an expression in the label of (r, r') is of the form $l, \theta, *$, meaning that firing r may change some relation of r' and henceforth create new instantiations of r' (θ is in $\{+, ?\}$), or invalidate some instantiations of r' ($\theta = -$).

There is an arc (e, r) from the event node e to a rule r if e denotes the event $+P$ or $-P$ and either P, ins_P, or del_P occurs in r. If r is not a rollback rule, the label of (e, r) is a set of expressions of the form e, θ, l, where l is a litteral in the action part of r, and θ is in $\{+, -\}$. Expression $(e, +, l)$ (resp. $(e, -, l)$) means that e may change the instance of some relation of r and henceforth produce new tuples (resp. invalidate tuples) in $Ev(r, l)$. The label expression on an arc (e, r) where r is a rollback rule is of the form $(e, \theta, *)$. Its meaning is that processing e may change some relation occurring in r and henceforth create new instantiations of r (resp. invalidate some instantiations of r).

We now show how to compute label expressions on the arcs of a triggering graph. Let (r, r') be an arc with rules of the form:

r : if $\cdots$ then $l \cdots$
r' : if l" $\cdots$ then $l' \cdots$,

and $l' = (\neg)Q(\vec{z})$. Tables 2 (a) and (b) below show how to compute the label of (r, r'). In Table 2 (a), we assume that r' has a monotonic effect on relation Q, i.e. firing r' cannot both insert and delete tuples in Q; on the contrary, in Table 2 (b), r' has a non monotonic effect on Q. In both tables, symbol "?" after l means that r is compensative wrt l. As a special case, if r and r' do not denote the same rule and (l, l') is of the form $(Q(\vec{y}), \neg Q(\vec{x}))$ or $(\neg Q(\vec{y}), Q(\vec{x}))$, the label expression is $(l, +, l')$.

l =	l" =					
	ins_P($\vec{x}$)	¬ins_P($\vec{x}$)	del_P($\vec{x}$)	¬del_P($\vec{x}$)	P($\vec{x}$)	¬ P($\vec{x}$)
$P(\vec{y})$	(l,+,l')	(l,-,l')	(l,-,l')	(l,+,l')	(l,+,l')	(l,-,l')
$\neg P(\vec{y})$	(l,-,l')	(l,+,l')	(l,+,l')	(l,-,l')	(l,-,l')	(l,+,l')
$P(\vec{y})$?			(l,-,l')	(l,?,l')	(l,?,l')	(l,-,l')
$\neg P(\vec{y})$?	(l,-,l')	(l,?,l')			(l,-,l')	(l,?,l')

(a) Label of arc (r , r') if r' has a monotonic effect on Q

l =	l" =					
	ins_P($\vec{x}$)	¬ins_P($\vec{x}$)	del_P($\vec{x}$)	¬del_P($\vec{x}$)	P($\vec{x}$)	¬ P($\vec{x}$)
$P(\vec{y})$	(l,+,l')	(l,+,l')	(l,+,l')	(l,+,l')	(l,+,l')	(l,+,l')
$\neg P(\vec{y})$	(l,+,l')	(l,+,l')	(l,+,l')	(l,+,l')	(l,+,l')	(l,+,l')
$P(\vec{y})$?			(l,?,l')	(l,?,l')	(l,?,l')	(l,?,l')
$\neg P(\vec{y})$?	(l,?,l')	(l,?,l')			(l,?,l')	(l,?,l')

(b) Label of arc (r , r') if r' has a non monotonic effect on Q

e =	l" =					
	ins_P($\vec{x}$)	¬ins_P($\vec{x}$)	del_P($\vec{x}$)	¬del_P($\vec{x}$)	P($\vec{x}$)	¬ P($\vec{x}$)
$+P$	(e,+,l')	(e,-,l')	(e,+,l')	(e,-,l')	(e,+,l')	(e,-,l')
$-P$	(e,-,l')	(e,+,l')	(e,-,l')	(e,+,l')	(e,-,l')	(e,+,l')

(c) Label of arc (e, r') if r' has a monotonic effect on Q

e =	l" =					
	ins_P($\vec{x}$)	¬ins_P($\vec{x}$)	del_P($\vec{x}$)	¬del_P($\vec{x}$)	P($\vec{x}$)	¬ P($\vec{x}$)
$+P$	(e,+,l')	(e,+,l')	(e,+,l')	(e,+,l')	(e,+,l')	(e,+,l')
$-P$	(e,+,l')	(e,+,l')	(e,+,l')	(e,+,l')	(e,+,l')	(e,+,l')

(d) Label of arc (e, r') if r' has a non monotonic effect on Q

Table 2: Label expression computation

Let e be an event node and (e, r') an arc in the triggering graph. Tables 2 (c) and (d) show how to compute the label (e, θ, l) of (e, r'). In Table 2 (c), we assume that r' has a monotonic effect on Q while in Table 2 (d) the effect is non monotonic. If e denotes an event $+Q$ (resp. $-Q$) and $l' = \neg Q(\vec{z})$ (resp. $l' = Q(\vec{z})$), the label expression is $(e, +, l')$. As a special case, if (e, l') is of the form $(+Q, \neg Q(\vec{x}))$ or $(-Q, Q(\vec{x}))$, the label expression is $(e, +, l')$. Finally, if r' is a rollback rule, an expression on an arc (r, r') (resp. $((e, r'))$) is of the form $(l, \theta, *)$ (resp. $(e, \theta, *)$), where θ is computed according to Table 2 (a) (resp. Table 2 (c)).

The migration of data from the transaction to the rules and from a rule to the other rules is modelled by propagation paths in $\mathcal{G}_{\mathcal{T}}$. This is formalized below.

Propagation path: Let $\mathcal{G}_{\grave{},\mathcal{T}}$ be a triggering graph and $\pi = x_0 x_1 \ldots x_n$, a path in $\mathcal{G}_{\grave{},\mathcal{T}}$. We say that π is a *propagation path* if x_0 is an event node and there exists a sequence $\sigma = \lambda_1, \cdots, \lambda_n$, where for each i in $\{1 \cdots n\}$, $\lambda_i = l_{i-1}\ \theta_i\ l_i$ is an expression in the label of (x_{i-1}, x_i), θ_i is in $\{+, ?\}$, $l_0 = x_0$, and, excepted for $i = n$, l_i is not in $\{*\}$. We say that π is *positive* (resp. *weakly positive*) w.r.t σ, if each θ_i is in $\{+\}$ (resp. if some θ_i is in $\{?\}$).

Example 5.1 Take for instance, the set of rules $\grave{}$ and transaction $\mathcal{T}$ below.

r_1 :	if $ins\_A(x,y), del\_B(y,z), C(z,t)$	then	$E(x,z)$
r_2 :	if $del\_D(x,y), C(y,z), E(z,t)$	then	$D(x,y)$
r_3 :	if $F(x,y), B(y,z), D(y,z), E(z,t)$	then	$H(x,y)$
r_4 :	if $ins\_E(x,y), H(y,z), K(z,t)$	then	$L(z)$

Priorities: $r_1 \leq r_2 \leq r_3 \leq r_4$

Transaction $\mathcal{T}$ (bot and eot are omitted):

```
ifthen  whiledo +A; ifthen -D; else +C; endif ; chk; od; +F;
else    -B; whiledo +F; od; chk; endif;
+B;
```

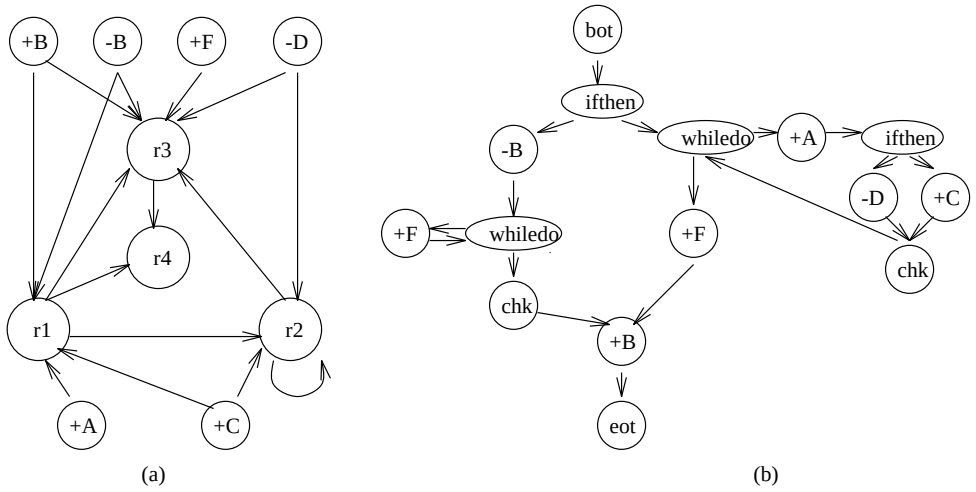

Figure 1: triggering graph and flow graph for $\mathcal{T}$

Figure 1 (a) represents the triggering graph for $\mathcal{T}$ and (b) its flow graph. Table 3 gives the labels on the arcs of the triggering graph. $+A\ r_1\ r_2\ r_3$ is a weakly positive propagation path with the associated sequence of label expressions $(+A, +, E)(E, +, D)(D, ?, H)$. The label expression $(D, ?, H)$ indicates that r_2 is compensative wrt D.

5.3 Triggering Graph Simplification

In this section, we present optimization criteria that enable to eliminate irrelevant rules and label expressions from a triggering graph.

5.3.1 Simple Transactions

Definition 5.2 Let $\mathcal{T}$ be a transaction (program) and R a fragment of $\mathcal{T}$, called a *region* in the sequel. We say that R is a *simple region* if R consists of an event type or a checkpoint which is not embedded

	r_1	r_2	r_3	r_4
$+C$	$\{(+C,+,E)\}$	$\{(+C,+,D)\}$		
$+A$	$\{(+A,+,E)\}$			
$+B$	$\{(+B,-,E)\}$		$\{(+B,+,H)\}$	
$-B$	$\{(-B,+,E)\}$		$\{(-B,-,H)\}$	
$-D$		$\{(-D,+,D)\}$	$\{(-D,-,H)\}$	
$+F$			$\{(+F,+,H)\}$	
r_1		$\{(E,+,D)\}$	$\{(E,+,H)\}$	$\{(E,+,L)\}$
r_2		$\{(D,-,D)\}$	$\{(D,?,H)\}$	
r_3				$\{(H,+,L)\}$

Table 3: Label expressions of arcs in $\mathcal{G}_{\grave{},\mathcal{T}}$

in a whiledo control statement. We say that R is a *complex region* if R consists of an outermost whiledo control statement.

In the following, we focus on a particular kind of transactions, called *simple transactions*, that consist of a sequence [bot; R_1; ...;R_n; eot] ($n \geq 0$), where each R_i is either a simple region or a complex region.

Example 5.2 The following is a simple transaction called $\mathcal{T}_1$.

```
bot;  whiledo ; +A; ifthen -D ; else +C; endif; chk; od  +F; +B; eot
```

In this example, R_1 is the complex region [whiledo $+A$; ifthen -D; else +C; endif; chk; od], and R_2 (resp. R_3) is the simple region [$+F$] (resp.[$+B$]).

During a simple transaction execution, each statement occuring in a simple region is executed exactly once, and we shall assume that each statement in a complex region is executed more than once (because we shall perform a static analysis of transactions).

5.3.2 Irrelevant Rules

Given a simple transaction $\mathcal{T}$, the first simplification consists of removing rules that have no chance of being fired during any transaction execution.

Take for instance transaction $\mathcal{T}_1$ of Example 5.2, with the set of triggers of Example 5.1. $\mathcal{T}_1$ has the same triggering graph than transaction $\mathcal{T}$ of Example 5.1, excepted that event node $-B$, its adjacent edges, and the corresponding label expressions are removed. Let us consider rule r_4. Its initial triggering set is $\{+E\}$. The graph contains an arc (r_1, r_4) whose label expression $(E, +, L)$ indicates that firing r_1 generate insertions into E. The initial triggering set for r_1 is $\{+A, -B\}$. By inspection of the labels on the arcs ending at r_1, we see that there is no label of form (B, θ, l), thus $-B$ cannot be generated. Thus, r_1 is irrelevant wrt $\mathcal{T}_1$ and so is r_4.

We now formalize the notion of irrelevant rule.

Irrelevant rules: Let $\grave{}$ be a set of triggers, $\mathcal{T}$ = [bot (=R_0); R_1; ...; R_n; eot (=R_{n+1})] a simple transaction, and R_i a distinguished element of $\mathcal{T}$. Let E_i denote the set of event types occuring in at least one R_k ($1 \leq k \leq i$). Then, the set $\mathcal{N}_{R_i,\mathcal{T}}$ of *irrelevant rules* with respect to R_i in $\mathcal{T}$ is recursively defined as follows. Given a rule r,

1. if there is no propagation path π = x_0...x_n r such that (i) x_0 is in E_i, and (ii) π is positive wrt some sequence of label expressions, then r is in $\mathcal{N}_{R_i,\mathcal{T}}$,

2. if $Init\_r$ contains some event e, and there is no arc (e, r), and there is no propagation path π = $x_0...x_n$ r ($n > 0$) that satisfies the following properties: (i) x_0 is in E_i, and (ii) π is positive wrt some sequence σ = $(\lambda_k)_{1 \leq k \leq n+1}$ in which λ_n is of the form $(l, +, P(\vec{x}))$ (resp. $(l, +, \neg P(\vec{x}))$) with $e = +P$ (resp. $e = -P$), then r is in $\mathcal{N}_{R_i,\mathcal{T}}$,

3. if the initial triggering set of rule r contains some event $e = +P$ (resp. $-P$), and there is no propagation path π = $x_0...x_n$ r ($n > 0$) that satisfies the following properties: (i) x_0 is in E_i, (ii) for i in $\{1, ..., n\}$ x_i is not in $\mathcal{N}_{R_i,\mathcal{T}}$, and (iii) π is positive wrt some sequence σ = $(\lambda_k)_{1 \leq k \leq n+1}$ in which λ_n is of the form $(l, +, P(\vec{x}))$ (resp. $(l, +, \neg P(\vec{x}))$ with $e = +P$ (resp. $e = -P$), then r is in $\mathcal{N}_{R_i,\mathcal{T}}$.

4. if there is no propagation path π = $x_0...x_n$ r such that (i) x_0 is in E_i, (ii) π is positive wrt some sequence of label expressions, and (iii) for i in $\{1, ..., n\}$ x_i is not in $\mathcal{N}_{R_i,\mathcal{T}}$, then r is in $\mathcal{N}_{R_i,\mathcal{T}}$.

5. Only rules that satisfy items 1, 2, 3, or 4 are in $\mathcal{N}_{R_i,\mathcal{T}}$.

As a specific case, if R_i = [eot], the rules contained in $\mathcal{N}_{R_i,\mathcal{T}}$ have no chance of being fired during the execution of $\mathcal{T}$. We shall say that these rules are *irrelevant wrt* $\mathcal{T}$. Remark that the set of irrelevant rules wrt $\mathcal{T}$ only depends on the set of event types occuring in $\mathcal{T}$.

Example 5.3 Take transaction $\mathcal{T}_1$. We verify that r_3 is irrelevant with respect to R_1 in $\mathcal{T}_1$. E_1 = $\{+A, -D, +C\}$. As path π_1 = $+A$ r_1 r_3 (resp. π_2 = $+C$ r_1 r_3 is positive wrt σ_1 = $(+A, +, E)(E, +, H)$ (resp. σ_2 = $(+C, +, E)(E, +, H)$) item 1 above does not apply. Since no delta relation occurs in r_3, $Init\_r_3$ is empty. Thus items 2 and 3 do not apply. Item 4 leads to examine rule r_1. $Init\_r_1$ = $\{+A, -B\}$ and item 2 applies (there is no arc $(-B, r_1)$ and no path $x_0...x_nr_1$ ($n > 0$)). Thus r_1 is irrelevant wrt R_1 in $\mathcal{T}_1$ and so is r_3 (by item 4).

5.3.3 Irrelevant labels

Let $\mathcal{T}$ be a simple transaction and $\mathcal{G}_{\backslash,\mathcal{T}}$ its triggering graph where irrelevant rules have been removed. A further simplification consists of removing the label expressions occuring in some arc ending at some rule r, and denoting data modifications that cannot arise after the first firing of r. Unlike the previous one, this simplification depends on the structure of the transaction program.

For instance, take transaction $\mathcal{T}_1$ of Example 5.2. The first simplification leads to remove rules r_1 and r_4. Let us focus on the label expression λ = $(D, ?, H)$ on the arc (r_2, r_3). There are four propagation paths ending at r_3: π_1 = $-D$ r_2 r_3; π_2 = $+C$ r_2 r_3; π_3 = $+F$ r_3 and π_4 = $+B$ r_3. Path π_3 (resp.π_4) is positive wrt its associated sequence of label expressions. Paths π_1 and π_2 are both weakly positive wrt their associated sequences of label expressions. Hence, rule r_3 may only be triggered for the first time by data modifications vehicled by π_3 and π_4, and firing r_2 cannot produce events capable of triggering r_3 for the first time. $-D$ and $+C$ only occur in region R_1, while $+F$ (resp. $+B$) occurs in region R_2 (resp. region R_3). As region R_1 contains a checkpoint statement, the repetitive firings of r_2 wrt the whiledo control statement precedes the first firing of r_3. Hence, data modifications indicated by λ are irrelevant.

In what follows, we formalize the notion of irrelevant label expression. Given a simple transaction $\mathcal{T}$ = [bot; R_1; ...; R_n; eot], we shall note S_T the sequence extracted from $\mathcal{T}$ that contains eot, bot, plus the R_i's consisting of a checkpoint or a complex region including checkpoints. For instance, in transaction $\mathcal{T}_1$ of Example 5.2, S_{T_1} = (bot, R_1, eot). In the sequel S_T will be called *synchronization sequence* of $\mathcal{T}$.

Irrelevant labels: Let ` be a set of triggers, T = [bot (=R_0); R_1; ...; R_n; eot (=R_{n+1})] a simple transaction, S_T the associated synchronization sequence, and a = (x, r) an arc of the triggering graph with an expression λ in its label. We shall say that λ is *irrelevant* wrt $\mathcal{T}$ if S_T contains two consecutive elements R_{min} and R_{max} ($min < max$) such that

1. r is irrelevant wrt R_{min} in $\mathcal{T}$, and
2. there is no path π = x_0 ...x_n r satisfying the following properties: (i) x_n = x, (ii) π contains no irrelevant rule wrt R_{max} in $\mathcal{T}$, (iii) π contains some sequence λ_1, ..., λ_n, λ, (iv) x_0 occurs in some R_k ($min \leq k < max$), and (v) x_0 occurs in some R_k ($k \geq max$), or π contains some rule x_i ($1 \leq i \leq n$) having no priority over r.

Example 5.4 Going back to the previous example, we verify that label expression λ = $(D, ?, H)$ on arc a = (r_2, r_3) is irrelevant wrt $\mathcal{T}_1$. We take R_{min} = R_1 and R_{max} = [eot]. As r_3 is irrelevant wrt R_1 in $\mathcal{T}_1$, item 1 applies. Item 2 leads to examine the paths containing both arc a and a sequence of the form λ_1, ..., λ. Paths π_1 = $-D$ r_2 r_3 and π_2 = $+C$ r_2 r_3 hold. Both $-D$ and $+C$ occur in R_1. Rule r_2 has priority over r_3. As R_{max} = [eot], neither $-D$ nor $+C$ occurs in some R_k ($k \geq max$). Hence, neither π_1 nor π_2 satisfies item 2 (v). Thus item 2 applies.

5.4 General transactions

To simplify the triggering graph for a general transaction, $\mathcal{T}$, the idea is first to decompose $\mathcal{T}$ into a set of simple transactions, and then to eliminate an irrelevant rule or label if the irrelevance can be proved with respect to every simple transaction using the previous results.

Take transaction $\mathcal{T}$ of Example 5.1. Any possible execution may be described by one of the following simple transactions: $\mathcal{T}_1$ = [bot; whiledo +A; ifthen -D ; else +C; endif; chk; od; +F; +B; eot], and $\mathcal{T}_2$ = [bot; $-B$; whiledo $+F$; od; chk; $+B$; eot]. Take rule r_1. Its initial triggering set is $\{+A, -B\}$, and there are for instance two arcs ($+A$,r_1) and ($-B$,r_1) whose labels $(+A, +, E)$ and $(-B, +, E)$ indicate that r_1 is not a priori irrelevant. But no transaction execution may process both insertions into A and deletions from B, thus r_1 is indeed irrelevant. So, verifying that r_1 is irrelevant wrt both $\mathcal{T}_1$ and $\mathcal{T}_2$ allows to conclude that r_1 is irrelevant wrt $\mathcal{T}$.

Definition 5.3 Let G be a single entry single exit directed graph, E_{in}, (resp. E_{out}) the entry node (resp. exit node), and g a subgraph. We say that g is a *basic component* of G if (i) g contains E_{in} and E_{out}, (ii) there is a traversal path of G that contains every node of g, and (iii) for any node n in g, if n belongs to some cycle C in G, then g contains C.

Definition 5.4 Given a transaction $\mathcal{T}$ and its flow graph $\mathcal{F}$, we shall call *decomposition* of $\mathcal{T}$ the set $\mathcal{D}_\mathcal{T}$ consisting of the transactions associated with every basic component of $\mathcal{F}$.

The simplification of a triggering graph for a general transaction is based on the following proposition.

Proposition 5.1 Let $\mathcal{T}$ be a transaction, $\mathcal{D}_\mathcal{T}$ = $\{\mathcal{T}_1, ..., \mathcal{T}_n\}$ its decomposition, ` a set of triggers, $r$ a rule, and $(x, r)$ an arc of $\mathcal{G}_{`,\mathcal{T}}$ with an expression λ in its label. Then

1. r (resp. λ) is irrelevant wrt $\mathcal{T}$ if, for i = 1..n, r (resp. λ) is irrelevant wrt $\mathcal{T}_i$.
2. each $\mathcal{T}_i$ is a simple transaction.

6 Algorithm

Given a set of triggers $\grave{}$, and a transaction $\mathcal{T}$, our main algorithm constructs a simplified triggering graph in three steps. The first step is an initialization step. Four global variables are initialized: G is initialized with the non simplified graph $\mathcal{G}_{\grave{},\mathcal{T}}$, D has the decomposition of $\mathcal{T}$, $Non\_Simplifiable\_rule$, and $Non\_Simplifiable\_label$ are set to the empty set. During the second step, each transaction in D is examined, and we use the *Simple_transaction* algorithm for computing its corresponding sets of rules and labels that are not irrelevant. The results are cumulated into variables $Non\_Simplifiable\_rule$ and $Non\_Simplifiable\_label$. At step 3 every rule (resp. every label expression) that doesn't occur in $Non\_Simplifiable\_rule$ (resp. $Non\_Simplifiable\_label$) is removed from G.

The core of the algorithm is the *Simple_transaction* algorithm presented in Figure 2.

Simple_transaction algorithm:
input: a simple transaction, T, a set of triggers, $\grave{}$, and the triggering graph, G;
output: the set of rules and the set of label expressions contained in the simplified triggering graph for $\grave{}$ and T;

Initialization step:
let E (resp. S) denote the set of event type occuring in T
(resp. the synchronization sequence of T);

non_simplifiable_rule computation:
let N denote the set of irrelevant rules wrt T; $non\_simplifiable\_rule := \grave{} - N$;

non_simplifiable_label computation:
$Pertinent\_event := \{e \in E \mid \exists r \in non\_simplifiable\_rule, (e,r) \in G\}$;
$non\_simplifiable\_label := \emptyset$; R_{max} := eot;
repeat until $Pertinent\_event$ is empty
  R_{min} := the region of T that immediatly precedes R_{max} in S;
  $Curr\_event$:= {e ∈Pertinent_event | e occurs in R_{min} or in a region succeeding R_{min} in T};
  let $Curr\_graph$ denote the subgraph of G that contains:
  1. each event-node associated with an event type in $Curr\_event$
  2. each rule r s.t. G contains a propagation path $\pi = x_0 \ldots x\ r$ in which no rule is irrelevant wrt R_{max}, and $x_0 \in Curr\_event$
  for each arc (x,r) in $Curr\_graph$ do
    for each expression λ on (x,r) s.t $\lambda \notin non\_simplifiable\_label$ do
      /* Test λ for irrelevancy */
      if r is not irrelevant wrt R_{min} then $Irrelevant$:= false;
      else
        P := {$\pi = x_0 \ldots x\ r$ in $Curr\_graph$ | π contains a sequence of labels $\lambda_1 \ldots \lambda$};
        if for each $\pi = x_0 \ldots x\ r$ in P, each rule in $\{x_1, \ldots, x\}$ has priority over r
        then $Irrelevant$:= true else $Irrelevant$:= false;
      if not $Irrelevant$ then add λ to $non\_simplifiable\_label$;
  $R_{max} := R_{min}$; $Pertinent\_event := Pertinent\_event$ - $Curr\_event$;
return $non\_simplifiable\_rule$, $non\_simplifiable\_label$;

Figure 2: Simple_transaction algorithm

We run the main algorithm on transaction $\mathcal{T}$ of Example 5.1. After the first step, G contains the triggering graph of Figure 1 with label expressions given in Table 3. D contains transaction $\mathcal{T}_1$ of Example 5.3, and transaction $\mathcal{T}_2$ = [bot; $-B$; whiledo $+F$; od; chk; $+B$; eot]. The *Simple_transaction algorithm* is successively applied to $\mathcal{T}_1$ and $\mathcal{T}_2$.

We start with $\mathcal{T}_1$. After the initialization step, E and S respectively contain {$+A$, $+B$, $+F$, $-D$, $+C$} and (bot, R_1, eot). After the non_simplifiable_rule computation step, $non\_simplifiable\_rule$

contains r_2 and r_3. At the first iteration of the non_simplifiable_label computation step, $Pertinent\_event$ = $\{+B, +F, -D, +C\}$ = $Curr\_event$, and R_{min} = R_1. $Curr\_graph$ contains r_2, r_3, and the event nodes associated with $Pertinent\_event$ As rule r_2 is not irrelevant wrt B_1, the test for irrelevancy fails for every label expression on the arcs ending at r_2. These expressions are added to $non\_simplifiable\_label$. We now examine the arcs ending at r_3. This rule is irrelevant wrt R_{min}. The test for irrelevancy succeeds for every label expression on the arcs $(+F, r_3)$, $(+B, r_3)$, and (r_2, r_3). As $Curr\_event$ is empty, the process stops. The result is cumulated into the global variables $Non\_simplifiable\_rule$ and $Non\_simplifiable\_label$.

Now, the algorithm is applied to $\mathcal{T}_2$. The regions of $\mathcal{T}_2$ are R_1 = $[-B]$, R_2 = [whiledo $+F$; od], R_3 = [chk], and R_4 = $[+B]$. S_{T_2} = (bot, R_3, eot). As rules r_1, r_2 and r_4 are irrelevant wrt $\mathcal{T}_2$, $non\_simplifiable\_rule$ = $\{r_3\}$ and $Pertinent\_event$ contains $-B$, $+B$, and $+F$. At the first iteration, R_{min} = R_3, R_{max} = [eot], and $Curr\_event$ = $\{+B\}$. As r_3 is not irrelevant wrt R_3, the expression on arc $(+B, r_3)$ is not irrelevant wrt $\mathcal{T}_2$; it is added to $non\_simplifiable\_label$. At the second iteration, R_{max} = R_3, R_{min} = [bot] and $Curr\_event$ = $\{-B,+F\}$. As r_3 is irrelevant wrt bot, the expressions on the arcs $(-B, r_3)$ and $(+F, r_3)$ are irrelevant wrt $\mathcal{T}_2$. The process stops.

The resulting simplified triggering graph contains rules r_2 and r_3, and event nodes $+C$, $+B$, $-B$, $-D$, and $+F$. Table 4 below gives the label expressions.

	$+C$	$+B$	$-D$	r_2
r_2	$(+C, +, D)$		$(-D, +, D)$	$(D, -, D)$
r_3		$(+B, +, H)$		

Table 4: Label expressions on the arcs of $\mathcal{G}_{\mathcal{T}}$

Given a general transaction $\mathcal{T}$, the following simple transaction's properties allow to minimize the number of simple transactions to consider in the decomposition.

Let $\mathcal{T}_1$ and $\mathcal{T}_2$ be two simple transactions, S_{T_1} (resp. S_{T_2}) the associated synchronization sequence. We shall say that $\mathcal{T}_2$ is *more general* than $\mathcal{T}_1$ if each S_i in S_{T_1} may be associated with some S_j in S_{T_2} s.t. (i) every event type occuring in some region preceding S_i in $\mathcal{T}_1$ also occurs in some region preceding S_j in $\mathcal{T}_2$, (ii) every event type occuring in some region succeeding S_i in $\mathcal{T}_1$ also occurs in some region succceeding S_j in $\mathcal{T}_2$, and (iii) every event type occuring in S_i also occurs in S_j.

For instance, transaction $\mathcal{T}_2$ = [bot; $+F$; whiledo $+A$; chk; od; $-B$; eot], is more general that transaction $\mathcal{T}_1$ = [bot; $+A$; whiledo $-B$; od; eot].

Proposition 6.1 Let $\mathcal{T}$ be a transaction, $\mathcal{D}_{\mathcal{T}}$ its decomposition, $\mathcal{T}_1$ and $\mathcal{T}_2$ two simple transactions in $\mathcal{D}_{\mathcal{T}}$ s.t. $\mathcal{T}_2$ is more general than $\mathcal{T}_1$. Let r a be rule, (x, r) an arc of the triggering graph and λ an expression in the label. If r is irrelevant wrt $\mathcal{T}_2$, then r is also irrelevant wrt $\mathcal{T}_1$, and if λ is irrelevant wrt $\mathcal{T}_2$, then λ is also irrelevant wrt $\mathcal{T}_1$.

7 Applications of a Triggering Graph

In this section, we first discuss how a triggering graph can be used by a global caching strategy. Then, we discuss how the construction of a triggering graph can be adaptated for different active rule language than the one considered in this paper.

7.1 Applying a Global Caching Strategy

We come back to Example 3.1. First, consider transaction $\mathcal{T}_1$ = [bot; whiledo $+A$; chk; od; eot] of Section 3. Figure 3(a) gives the simplified triggering graph for this transaction. According to the

flow graph, the sequence $+A$; chk; can be repeated and the arcs $(+A, r_1)$, and (r_1, r_3) indicate that this may cause repeated triggerings of both r_1 and r_3. The labels of these arcs also indicate that re-triggering r_1 (resp. r_3) results from insertions into A (resp. into D).

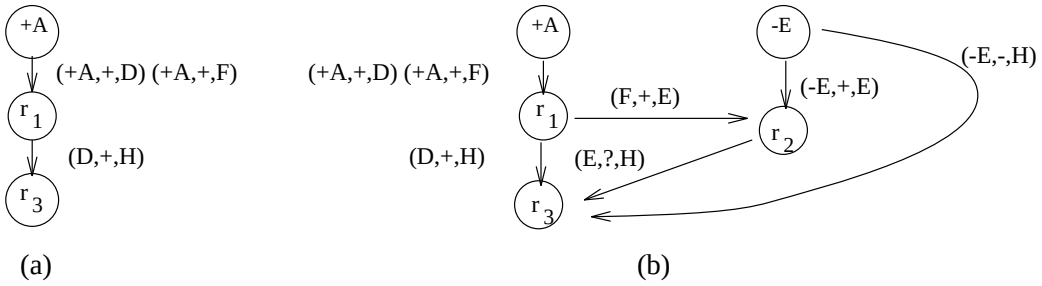

Figure 3: Simplified triggering graphs for $\mathcal{T}_1$ and $\mathcal{T}_2$

The global caching strategy presented in [FRS93] aims to compute the largest "autonomous"[4] expression in the condition part of a rule. In this example, $R_1 = B(y, z) \bowtie C(t, x)$ is the largest autonomous expression with respect to the changes to r_1's relations induced by the triggering graph (i.e., insertions into A). Similarly, $R_2 = E(y, z) \bowtie G(z, t)$, is the largest autonomous with resepct to the changes to r_2's relations induced by the triggering graph.

Consider now transaction $\mathcal{T}_2$ = [bot; whiledo $+A$; $-E$; chk; od; eot]. Its simplified triggering graph is shown Figure 3(b). Combining the flow graph with the triggering graph, we infer that the sequence $+A$; $-E$; chk; can be repeated and may cause repeated triggerings of r_1, r_2, and r_3. The labels of the arcs indicate that re-triggering r_1 results from insertions into A, and re-triggering r_2 (resp. r_3) results from insertions into F and deletions from E (resp. insertions into D and deletions from E). The label on the arc (r_2, r_3) indicates that r_2 is compensative wrt E and no new tuple is generated into E. From that, the global caching strategy of [FRS93] concludes that R_1 and R_2 are still good caching decisions.

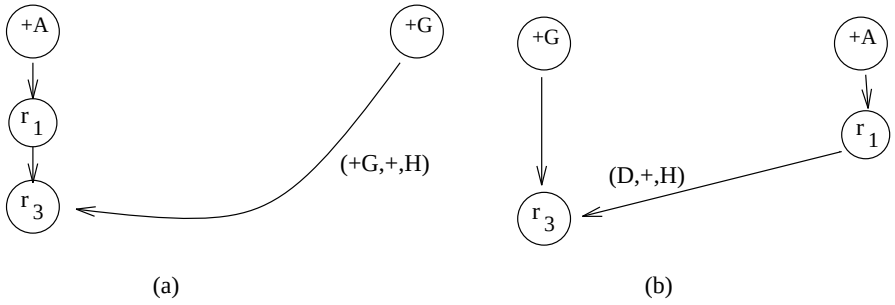

Figure 4: Simplified triggering graph for $\mathcal{T}_3$ and $\mathcal{T}_4$

Consider now transactions $\mathcal{T}_3$ = [bot; $+A$; chk; $+G$; eot] and $\mathcal{T}_4$ = [bot; $+G$; chk; $+A$; eot]. The simplified triggering graph of $\mathcal{T}_3$ is shown in Figure 4(a). As there is no label expression on the arcs ending at r_1, we conclude that r_1 can be fired only once. While the label $(+G, +, H)$ on the arc $(+G, r_3)$ indicates that r_3 may be fired more than once and and re-triggering r_3 results from

[4] An expression is said be autonomous with respect to a set of changes $\mathcal{C}$ to its operand relations if its associated relation can be computed only from its previous state and the changes in $\mathcal{C}$.

insertions into G. $R_3 = D(x,y) \bowtie E(y,z)$ is the maximal autonomous expression of r_3. The global caching strategy concludes that R_3 is profitable to cache. The simplified triggering graph of $\mathcal{T}_4$ is shown in Figure 4(b). Here, the labels indicate that re-triggering of r_3 results from insertions into D. $R_2 = E(y,z) \bowtie G(z,t)$ is the maximal autonomous expression of r_3. The global caching strategy concludes that R_2 is profitable to cache.

7.2 The Case of ECA rules

We briefly give an idea of how the definition of a triggering graph needs to be changed in the case of an ECA rule language [WCD95]. Using our notations, an ECA rule of the form "when $Event$ if $Condition$ then $Action_1$, ..., $Action_n$" is defined as follows. $Event$ specifies the triggering set for the rule. The condition part specifies a rule of the form: "if $Condition$ then OK". Then each Action is a database statement that can be viewed in our framework as a rule "if OK and $condition_i$ then A_i".

To illustrate, take an ECA rule r expressed with our notations:

$Trig\_r$	=	$\{+A\}$	
$Condition$	:	if $ins\_A(x,y), B(y,z), C(t,x)$	then OK
$Action_1$	:	if $ins\_A(x,y), B(y,z), OK$	then $D(x,z)$
$Action_2$	:	if $ins\_A(x,y), C(t,x), OK$	then $F(t,x)$

Suppose we have a transaction $\mathcal{T}_1$ = [bot; whiledo $+A$; chk; od; eot]. Then the triggering graph for $\mathcal{T}_1$ and r will be represented as shown in Figure 5. Node r_1 represents the condition part, while r_2 and r_3 respectively represent $Action_1$ and $Action_2$.

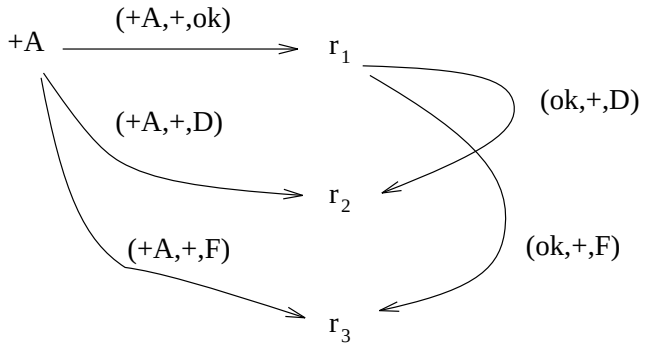

Figure 5: Simplified triggering graph for $\mathcal{T}_1$ and r

Note that our results on triggering events can still be useful for computing the influence of events and rules on the condition and action parts of an ECA rule.

8 Conclusions

We proposed a new approach for optimizing repetitive evaluation of database triggers within a transaction. The key idea is to analyze how repetitive triggerings of rules can be caused by the structure of transaction programs. We showed that without such an analysis, well known incremental rule evaluation algorithms can take erroneous caching decisions when they are applied to a set of database triggers. We presented an algorithm that, given a transaction's flow graph, constructs a compact data structure called a triggering graph. First, for each possible transaction execution, this

graph indicates which rules may be triggered. Second, for every rule r capable of being triggered and fired several times, the graph represents the "influence" of both the transaction and the rules on r.

We believe that, in an active database framework, our approach is more effective than the approach followed by most incremental rule evaluation algorithms such as RETE or TREAT. First, although a rule base may consist of a large number of active rules, only a few rules will be triggered by any single transaction execution. Thus, only these rules are worth to optimize. Second, for set-oriented rules[5], RETE and TREAT -like algorithms at best optimize the recursive processing of rules. However, recursion arises quite rarely in active rules, according to our experience. On the contrary, when one looks at the structure of a transaction program, repetitive triggerings of rules are quite frequent. Typically, they arise when rules are defined as immediate[6] and the transaction uses iterative statements such as cursor-based statements in embedded SQL.

However, the benefits offered by our approach are paid at the price of a loss of flexibility. If a transaction is known in advance then its associated triggering graph can be built and optimized versions of rules can be generated for that transaction. But changes to the rule base may require to redo the optimization process. This is clearly not the case if rules are optimized separately as RETE or TREAT-like algorithms do.

We envision two future directions of research. One is to adapt the construction of our triggering graph for ECA rule languages along the line indicated in Section 7. Second, we wish to investigate a more dynamic approach (in contrast to the purely syntactic approach described here). We could for instance exploit the fact that the "if then" branch of a transaction is traversed in 90% of the executions.

Acknowledgements: We wish to thank Francois Llirbat and Maja Matulovic for their helpful comments on previous versions of this paper.

References

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

[DWE89] L.M. Delcambre, J. Waramahaputi, and J.N. Etheredge. Pattern Match Reduction for the Relational Production Language in the USL MMDBS. *SIGMOD Record*, 18(3):59–67, September 1989. Special Issue on Rule Management and Processing in Expert Database Systems.

[For82] C. Forgy. RETE, a fast algorithm for the many patterns many objects match problem. *J. Artificial Intelligence*, 19:17–37, 1982.

[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.

[Han92] E. Hanson. Rule Condition Testing and Action Execution in Ariel. *Proc. of the ACM SIGMOD International Conference on Management of Data*, pages 49–58, June 1992.

[Han95] E. Hanson. The Ariel Project. In [WCD95].

[Mir87] D.P. Miranker. TREAT: A Better Match Algorithm for AI Production Systems. In *Proceedings of the National Conference on Artificial Intelligence*, Seattle, Washington, 1987.

[5] that correspond to "for each statement" rules in SQL3

[6] Immediate rules is the default mode in SQL3 and all commercial database systems

[Pai86] R. Paige. Programming with Invariants. *IEEE Software*, 3,1:56–69, 1986.

[PH87] R. Paige and F. Henglein. Mechanical Translation of Set Theoretic Problem Specifications into Efficient RAM Code–A Case Study. *Journal of Symbolic Computation*, 4:207–232, 1987.

[PK82] R. Paige and S. Koenig. Finite Differencing of Computable Expressions. *ACM Transactions on Programming Languages and Systems*, 4(3), 1982.

[SLR88] T. Sellis, C. Lin, and L. Raschid. Implementing large production systems in a DBMS environment: Concepts and algorithms. In *Proc. International Conference SIGMOD*, Chicago, May 1988.

[SK95] E. Simon and J. Kiernan. The A-RDL System. In [WCD95].

[SZ91] A. Seguev and J. Leon Zhao. Data Management for Large Rule Systems. In *Proc. International Conference on Very Large Databases*, Barcelona Catalonia, Spain, Aug. 1991.

[WCD95] J. Widom, S. Ceri, and U. Dayal. *Active Database Systems: Triggers and Rules for Advanced Database Processing*. Morgan-Kaufmann, San Francisco, California, 1995. to appear.

[Wid93] J. Widom. Deductive and Active Databases: Two Paradigms or End of Spectrum? In *Proc. of the first International Workshop On Rules In Databases Systems*, Edinburgh, Scotland, Sept. 1993.

[Wid95] J. Widom. The Starburst Rule System. In [WCD95].

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