Implementing High Level Active Rules on top of a Relational DBMS

E. Simon, J. Kiernan, C. de Maindreville

INRIA Rocquencourt,
78153 Le Chesnay,
France
Eric.Simon@inria.fr, kiernan@almaden.ibm.com, maindrev@aar.alcatel-alsthom.fr

Abstract: Active database systems have rules (usually called triggers), consisting of an event that causes a condition to be evaluated, and if true, results in the execution of a predefined action. However, existing trigger languages have a few drawbacks. First, the proposed semantics do not take advantage of well understood and accepted formalisms developed for rule-based systems, and thereby do not capitalize on existing rule-based technology. Second, trigger languages are low-level languages. These languages require that the user provides all triggering conditions associated with rules. This makes difficult the specification of triggers and their maintenance. In this paper, we present an extension of a deductive database language, namely RDL1, towards active rules. By active, we mean rules that react to external events. Rules are expressed at a high level so that triggering conditions are derived from rules by the system. The semantics of our rule language is formally described by means of a partial fixpoint operator which encompasses the deductive database and active database paradigms. We also present an architecture in which the system responsible for detecting events issued by application programs and triggering rules, is front-ended to a relational DBMS.

Proceedings of the 18th VLDB Conference
Vancouver, British Columbia, Canada 1992

1. Introduction

Integrating rules within a DBMS has been the focus of important research on active database systems [Han89, SJGP90, WCL91, CBB+89]. A rule generally consists of an event that causes a condition to be evaluated, and if true, results in the execution of a predefined action. Events are modifications of the database, conditions correspond to database queries, and actions perform changes to the database. Sometimes the condition is omitted. Rules of this kind are often called *triggers* or *Event Condition Action* rules. They are powerful to express integrity constraints like: "the salary of an employee can only increase", or "only those departments which have no employee can be deleted".

Existing trigger languages suffer from two main drawbacks. First, the semantics of an active database rule system is not well understood. Different rule system semantics have been proposed [WCL91, SJGP90, McD89, Han89] using descriptions ranging from natural language to pseudo-code procedures. A Petri-net model is used in [ZB90] to formally compare the semantics of active database systems, but the model essentially concentrates on couplings between events and conditions and actions of rules. An imperative database programming language is used in [HJ91] to describe the semantics of rules in active database systems (e.g., Starburst). Nevertheless, most of the proposed semantics do not take advantage of well known and accepted formalisms developed for rule-based systems such as production systems in AI (e.g., OPS5), logic programming languages (e.g., Prolog), or deductive databases (Datalog-like languages). Hence, active database systems do not capitalize on existing rule-based technology (optimization techniques,

algorithms, programming environments, etc.) and important issues like: when should rules be fired ?, how should they be fired ?, how should their effects be combined ?, are not given a uniform and formal treatment.

A second weakness of existing active database rule languages is that triggers are very similar to daemons or database procedures, as specified for instance in [Cod73], i.e., trigger languages are low-level languages. This makes difficult the specification of triggers and their maintenance. It is hard to have a global view of what tasks are being performed by a set of triggers, because a user-level rule (e.g., an integrity constraint) is chopped into many triggers. For instance, consider the two relations:

Emp (name, salary, dept_no, emp_no)

Dept (dept_no, mgr_no)

The referential integrity constraint saying that: "every employee works in at least one department" will be specified by several rules respectively triggered by insertions into *Emp*, updates to *Emp* or *Dept*, and deletions from *Dept*.

Our starting point is that deductive database languages provide a good basis for defining an active database rule language. These languages have a simple and fairly well understood semantics, formally defined using fixpoint operators. They provide a formal basis to other rule-based languages like OPS5 or Prolog. Implementation techniques have been developed in a database framework, including optimization algorithms (e.g., [Sell89]) Finally, various extensions of Datalog have been proposed to obtain powerful languages. Examples of deductive database systems are described in [NT89, KMS90, BF89, PDR91].

Deduction rules can always be translated into triggers. One solution, described in [W91], is to materialize all intensional data defined by deduction rules, and specify triggers for maintaining these data whenever extensional data are updated. Practically, such simulation entails a clear space-time tradeoff. On the other hand, triggers cannot always be mapped into deduction rules. A major reason is that deductive database systems are not designed to manage events (like delete or update operations), and deduction rules cannot refer to the event's effects.

Our first contribution in this paper, is to extend a deductive database rule language, namely RDL1 [KMS90], towards an active database rule language, thereby capitalizing on deductive database technology. As a result, our language offers three main features compared to existing trigger languages. First, rules are expressed at a higher level. For instance, an integrity constraint does not need to be decomposed into as many triggers as the number of events that can violate it (a notable exception is Ariel [Han89], which also has this feature for static constraints). Second, the meaning of a set of rules is formally described by two distinct aspects: (i) the coupling between rules and events (including transaction boundaries), and (ii) the semantics of a set of rules, (on which database state are the rules executed ?, how are they executed ?). The second aspect is common to both deductive and active database rule languages. Therefore, we take advantage of the formalisms developed within the framework of deductive databases to formally characterize rule application semantics. Finally, our rule language facilitates the integration of deduction rules with *active rules*. By active rules, we mean rules that react to events. We provide a single uniform notion of rule and our rule semantics covers both the deductive database and active database paradigms. In particular, a set of rules whose evaluation is triggered by some events may use rules that deduce data in order to perform intermediate computations.

The second contribution of this paper is to propose an architecture in which the system responsible for detecting events and triggering rules is tightly coupled with a relational DBMS. Most existing architectures for active database systems integrate a rule-based system within a DBMS (e.g., Postgres, Starburst, Alert). Our approach does not require any change to an existing DBMS. Rules, or more generally modules of rules, can be dynamically defined. A newly defined module is first compiled into an executable C/SQL procedure. It also yields an incremental compilation of an environment initialization procedure. These two procedure codes are then assembled together within a specific *Toolbox*. The resulting system, called *Trigger Monitor*, is activated whenever an application program is connecting to the database system. It analyzes the successive database commands issued by the application program and automatically triggers the evaluation of rules. This coupling approach has the advantage of being flexible, and portable on various kinds of DBMSs. However, it can be less efficient than the integrated one because it cannot take advantage of low-level system features provided by the DBMS.

The paper is organized as follows. Sections 2 and 3 respectively present the syntax and semantics of our rule language. In Section 4, we describe the process and functional architectures of the Trigger monitor. The Toolbox we have implemented at INRIA is then

described in Section 5. Comparisons with related work are reported in Section 6. The last section concludes.

2. A Language for Active Rules

In this section, we extend a deductive database language, namely RDL1 [KMS90], towards a language that supports *active* rules[1]. The syntax of rules has been augmented so that rules can refer to events, and couplings between rules and events can be specified. We first specify our meaning of events and introduce the notion of delta relations. Then, the couplings between rules and events is defined. Finally, the general syntax of rules is given and illustrated.

2.1 Events and Delta Relations.

Throughout this paper, we shall consider that rule processing is part of the execution of a given transaction which can either be embedded into an application program or interactively produced by a user. That is, rules are activated and executed as a result of operations issued by a transaction. We view a transaction as a stream of operations consisting of SQL commands (like select, insert, delete, commit, ...) and non SQL database commands.

Most existing trigger languages require rules to be explicitely attached to events for which they can react, using a specific statement (e.g., "WHEN <events> ...") preceeding the specification of the rule. In our language, triggering events need not be specified by the user when defining rules. Instead, they are implicit in rule definitions and can be derived by the system at rule compile-time. We simply provide system-defined relations, called *delta relations*, that enable to refer to the effect of database events within rules's conditions. These relations record the net effects of database changes performed by SQL commands: insert, update, delete. Similar kind of relations are used in [WF90, HJ91, RCBB89]. We follow a syntax close to [WCL91] to denote delta relations.

• *Delta relations*: If T (A1,...,An) is a relation schema, then the delta relations associated with T have the following schemas:

inserted_T (A1,...,An)

deleted_T (A1,...,An)

updated_T (oldA1, ..., oldAn, A1,...,An)

Intuitively, *inserted_T* (A1,...,An) refers to the tuples currently inserted into T, *deleted_T* (A1,...,An) refers to the tuples currently deleted from T, and *updated_T* (oldA1, ..., oldAn, A1,...,An) refers to the tuples currently updated in T with their new value.

• *Properties of Delta relations*: We impose that delta relations satisfy the following:

1. inserted_T ∩ deleted_T = ∅;
2. deleted_T ∩ $\Pi_{oldA1, ..., oldAn}$ (updated_T) = ∅,
   deleted_T ∩ $\Pi_{A1, ..., An}$ (updated_T) = ∅;
3. inserted_T ∩ $\Pi_{A1, ..., An}$ (updated_T) = ∅, and
   inserted_T ∩ $\Pi_{oldA1, ..., oldAn}$ (updated_T) = ∅;
4. The *current value* of T is defined to be:
   T − [deleted_T ∪ $\Pi_{oldA1,...,oldAn}$ (updated_T)]
   ∪ [inserted_T ∪ $\Pi_{A1,...,An}$ (updated_T)]

□

2.2 Coupling Rules with Events and Transaction Boundaries

As noted in [ZB90], a crucial point in the specification of triggers, is to express how rule execution relates to events, including those that mark the transaction boundaries (commit, exit, rollback). Different coupling modes can be envisaged by stating when the condition (or the action) of a rule is evaluated relative to the transaction in which the triggering event is signaled.

A single coupling mode is defined in our rule system. It specifies if a rule must be evaluated either when the triggering event occurs or when the transaction reaches a commit point. In the former case, we say that the evaluation is *immediate* relative to the event that triggered it, otherwise we say that the evaluation of the rule is *deferred* until the end of the transaction. We do not provide means to specify a coupling mode in which rule evaluation is decoupled from the triggering transaction as in Hipac (see [ZB90]).

Both immediate and deferred rules are useful. For instance, immediate rules enable to detect an inconsistent intermediate database state as soon as it occurs. An immediate decision can be taken, like aborting the transaction, or issuing some compensating actions in order to ensure database consistency. Typically, the rule saying that: "the salary of an employee cannot decrease" can be checked immediately. Other rules need to be checked at the end of the transaction because they are interested in the final database state reached by the transaction

1 We shall use the words *active rule* instead of the word *trigger* to denote a rule in our active database system.

(intermediate inconsistent states w.r.t the rule are allowed). The referential integrity constraint between EMP and DEPT mentioned before is an example of deferred rule. Deferred rules can also be checked before the end of the transaction at *integrity checkpoints* (also called assertion points in [ANSI90]). We introduce a "CHECKPOINT" command that can be used within a transaction to trigger the evaluation of all deferred rules.

2.3 General Syntax of Rules

The syntax of our rules is based on that of RDL1 [KMS90], and incorporates the procedural extensions described in [KM91]. A rule consists of an if-then statement, where the if-part (also called condition) is a tuple relational calculus expression. The then-part (also called action) of the rule is a set of elementary actions, each being either a database update, a variable assignment or a procedural call (which does not involve any database update).

Rules are encapsulated within rule modules. A *module* contains a relation declaration section which defines input, output, base, and deduced relations. *Base* relations correspond to relations that are physically stored in the database. *Input* relations can be passed as arguments to a module which produces a set of *Ouput* relations as a result. Input and output relations are always extensional. *Deduced* relations are temporary (i.e., intermediate) relations computed by a module during execution. We refer to [KMS90] and [KM91] for a more detailed presentation of the rule language.

To support the declaration of active rules, the RDL1 syntax is enriched in two ways. First, the coupling mode, immediate or deferred, can be specified at the module level or at the individual rule level. Two key words, IMMEDIATE and DEFERRED, can be used just after the key word "rules", or the key-word "is", as shown in the examples below. Second, system-defined relations can be referenced in the condition part of rules.

We now present examples of rule modules and give their intuitive semantics.

Example 2.1: The module below defines a referential integrity constraint between relations EMP and DEPT.

```
module ref_constraint_emp_dept;
base EMP (name string, emp_no integer, dept_no
          integer, salary integer);
     DEPT (mgr_no integer, dept_no integer);
rules
r is DEFERRED
  if EMP (x) and not exists y in DEPT (x.dept_no =
  y.dept_no)  then - EMP (x);
  end module
```

In the above rule, tuple variable *x* ranges over relation EMP, and *y* is a quantified variable ranging over relation DEPT. Intuitively, this module defines an active rule that is activated whenever the EMP or DEPT relations are modified. Here, EMP and DEPT always refer to the *current* values of the employee and department relations. Thus, if an employee with no department is inserted it will be rejected (i.e., deleted from the set of employees to be inserted). If a department that has employees working in it is deleted then all its employees will be deleted. As said before, triggering events are *not* specified by the user but are rather implicit. A crucial point is to determine how triggering events (e.g., insert to EMP, delete from DEPT) can be derived from rule conditions.

Next example shows that one can also explicitly refer to event's effects within rules.

Example 2.2: This example is borrowed from [WCL91]. We test if any inserted or updated employee has a salary greater than 100. If true, the action sets the salaries of all inserted employees to 50 and reduces each existing employee's salary by 10% if it is greater than 100.

```
module salary_control;
var integer change;
base EMP (name string, emp_no integer, dept_no
integer, salary integer);
rules DEFERRED
r1 is
if (exists z in Inserted_EMP (z.salary > 100)) or (exists
z in updated_EMP (z. salary > 100))
thenonce change = 1;

r2 is
if inserted_EMP (x) (change = 1)
thenonce -/+ EMP (x; salary = 50);

r3 is
if EMP (x) (change = 1 and x.salary > 100)
then  -/+ EMP (x; salary = .9 * x.salary);

control priority (r1, r2, r3)
init {change = 0;}
end module
```

We use a global variable, *change*, to enable and disable the changes to EMP performed by r2 and r3. The variable is initialized in the "init" section, and then updated in the action part of rule r1. In fact, this variable simulates a rule r1 saying: "if <r1's condition> then

(execute r2; execute r3;)*". Anticipating the description of our procedural control language in Section 3.3, the control string "priority (r1, r2, r3)" indicates that r1 has priority over r2 which has priority over r3. When the module runs, rule r1 is fired first. The "thenonce" keyword means that if the rule is fired then it will never fire again. If the value of variable *change* is set to 1, rule r2 fires and all employees in inserted_EMP are updated with a salary equal to 50. Then rule r3 recursively updates employees who have a salary greater than 100 in EMP (i.e., the EMP relation and its associated delta relations). Notice that here again, triggering events are not specified by the user. Instead, the user may refer in the rules to the cumulated effect of previous events.

For instance, suppose the situation is as follows. EMP contains two tuples, Bob with salary 90K and Alice with salary 120K. Suppose the transaction inserts a new employee Joe with salary 110K and updates Bob's salary to 110K. What happens ? Rule r1 sets the value of change to 1. Then, rule r2 deletes Joe from inserted_EMP and inserts instead tuple (Joe, 50K). Rule r3 deletes Bob from updated_EMP and inserts instead tuple (Bob, 90K, Bob, 99K), and inserts tuple (Alice, 120K, Alice, 108K) into updated_EMP. Finally, r3 deletes Alice from updated_EMP and inserts instead (Alice 120K, Alice 97.2K).

The last example demonstrates a deduction capability.

Example 2.3 : We define a deduced relation *Manages* (sup, sub) to contain the management hierarchy of the company using the rules r1 and r2 (transitive closure of a relation obtained by joining relations EMP and DEPT).

```
module recursive_rule ;
base  EMP (name string, emp_no integer, dept_no
integer, salary integer);
      DEPT (mgr_no integer, dept_no integer);
deduced  MANAGES (sup integer, sub integer) ;
rules IMMEDIATE
r1 is
if DEPT (x)  and EMP (y) (x.dept_no = y.dept_no)
then + MANAGES (sup = x.mgr_no, sub = y.emp_no) ;

r2 is
if MANAGES (x) and MANAGES (y)  (x.sub = y.sup)
then + MANAGES (sup = x.sup, sub = y.sub) ;
end module
```

As in RDL1 [KMS90], an SQL select operation on MANAGES will *immediately* activate the two rules. The transformation of the select query is detailed in Section 4.3.2. This example shows that there is a single notion of rule for both the deductive database and active database paradigms.

3. Semantics of Rules

As mentioned before, rules are activated and executed as a result of events issued by a transaction. The semantics of our rule system is described in three steps. First, in Section 3.1, we describe when rules are activated with respect to the events of the transaction. Second, in Section 3.2, we define how a given set of activated rules is executed using a partial fixpoint operator. Finally, the notion of procedural control over a set of rules is introduced in Section 3.3 and the control language is presented.

3.1 Activation of Rules

The way rules are activated with respect to the events of a transaction is described by a recursive function *evaluate*, which takes as parameters a stream of events and a database state. The *execute_imm* function computes the partial fixpoint[2] of a database instance using some immediate rules. Finally, the *execute_diff* function computes the fixpoint of a database instance using some deferred rules. In the following, we use the abbreviations: T_X for a transaction, R for a rule base, and I for a database instance (including delta relations). Also, we denote e(I) the database instance where delta relations in I have been updated accordingly to event e, the notation x.S, means that x is the first element of a stream S, and [] denotes the empty stream.

evaluate ([], I) = execute_diff (R, I)

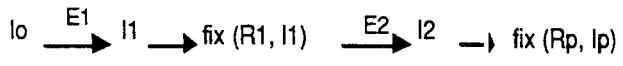
evaluate ($e.T_X$,I) = evaluate (T_X, execute_imm (R, e(I)))

```
% at each invokation of evaluate: event e
% is processed and immediate rules are
% evaluated
```

The evaluation process can be depicted using a graphical notation close to the one of [WF90]. Let I_0 stand for the initial database state, we denote E_i the i^{th} event of transaction T_T, and fix (R, I_j), a fixpoint of I_j using rules in R (if the rule execution terminates). We assume that T_X contains p events and that R_i denotes the set of rules actually activated by event E_i. If E_p is the event associated with the "commit" action of T_X, then R_p is the set of deferred rules. If $i \neq p$ then R_i is the set of immediate rules.

---

2 Indeed, this is a partial fixpoint because neither the termination of the execution nor the unicity of the result can be guaranteed for general rule programs [AS91]. For simplicity, we abusively use the word fixpoint.



On this diagram, event E_1 maps the initial state into a new state I_1 in which delta relations, initially empty, may contain some tuples (if the event is an SQL select then $I_0 = I_1$). Then, all immediate rules are used to compute the fixpoint of I_1. The next event is then processed, and so on so forth, until the end of the transaction is reached.

3.2 Partial Fixpoint Semantics of Rules

In this section, we concentrate on the meaning of a set of rule modules. We first define the notion of immediate consequence of a database state using a rule instantiation. Let I be a database state, and r be a rule. An *instantiation* of r, henceforth r', is a rule in which every free variable ranging over a relation T has been substituted by a tuple in the "current value" of T in the state I. If T is not a delta relation, then the "curent value" of T, denoted by *current_T*, in I is: $(T - \Delta T^-) \cup \Delta T^+$ where ΔT^- refers to all tuples currently deleted from T, ΔT^+ refers to all tuples currently inserted into T. Delta relations ΔT^+ and ΔT^- are required to satisfy the properties given in Section 2.1, in particular: $\Delta T^- \cap \Delta T^+ = \emptyset$. Formally, we shall treat an update of a relation T as a deletion from T and an insertion into T.

If r' is such that its condition part is true in the state I, then the action part of r' is called an *immediate consequence* of I using r'. Given a rule r, Imm_Cons (r,I) is defined to be the set of all the immediate consequences of I using instantiations of r.

Note that there is a very simple way of constructing Imm_Cons (r,I). Suppose that r's condition has q free variables ranging over relations T_1, ..., T_q (not necessarily pairwise distincts). The set of all tuples in the product $T_1 \times T_2 \times ... \times T_q$ that satisfy the condition part of r, is first retrieved using a relational query. This returns the set of all instantiations of r that satisfy the condition part. Imm_Cons (r, I) is then obtained by projecting these instantiations on the attributes of the relations that appear in the action part of r.

• *Set-oriented semantics*. The set of rules R of a module defines a *relation* among database instances as follows. For each state I, J = R(I) if for some rule r in R, J is such that:

1. If current_T(t) is in I, and + T(t), - T(t) are both in Imm_Cons (r, I), then current_T(t) is in J.
2. If + T(t) is in Imm_Cons (r,I) and - T(t) is not in Imm_Cons (r,I), then: if $\Delta T^-(t)$ is not in I then $\Delta T^+(t)$ is in J otherwise $\Delta T^-(t)$ is not in J.
3. If - T(t) is in Imm_Cons (r,I) and + T(t) is not in Imm_Cons (r,I), then: if $\Delta T^+(t)$ is not in I then $\Delta T^-(t)$ is in J otherwise $\Delta T^+(t)$ is not in J.
4. If current_T(t) is in I and - T(t) is not in Imm_Cons (r, I), then T(t) is also in J

If the sequence R(I), R(R(I)), ... has a limit, it is denoted *fix (R,I)*. []

Intuitively, this definition reflects the facts that: (i) every relation T in the condition part of a rule refers to the current value of T, (ii) if both a fact and its negation are produced by some rule, the effect of the rule w.r.t. this fact is null, and (iii) the delta relations are always pairwise disjoint sets for every relation T. Every rule is fired deterministically, but the order of firing rules is left unspecified, thereby introducing non-determinism in the computation. This semantics captures the semantics of deductive rules in RDL1 [KMS90].

• *Semantics of modules*. A module is composed from a set of rules R. A set of modules $M = \{R_1, ..., R_n\}$ defines a relation among database instances as follows. For each state I, J = M(I) if there exists j, $1 \le j \le n$, such that J = fix (R_j, I). []

Thus, modules are computed one after the other and each module is computed up to saturation before executing the next one. Ordering between modules is described in the next section.

3.3 Controlling the Execution of Rules

The execution order of rules that belong to a triggered module is specified using a procedural control language derived from [MS88]. A similar control language, called a rule algebra, has been proposed in [NT89] in the framework of the LDL language. Our control language includes basic symbols that are rule names and three primitives: sequence, saturation, and disjunction. The control language is used to declare a control string in the "CONTROL" section of a rule module. The syntax of the control language is now given.

<exp> := <rule_name> | <sequence> | <saturation> | <disjunct>
<sequence> := seq (<exp1>, ..., <exp2>)
<saturation> := [<exp1>, ..., <expn>]
<disjunct> := <exp1> + <exp2>

The *sequence* primitive means that argument expressions are evaluated in their specified order. The

saturation primitive means that argument expressions are evaluated up to saturation in any order. Finally, the *disjunct* primitive specifies an exclusive "or" between argument expressions. More formally, the semantics of these primitives is given by the *eval* function below.

eval (r) = fire r if r is firable and returns r, nil otherwise
eval (seq (<exp1>, ..., <expn>)) = eval (<exp1>); ... ;
eval (<expn>); ...
eval ([<exp1> ..., <expn>]) =
repeat eval (<expi>), i ∈ {1, ..., n}
until all <expi> evaluate to nil
eval (<exp1> + <exp2>) =
eval (<exp1>) or eval (<exp2>)
<exp>; nil = <exp>

Example 3.1: Consider the control string: s = seq (r1 + r2, r3). This is interpreted as: r1 or r2 is first evaluated and then r3 is evaluated. []

Because *priorities* between rules are often useful, we introduced a special key-word *priority* such that priority (<exp1>, ..., <expn>) expresses that <exp1> has priority over <exp2> which has priority over ... over <expn>. Formally, priority (<exp1>, ..., <expn>) is defined by:

[seq ([seq ([seq ... ([seq ([<exp1>], <exp2>)], ..., <expn>)]
<— (n - 1) times —>

Example 3.2: The control string: s = [priority (r1, r3), priority (r1, r2)], expresses that r1 has priority over both r2 and r3, but no priority exists between r2 and r3. []

Two kinds of *default priorities* between rules are allowed. First, if no control string is specified in a rule module, rules are evaluated in their specification order and every rule is executed up to saturation. Now, suppose that a control string s only contains some of the rules composing the module and that rules r1 ... rk do not occur in s. The *partial_eval* function is defined to evaluate such a control string. Formally, we have:

partial _eval (s) = eval (priority (s, [r1] + [r2] + ... + [rk]))

Essentially, the *partial_eval* function enforces that the control string has always priority over the other rules. Suppose that s is evaluated up to saturation then s will be evaluated again.

Finally, a *default ordering* relationship is defined between modules triggered at the same time. This ordering expresses that the least recently created module is executed first.

Our control language is more powerful than a priority system as proposed in Starburst [ACL91, WCL91]. For instance, a simple ordering like: "fire r1 once, fire r2 once, fire r3 once, and repeat this up to saturation", is not expressible with priorities as soon as recursive rules are allowed. In fact, our language enables to describe any sequential computation of a set of rules.

A limited control language can be compensated by expressing control within rules (e.g., using temporary relations that play the role of control predicates). Our desire to have control separated from rules as much as possible has influenced the design of a powerful control language. Note that in Example 2.2, we use control both within rules and with a control string.

4. Rule System Implementation

This section presents the functionality and the architecture of an active database system resulting from the specification of a set of rule modules.

4.1 Basic Assumptions and Design Decisions

A number of important decisions underly the architecture of our active database system: (i) a rule base is *compiled* into an executable system called Trigger Monitor which automatically activates and executes rules depending on the actions taken by an application program, and (ii) the Trigger Monitor is *coupled* with a relational database system.

Most current implementations of active database systems integrate rule processing within an existing DBMS (e.g., Postgres and Starburst rule systems). This should yield efficiency because the implementation of rule processing can take advantage of low level system capabilities like attachments in Starburst [WCL91], or tuple markers in Postgres [Ston90].

However, based on our previous experience in developing an integrated deductive rule system [KMS90], we believe that the integrated approach suffers from two drawbacks. First, the integrated system is hard to maintain and to change because its implementation is specific to the extended DBMS. Active database rule languages differ significantly in their semantics, and no sufficient experience has been gained in order to agree on a common semantics. Existing rule languages are then evolving and changing their semantics may require considerable changes in the implementation if it is made too dependent on the usage of low level system features. A second point is

heterogeneity. The integrated approach has the drawback of being not portable. On the other hand, the coupled approach facilitates the implementation of a rule system on different DBMSs that accept a common interface protocol like SQL (which is the case of relational DBMSs and some object-oriented DBMSs). We argue that portability on multiple, existing SQL systems is *the* advantage of our approach.

We assume a client-server architecture where an application program is linked with a library of communication procedures to interface a DBMS server. We consider a typical library including procedures like *SqlConnect*, *SqlDisconnect*, *SqlRead*, and *SqlExec*. The *SqlConnect* and *SqlDisconnect* procedures respectively open and close a connection between the application process and a DBMS process. The *SqlExec* procedure takes an SQL command as input and transmits it to the corresponding DBMS process. Such communication procedures may vary from one DBMS to another. However, our library can be easily emulated on existing DBMS.

4.2 Process Architecture of the Trigger Monitor

The Trigger Monitor is an executable program that automatically activates and executes rule modules according to the operations performed by an application program. This program results from the compilation of a rule base. In this section, we describe the process architecture.

Since we assume no change on the underlying DBMS, the communication between an application process and a DBMS process must be intercepted by the Trigger Monitor. This is achieved by using renamed communication procedures to establish and relax the connection between the application and the DBMS. The *SqlConnect* procedure call is replaced by an *SqlConnect** procedure call that creates a Trigger Monitor process instead of a DBMS process, at application start-up time. Communication with a local or remote DBMS process is then established by the Trigger Monitor. Thereafter, the Trigger Monitor intercepts all commands issued by the application to the DBMS. A Trigger Monitor process is created for every application process and interfaces the DBMS process which the application process communicates with.

The Trigger Monitor and the application processes reside on the same client workstation. Figure 4.1 depicts the run-time process architecture.

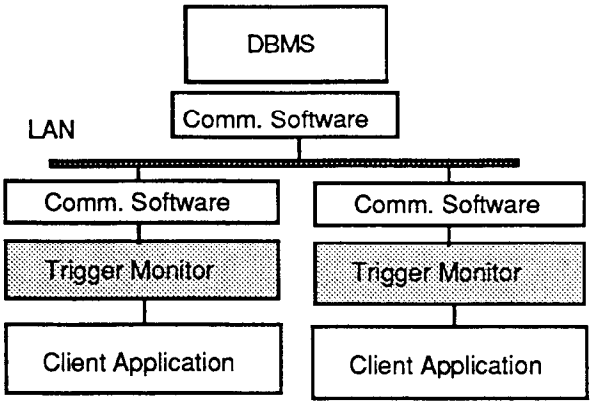


Figure 4.1 : Run-time Process Architecture.

4.3 Functional Architecture of the Trigger Monitor

4.3.1 General Structure

The Trigger Monitor is functionnally decomposed into three main components: the environment initialization, the event handler, and the rule evaluator. The pseudo-code procedure below describes the logical structure of the Trigger Monitor.

```
Trigger Monitor
begin
 create_DBMS_process();
 init_Environment();
 while (an SQL EXIT command is not issued by the
     application) do
     event = read_Client_Event();
     Handle_Event();
     send_Result_to_Client();
 endwhile
end
```

The *Init_environment* procedure performs two tasks. First, it builds a data structure describing all the delta relations that should be managed for executing rules in the rule base. For instance, if there is a rule referencing the EMP relation or any of the delta relations associated with EMP then the three delta relations associated with EMP have to be managed. The second task is to build a data structure containing the names of all the rule modules that make the rule base.

The *read_Client_Event* procedure is a (simplified) SQL parser that analyzes incoming database statements (e.g., Sql_Exec). The parser isolates the SQL commands. If the command updates the database, it determines which relation is updated and which relations participate in the command.

4.3.2 The Event Handler

The Event Handler performs a case analysis of the SQL commands read by the *read_client_Event* procedure. If the command is a SELECT involving deduced relations, then all modules participating in the definition of the deduced relations are executed. A modified SELECT statement in which deduced relations are replaced by the temporary relations containing their extensions, is sent to the DBMS. If the SELECT only involves base relations, it is issued to the DBMS and the result is returned to the client application. If the command is an UPDATE, it is sent to the DBMS. We assume that the result of an SQL data manipulation command can be stored as a temporary relation; a special command "NAME" assigns a relation name to the last query result. In the case of an UPDATE command, the temporary relation returned by the system only contains the updated tuples. A specific treatment is then necessary to build the delta relation associated with updates. When the command "NAME" is used, a specific variable indicates the number of tuples in the temporary relation created by the command. This number indicates that an update has changed the database state. If so, the *Manage_Update* procedure updates the corresponding delta relations, if any, according to the set-oriented semantics described in Section 3.2. Then, the Rule Evaluator executes immediate rules.

If the command issued by the application program is a COMMIT or a CHECKPOINT, deferred rules are evaluated and then the query is sent to the DBMS (only in the case of a COMMIT). Below, we give a non-exhaustive description of the analysis performed by the Event Handler.

```
Handle_Event(event) {
switch (event.type)
 case UPDATE:
  send_Query_to_DBMS();
  receive_Result_From_DBMS();
  if (event.result.tupleCount > 0) {
  manage_Update(event.updatedRelation,
      UPDATE);
  evaluate_rules(event);}
 /* INSERT, DELETE similar to UPDATE */
 case SELECT:
  if (query involves deduced relations)
  {evaluate_rules(event);modify_query;}
  send_Query_to_DBMS();
  receive_Result_From_DBMS();
 case COMMIT:
  execute_deferred_rules;
  send_Query_to_DBMS();
  receive_Result_From_DBMS();
  reset_All_Events();
 case ROLLBACK:
  send_Query_To_DBMS();
  receive_Result_From_DBMS();
  reset_All_Events();
 default:
  send_Query_To_DBMS();
  receive_Result_From_DBMS();
}
```

4.3.3 Evaluation of Rules

The evaluation of rules is part of the *evaluate_rules* and *execute_deferred_rules* procedures. We essentially describe the former procedure since the evaluation of rules is done similarly in the second procedure. The *evaluate_rules* procedure cycles over the set of compiled rule modules and successively invokes the program resulting from the compilation of each rule module (by the Rule Compiler) until the database does not change.

```
evaluate_rules (E: event);
/* E is represented by delta relations */
while the database changes {
 execute_module[i] (E), for all modules i;}
```

We now detail the execution of a module. Three phases are distinguished. First, the sensitivity of the module with respect to the *current cumulated event* is tested. This event is represented by the state of the delta relations. For instance, if there is a non empty delta relation associated with relation T and T occurs in a rule r, then the module is sensitive to the event. Notice that T may either occur in the condition or action part of r. This test is produced by inspection of the rules in the module at the time the module is compiled by the Rule Compiler.

If a module is relevant, then the second phase consists of building a specific data structure, called Production Compilation Network (PCN) in main memory. This structure describes the relationships between relations, main memory variables, and rule conditions [MS88].

The third phase is the execution of rules using the PCN structure. A rule is selected according to the control strategy specified in the module (or the default strategy if no strategy has been specified) and evaluated. If the rule is fired then delta relations, temporary relations, and main memory variables assigned in the rule are updated. A next rule is then selected and fired until no more rule is firable. Contextual data structures (temporary relations, PCN) are then updated. This processing is summarized below.

```
execute_module[i] (E: event); {
  test_module_relevance (E);
  init_PCN ();
  init_Control ();
  select_firable_rule ();
  while there exists a firable rule
  /* the choice between immediate and
  deferred depends on the event E */
     {fire_rule ();
      update_delta_relations ();
      monitor changes to main memory
      variables;
      select_firable_rule ();}
     free temporary relations no longer
      needed;
     maintain_PCN ();}
```

Suppose that a module which has already been executed is considered again for execution. The system (in the *evaluate_rules* procedure) checks whether the database state over which the module executes has changed since its last execution. If not, the system considers the next module.

5. A Toolbox for Generating the Trigger Monitor

In this section, we describe a toolbox which takes as input a set of rule modules and generates a Trigger Monitor.

The Toolbox consists of several software components. Two levels of compilation are used to generate a Trigger Monitor from a set of rule modules. At the first level, a Rule Compiler compiles each source module into a C/SQL procedure, and an Environment Compiler generates the *Init_Environment* procedure mentioned before. The second level of compilation then follows. A standard makefile facility is used to generate a Trigger Monitor from the output of the first compilation phase, the Event Handler, the Interface Procedures (like SqlConnect* described before), and user-supplied C procedures invoked in rule modules.

Changes to a rule requires to rebuild the Trigger Monitor. Since rules are organized into modules, only those modules which have been updated need to be recompiled. The initialization procedure has also to be recompiled. The Trigger Monitor is then reassembled from linking together the set of compiled modules.

The functional architecture of the Toolbox is depicted on Figure 5.1. Square boxes represent the compilers and the makefile facility. Grey circle boxes represent the user-provided components.

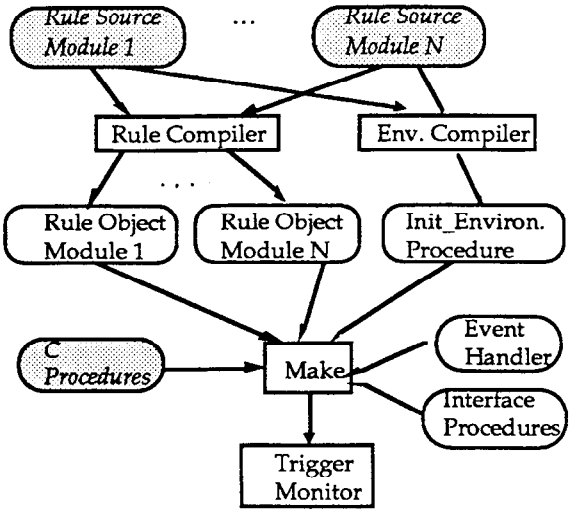


Figure 5.1: Functional Architecture of the Toolbox.

6. Comparisons with Related Work

This section briefly surveys previous work on active database systems and relates it to our work.

Alert is an extension architecture designed for transforming a passive SQL DBMS into an active DBMS [SPAM91]. Alert rules are SQL queries (called active queries) which are defined over active tables. Active tables are append-only tables created by the user in order to record events. Therefore, events can be general and are not limited to built-in operations like SQL insert, ... Active queries differ from usual SQL queries in their cursor behaviour. When a cursor is opened for an active query, tuples added to the underlying active table after the cursor was opened contribute to the query. Thus, rules wait for tuples to be appended to the active table and are instantiated with each new tuple (i.e., a rule is executed in a tuple-oriented fashion). Unlike our system, format of events must be declared by the user (in active tables) and rules are explicitely attached to these events. Alert provides several coupling modes between events and rules. Rules can run in the same or in separate transactions as the triggering transaction. The triggering transaction can be halted for the execution of triggered transaction or it can be run in parallel. Finally, a rule can be immediate or deferred. Coupling modes are specified separately from the rules using a command activate. Thus, a rule can be activated with different coupling modes. Unlike our system, rules are executed in a tuple-oriented fashion. however, the semantics of a set of rules is not formally defined and interaction between rules is not clear. Therefore it is not easy to see how an

arbitrary rule system could be supported by the Alert architecture.

Triggers in Starburst [WCL91] are expressed using set-oriented production rules where conditions are relational expressions and actions consist of sequences of SQL commands. Triggering events are associated with the built-in operations: update, insert, delete, and they are explicitely attached to each rule. Unlike our language, all rules are deferred and evaluated at the end and as part of the triggering transaction. Events are implemented using transition tables that are similar to our delta relations. A rule is executed with respect to the net effects of the transaction (including the effects of rules already executed). However, unlike Hipac and our system, the net effects are computed separately for each rule according to the last time the rule was executed. The idea is to prevent a rule from being fired twice with the same tuples in transition tables. This semantics is very similar to the notion of *refraction* used in OPS5 [BFMK85]. Our language enables to simulate this behaviour using control predicates in rules. Finally, control between rules is expressed using a priority mechanism [ACL91].

In Hipac [DBB+88, CBB+89], triggers are specified as event-condition-action statements. Events can be built-in (including timing events, hardware signals), or user-defined. Events can be composed using a specific event language. Changes made by database operations in a transaction are kept into delta relations similar to ours. Like in our system, delta relations record the net effects of database changes. Hipac also offers a rich variety of coupling modes (including those of Alert) [ZB90]. Howewer, as noted in [SPAM91], it is not clear to see which coupling modes are essential and which ones simulate some form of control over rules. Hipac's execution model is a nested transaction model, and an assignment of condition evaluation and action execution to transactions based on coupling modes. As a result, there is no conflict resolution policy that chooses a single rule to fire, or a serial order to fire the rules. Instead, all the rules fire concurrently as subtransactions [ZB90]. This semantics makes the expression of control between rules difficult to express.

The Postgres rule language PRSII has a syntax quite close to that of Starburst [SJGP90]. Similar to Hipac, rules consist of event-condition-action triples and are low-level statements. Similar to Starburst, events correspond to built-in database operations: select, insert, delete, ... PRSII allows a single coupling mode between rules and events: rules are immediate and are executed within the triggering transaction. Unlike Starburst and Hipac, but like Alert, rules are tuple-oriented. When an individual tuple is accessed, updated, inserted or deleted in a transaction, then the transaction appropriately instantiate the triggered rules and execute them concurrently. A special algorithm uses special locks to mark tuples or table columns whose changes or retrievals would trigger one or more rules. Thus, there is no notion of delta relations. Unlike our system, PRSII does not provide a control language over rules, or a priority system like in Starburst. PRSII enables to define a rule as an exception to another rule.

7. Conclusion

We have presented an extension of a deductive database language, namely RDL1, towards rules that react to events. Events consist of built-in database operations (select, insert, delete, update). The net effects of database operations are recorded into delta relations. These relations can be used in rule's conditions. Our language has the following features. First, unlike Hipac, Alert, Starburst and PRSII, our rules are expressed at a high level. Triggering events are not provided by the user but are instead derived from rules by the system. Second, our rule system is formally described by means of a partial fixpoint operator, which encompasses both the deductive database and active database paradigms. Hence, a rule module may consist of rules that deduce data and rules that modify the database as reaction to external events. In this formal framework, existing work on rule-based systems can be reused. Finally, we presented a control language that enables to specify a rich variety of rule execution orderings.

We have also presented a system architecture in which the system responsible for detecting events issued by application programs and triggering rules is front-ended to an existing relational database system. This approach can be used over any relational DBMS which supports run-time interpretation of SQL commands. A major feature of our approach is that it enables to rapidly develop rule modules over an *existing* database.

Two research issues are envisioned in the next future. One is the development of an optimizer integrated within our Rule Compiler. Second, we wish to incorporate error and exception handling mechanisms in the rule language and study various alternative ways of implementing them.

Acknowledgements: We would like to thank Rakesh Agrawal, Patrick Valduriez, Allen van Gelder, and Jennifer Widom for their detailed comments and

suggestions that greatly contributed to improve the paper.

References

[ANSI90] ISO-ANSI Working Draft: Database Language SQL2 and SQL3; X3H2/90/398; ISO/IEC JTC1/SC21/WG3, 1990.

[AS91] S. Abiteboul, E. Simon : "Fundamental Properties of Deterministic and Non-deterministic Extensions of Datalog", *Journal of Theoretical Computer Science*, 78, pp 137-158, 1991.

[BF89] J. Bocca, J. C. Freytag : "Rules for Implementing Very Large Knowledge Base Systems", *Sigmod Record, 18(3):* , Sept. 89.

[BFKM85] L. Brownston, R. Farrel, E. Kant, N. Martin: "Programming Expert Systems in OPS5: An introduction to Rule-Based Programming", Addison-Wesley, 1985.

[CBB+89] S. Chakravarthy, B. Blaustein, A. Buchmann et al. : "HIPAC : A Research Project in Active, Time-Constrained Database Management. Final Technical Report, Xerox Advanced Information Technology, May 1989.

[CW90] S. Ceri, J. Widom : "Deriving Production Rules for Constraint Maintenance", in *Proc. of Int. Conf. on VLDB*, Brisbane, Australia, Aug. 1990.

[CW91] S. Ceri, J. Widom: "Deriving Production Rules for Incremental View Maintenance", *Proc. of Int. Conf. on VLDB*, Barcelona, Spain, Aug. 1991.

[Cod73] CODASYL Data Description Language Committee, CODASYL Data Description Language Journal of Development, June 1973

[DBB+88] U. Dayal, B. Blaustein, A. Buchmann et al. : " The HiPAC Project : Combining Active Databases and Timing Constraints", *ACM SIGMOD RECORD Vol. 17, N°1*, March 1988.

[Han89] E.H. Hanson : "An initial report on the design of Ariel : A DBMS with an integrated production rule system" in [Sell89]

[HJ91] R. Hull, D. Jacobs : "Language Constructs for Programming Active Databases", *Proc of Int. Conf. on VLDB*, Barcelona, Spain, Sept. 1991.

[KMS90] G. Kiernan, C. de Maindreville, E. Simon : "Making Deductive Database a Practical Technology: A Step Forward", *Proc. of Int. Conf. SIGMOD*, Atlantic City, June. 1990.

[KM91] G. Kiernan, C. de Maindreville : "Compiling a Rule Database Program into a C/SQL Application" *Proc of 7th international Conference on Data Engineering*, Kobe Japan, 1991.

[McD89] D. McCarthy, U. Dayal: "The Architecture of an Active Database Management System", *Proc. of Int. Conf. SIGMOD*, June 89

[MS88] C. de Maindreville, E. Simon : "Modelling non-deterministic Queries and Updates in a Deductive Database", *Proc. of Int. Conf. on VLDB*, Los Angeles, Aug. 1988.

[NT89] S. Naqvi, S. Tsur : "A language for Data and Knowledge Bases", book, *W.H. Freeman*, 1989.

[PDR91] G. Phipps, M.A. Derr, K.A. Ross: "Glue-Nail: A deductive Database System", *Proc. of ACM SIGMOD Int. Conf.*, Denver, Colorado, May 1991.

[RCBB89] A. Rosenthal, S. Chakravarthy, B. Blaustein, J. Blakeley : "Situation Monitoring for Active Databases", in *Proc. Int. Conf. on VLDB*, Amsterdam, Aug. 1989.

[Sell89] T. Sellis (editor), SIGMOD Record, Special Issue on Rule Management and Processing in Expert Database Systems, 18 (3), Sept. 1989.

[SJGP90] M. Stonebraker, A. Jhingran, J. Goh, S. Potamianos : " On Rules, Procedures, Caching and Views in Data Base Systems",*Proc. of SIGMOD*, Atlantic City, June. 1990.

[SPAM91] U. Schreier, H. Pirahesh, R. Agrawal, C. Mohan : "Alert : An Architecture for Transforming a Passive DBMS into an Active DBMS", *Proc of Int. Conf. on VLDB*, Barcelona, Spain, Sept. 1991.

[WF90] J. Widom, S. Finkelstein : "A Syntax and Semantics for Set Oriented Production Rules in Relational Databases, " *Proc. of Int. Conf. SIGMOD*, Atlantic City, June. 1990.

[WCL91] J. Widom,R.J. Cochrane, B.G. Lindsay : "Implementing set-oriented production rules as an extension to Starburst", *Proc of Int. Conf. on VLDB*, Barcelona, Spain, Sept. 1991.

[W91] J. Widom : "Deduction in the Starburst Production Rule System" IBM Almaden Research Report, May 1991.

[ZB90] D.R. Zertuche, A. Buchmann : "Execution Models for Active Database Systems: A Comparison". GTE Research Report TM-0238-01-90-165.