Active Rules based on Object-Oriented Queries

Tore Risch
torri@ida.liu.se

Martin Sköld
marsk@ida.liu.se

Department of Computer and Information Science
Linköping University
Sweden

Abstract

We present a next generation object-oriented database with active properties by introducing rules into OSQL, an Object-Oriented Query Language. The rules are defined as Condition Action (CA) rules and can be parameterized, overloaded and generic. The condition part of a rule is defined as a declarative OSQL query and the action part as an OSQL procedure body. The action part is executed whenever the condition becomes true. The execution of rules is supported by a rule compiler that installs log screening filters and uses incremental evaluation of the condition part. The execution of the action part is done in a check phase, that can be done after any OSQL commands in a transaction, or at the end of the transaction. Rules are first-class objects in the database, which makes it possible to make queries over rules. We present some examples of rules in OSQL, some implementation issues, some expected results and some future work such as temporal queries and real-time support.

Key Words: Active Database, Object-Oriented Query Language, Object-Oriented Rules

1 Introduction

A powerful query language will be an essential part of the next generation Object-Oriented (OO) database systems. When active properties are introduced into these databases, the query language should be extended to support them.

The HiPac[4] project introduced *ECA rules* (Event-Condition-Action). The event specifies when a rule should be triggered. The condition is a query that is evaluated when the event occurs. The action is executed when the event occurs and the condition is satisfied.

In Ariel[6] the event is made optional, making it possible to specify *CA rules*, which use only the condition to specify *logical events* which trigger rules. Rules in OPS5[1] and monitors in [8] have similar semantics. In ECA rules the user has to specify all the relevant *physical events* in the event part. We believe that CA rules are more suitable for integration in a query language, since they are more declarative. CA rules make physical events implicit, just as a query language makes database navigation implicit.

We define active rules by extending the OO query language OSQL of Iris[5]. OSQL is based on functions for associating stored and derived attributes with objects. OSQL permits functional overloading on types, and types and functions are first-class objects. Likewise, rules are first-class objects in the database too[3]. This makes it possible, e.g., to make queries over rules. By implementing rules on top of OSQL, overloaded and generic rules are possible, i.e. rules that are parameterized and that can be instantiated for different types. We also utilize the optimizations performed by the OSQL compiler[7].

Each rule is defined by a pair <Condition,Action>, where the condition is a declarative OSQL query and where the action is an OSQL database procedure body. The rule language thus permits CA rules, where the action is executed (i.e. the rule is triggered) whenever the condition becomes true, similar to OPS5 and Ariel. Unlike those systems, the condition can refer to derived functions (which correspond to views). Data can be passed from the condition to the action of each rule by using shared query variables. By quantifying query variables set-oriented action execution is possible[11].

We are implementing our ideas in the research prototype AMOS[1] (Active Mediators Object System) by extending a Main-Memory version of Iris, WS-Iris[7]. OSQL queries are compiled into execution plans

[1]The AMOS project is supported by Nutek (The Swedish National Board for Industrial and Technical Development) and CENIIT (The Center for Industrial Information Technology), Linköping University

in an OO logical language. The system logs all side effect operations on the database. The rule compiler analyzes the execution plan for the condition of each rule. It then generates 'log screening filters' which check events that are added to the log. When a log event passes a log screening filter associated with a condition, it indicates that the event can cause the corresponding rule to fire. The screening of the log is often complemented with incremental evaluation[9, 10] of the condition.

Distributed execution of AMOS is being implemented too, and we plan to introduce temporal queries and real-time facilities as well.

2 Object-Oriented Query Rules

The syntax for rules conforms to that of OSQL functions as closely as possible:

create rule *rule-name param-spec* **as**
    when [*for-each-clause* | *predicate-expression*]
    do [once] *action*
where
for-each-clause ::=
    for each *variable-declaration-commalist* **where** *predicate-expression*

The *predicate-expression* can contain any boolean expression, including conjunction, disjunction and negation. Rules are activated and deactivated by:
activate rule-name ([*parameter-value-commalist*])
deactivate rule-name ([*parameter-value-commalist*])

The semantics of a rule are as follows: If an event of the database changes the boolean value of the condition from *false* to *true*, then the rule is marked as *triggered*. If something happens later in the transaction which causes the condition to become false again, the rule is no longer triggered. This ensures that we only react to logical events[2]. In the *check phase* (usually done before committing the transaction), the actions are executed of those rules that are marked as triggered. If an action is to be executed only once per activation, the rule is deactivated after the action has been executed. We can also introduce an *immediate* coupling mode[4] by instructing the system that the check phase is to be done immediately after each OSQL command.

Example 1:

The salary changes of employees and managers are to be monitored. We want to ensure that only managers can have their salaries reduced. First we define the employee and manager types and the respective income functions, where managers receive an additional bonus:

```
create type person;
create type employee subtype of person;
create type manager subtype of employee;
create function name(person) -> charstring as stored;
create function mgrbonus(manager) -> integer as stored;
create function income(employee) -> integer as stored;
create function income(manager m) -> integer i
    as select i where i = employee.income(m) + mgrbonus(m);
create employee(name,income) instances
    :joe ('Joe Smith',30000);
create manager(name,employee.income) instances
    :harold ('Harold Olsen',80000);
setmgrbonus(:harold) = 10000;
```

Then we define procedures for what to do when a salary is decreased:

```
create procedure compensate(employee e)
    as set income(e) = previous income(e); /* employee income cannot be decreased */
create procedure compensate(manager); /* dummy procedure, managers are not compensated */
```

[2]To support physical events the system should provide functions that change values whenever a physical event occurs and thus can be referenced in the condition of a rule.

The function **compensate** uses the system operator **previous** to fetch the value of a function at the previous checkpoint.

Finally we define the rule to detect decreasing salaries for all employees:

```
create rule no_decrease() as
    when for each employee e
    where income(e) < previous income(e)
    do compensate(e);
```

Activate the rule:

```
activate no_decrease();
```

If an employee that is not a manager gets his salary decreased, the rule will automatically set the salary back to the old value at check time:

```
set income(:joe) = 20000; /* => reset income(:joe) to 30000 at check time*/
```

Note: Since the rule is defined for all employees, and manager is a subtype of employee, the rule is overloaded for managers. (Because the functions **income** and the procedure **compensate** are overloaded). If a person of type manager gets a salary reduction, no action is taken. This is an example of a set-oriented rule. The action is executed for every binding of the universally quantified variable e for which the condition is true.

Example 2:

Rules can be parameterized and instantiated with different arguments. Take a rule that ensures that a specific employee has an income below a certain maximum income, and the transaction is rolled back if an employee receives an income above the threshold. This maximum income is fixed for all employees, but can vary for individual managers.

```
create function maxincome(employee) -> integer
    as select 50000;
create function maxincome(manager) -> integer as stored;
create rule exceeding_maxincome(employee e) as
    when income(e) > maxincome(e)
    do rollback;
```

Set the income limit for Harold:

```
set maxincome(:harold) = 120000;
```

Activate the rule for a particular employee Joe and manager Harold:

```
activate exceeding_maxincome(:joe);
activate exceeding_maxincome(:harold);
set income(:joe) = 75000; /* rollback at check time because 75000 > 50000 */
set maxincome(:harold) = 90000; /* rollback at check time because 90000 + 10000 > 90000 */
set mgrbonus(:harold) = 45000; /* rollback at check time because 80000 + 45000 > 120000 */
```

It is non-trivial to determine the physical events that trigger an OSQL rule with many interdependent and overloaded functions, such as the rule above. Hence we let the compiler determine this. This illustrates the convenience of CA rules.

Example 3:

Since types are first class objects, one can write generic rules that are instantiated for a specific object type:

```
create rule exceeding_maxincome(type t) as
    when for each employee e
    where typesof(e) = t and
          income(e) > maxincome(e)
    do rollback;
```

Activate the rule for all managers:

```
activate exceeding_maxincome(typenamed('manager'));
```

Since rules are first-class objects in the database, one can make queries over rules. For example, the system could provide a function that returns all active rules dependent on a certain object type or a function that takes a rule as argument and returns all the functions it depends on.

3 Expected results

The extension of OSQL with rules is expected to give a powerful language to express active properties in an object-oriented database. The overloading of rules provides a way to specify reusable rules that can be applied uniformly in different situations. One of the goals in the project is to investigate if CA rules can be implemented as efficiently as ECA rules. This involves efficient event detection as well as incremental evaluation of rule conditions. We will verify the applicability of OO rules by investigating how they can be used for various applications, e.g. in CIM.

4 Future work

Temporal rules can be introduced by having functions that vary over time and by time-stamping events in the database. The condition can then refer to the time when a certain event occurred. By introducing a timer event, a rule can be triggered at a certain time. These extensions do not support all the possible reasoning that can be made in an event algebra such as [2]. However, it allows for reasoning about whether one event happened before another or vice versa (by comparing time-stamps).

Introducing real-time in the database would require to take the cost of executing an action into account. Active database facilities are important for real-time applications that, e.g., monitors combinations of sensor data and perform actions whenever 'interesting' situations occur. The rule language will need to be complemented with timeliness constraints for rule conditions and actions.

References

[1] Brownston L., Farell R., Kant E., Martin A.: Programming Expert Systems in OPS5, *Addison-Wesley, Reading Mass.* 1986

[2] Chakravarthy S., Mishra D.: An Event Specification Language (Snoop) for Active Databases and its Detection, *UF-CIS Technical Report, TR-91-23*, sept. 1991

[3] Dayal U., Buchman A.P., McCarthy D.R.: Rules are objects too: A Knowledge Model for an Active, Object-Oriented Database System, *Proc. 2nd Intl. Workshop on Object-Oriented Database Systems*, Lecture Notes in Computer Science 334, Springer 88

[4] Dayal U., McCarthy D., The architecture of an Active Database Management System, *ACM SIGMOD*, 1989, pp. 215-224

[5] Fishman D. et. al: Overview of the Iris DBMS, *Object-Oriented Concepts, Databases, and Applications*, ACM press, Addison-Wesley Publ. Comp., 1989

[6] Hanson E. N.: Rule Condition Testing and Action Execution in Ariel, *ACM SIGMOD*, 1992, pp. 49-58

[7] Litwin W., Risch T.: Main Memory Oriented Optimization of OO Queries using Typed Datalog with Foreign Predicates, *IEEE Transactions on Knowledge and Data Engineering* Vol. 4, No. 6, December 1992

[8] Risch T.: Monitoring Database Objects, *VLDB conf. Amsterdam* 1989

[9] Rosenthal A., Chakravarthy S, Blaustein B., Blakely J.: Situation Monitoring for Active Databases, *the VLDB conf. Amsterdam*, 1989

[10] Paige R., Koenig S.: Finite Differencing of computable expressions, *ACM Trans. Prog. Lang. Syst.* 4.3 (July 1982) pp. 402-454

[11] Widom J., Finkelstein S.J.: Set-oriented production rules in relational database system *ACM SIGMOD Int. Conf. on Management of Data* pp. 259-270, Atlantic City, New Jersey 1990