An Algorithm for the Analysis of Termination of Large Trigger Sets in an OODBMS

Thomas Weik*
Institut Praktische Informatik, TU Ilmenau
98716 Ilmenau, GERMANY

Andreas Heuer†
Computer Science Department, University of Rostock
18051 Rostock, GERMANY

Abstract

In this paper we describe an algorithm for the analysis of termination of a large set of triggers in an OODBMS. It is quite clear that, if the trigger mechanism is of sufficient complexity, the problem is undecidable. Yet, by the extensive use of object-oriented concepts, like derived classes, and lattice theory, we are able to give some sufficient conditions for termination which yield satisfying results. Another advantage of our approach is the uniform treatment of generic update operations on the one hand, and methods and abstract data types on the other.

Our algorithms are meant to be incorporated into a design tool which shows the rule designer conflicting subsets of rules. Then the designer can prove that the rules don't pose a problem for himself, or he can remodel his rules to remove the conflict.

1 Introduction

Most current OODBMS are passive, i.e. they only *react* to explicit requests by users or applications. An *active* DBMS executes operations *automatically* whenever certain events occur and/or certain conditions are met. There already are quite a few proposals for the integration of active behavior into DBMS, e.g. [Sto92], [GGD91], [Wid92], [DNP91] and [GJ92] among many others. Most of these approaches use ECA–rules which were first introduced by HiPAC ([DBM88], [DD89]). ECA means that if a certain **E**vent occurs and a certain **C**ondition is met, the DBMS automatically executes the specified **A**ction.

ECA–rules can be used for the solution of a big variety of problems in the DBMS context like automatic enforcement of dynamic integrity constraints, maintenance of materialized views and derived data, versioning, enforcement of complex authorization checking, and it can serve as a basis for implementing large, efficient, and flexible knowledge based and expert systems.

On the other hand, the introduction of ECA–rules into DBMS produces new problems, which have to be addressed. In general there are two main problems:

*e-mail: weik@PrakInf.tu-ilmenau.de
†e–mail: heuer@informatik.uni-rostock.de

Termination: If one rule's action triggers another rule (or even itself again), and this rule's action triggers a third one, the result of this might be an infinite cyclic execution of some rules in the DBS.

Confluence: If, by a complex update operation, there are more than one non–prioritized rules eligible for execution, this might give rise to a nondeterministic final database state after the execution of all triggered rules.

As stated in the abstract, one can easily verify, that the problem of detecting these properties at definition time is undecidable if the formalism for the expression of ECA–rules is of sufficient complexity.

Unfortunately, in the research community one can observe the trend, that almost all articles and reports about active DBMS are concerned with even more powerful languages for the expression of rules, which makes the analysis of the above mentioned properties more and more difficult, instead of addressing the nightmarish behavior of rules in DBMS where safety should be topic number one.

The goal of our work is to show, that, with a limited language for the expression of ECA–rules, which is able to simulate many of the complex constructs one can find in most publications about active DBMS, one can give very good sufficient conditions for termination and confluence of large sets of rules. In this article we present an algorithm for the analysis of termination. This algorithm is meant to be incorporated into an interactive tool for the development and definition of ECA–rules for large applications. The algorithm can either guarantee, that a certain set of rules will terminate, or it can isolate the rules, which might give rise to a non–terminating execution of rules and thus giving the programmer the opportunity to revise his rule design.

This article is organized as follows: In the next two sections we give a short introduction into OSCAR, the OODBMS prototype, into which our rules are incorporated, and into the syntax and semantics of our language for the expression of ECA–rules. Section 4 explains the concepts of our algorithm, followed by the comparison of our approach with related work in section 5 and some conclusions in section 6.

2 The object–oriented database model of OSCAR

As mentioned in the Introduction, these examinations are based upon the OODBMS OSCAR and especially upon the structural part of the object–oriented database model EXTREM [Heu89, HH91] and the OSCAR query operations [HFW90].

To distinguish between values and objects, we introduce disjoint infinite sets of *abstract domains* $D_{\mathcal{A}}$ representing objects in contrast to simple domains $D_{\mathcal{S}}$ consisting of values like INTEGER or STRING. Each element of an abstract domain is called an *object*, each element of a simple domain is called an *atomic value*. One special symbol in each of the domains is the null value $\perp$.

A *class* represents a (typed) set of possible objects. It can be an extensional *base class* or an intensional *derived class*. A base class either has an abstract domain or is a specialization of other classes (then it is called a *free class*). The assignment of domains to free classes is done by inheritance (see below).

```
CLASS                          Persons
             ATTRIBUTES        age :  INTEGER

CLASS                          Employees SPEC Persons
             ATTRIBUTES        salary :  REAL
                               department :  Departments
                               superior :  Employees
             UPDATE METHOD     adjust_salary()
                               MODIFIES salary

CLASS                          Applicants SPEC Persons

CONSTRAINT   DISJOINT          Employees, Applicants
```

Figure 1: Complete EXTREM scheme for the running example

Example 2.1 The example used throughout this paper collects information about persons and departments (see Figure 1). We introduce Departments and Persons as classes with an abstract domain. Since both applicants and employees are special persons which have common properties (or attributes) like Age, they are introduced as specializations (see below) of Persons.

□

An *object set* o for class C is a finite subset of the domain of C and denoted by $o(C)$. For free classes, the domains are fixed by the *class hierarchy*. The set of *specializations* is a binary relation SPEC over base classes. Each tuple of SPEC is denoted by C_1 SPEC C_2 where C_1 has to be a free class. C_1 is called *subclass*, C_2 *superclass*. The (*reflexive and*) transitive closure of SPEC is denoted by $<(\leq)$. We require $\leq$ to be a partial order over base classes. Formally, we have for each free class C

$$o(C) \subseteq \bigcap_{(C,C_i)\in SPEC} o(C_i).$$

Example 2.2 In the running example, we have the specializations Employees SPEC Persons and Applicants SPEC Persons. In Figure 1 specializations are introduced by the keyword SPEC. □

For the EXTREM scheme, we can define a set of integrity constraints. For example, the object sets of Employees and Applicants (both subsets of Persons by subclassing) are forced to be disjoint by the additional *disjointness constraint* in the scheme definition.

Each object is assigned a tuple of attribute values called its *state*. All objects of a fixed class are required to have the same *state type*, i.e., a fixed tuple of attributes. As attribute values, we can use simple or constructed values or even objects, then the corresponding attributes are called *simple*, *constructed*, or *object-valued attributes* resp.. We can recursively construct new domains by applying set-, tuple- and list-constructors on simple and abstract domains. Additionally, we can define *user-defined Abstract Data Types* (ADTs) by encapsulating the type contructors and accessing the values by ADT-functions visible at the interface of the ADT.

Employees	*salary*	*department*	*superior*	*age*
β_1	3000	α_1	$\perp$	51
β_2	1200	α_1	β_1	60
β_3	800	α_1	β_1	33
β_4	1050	α_2	β_5	25
β_5	4000	α_2	$\perp$	55

Departments	*name*
α_1	Toys
α_2	Drugs

Figure 2: Object relations for the classes Employees and Departments

Each attribute A is assigned a unique domain $dom(A)$. An attribute with a set- or list-valued domain is called a *complex attribute*.

Example 2.3 In Figure 1, attributes are defined in the `ATTRIBUTES`-section of the EXTREM scheme definition language. For example, the attributes of the class Employees are salary (a simple attribute), department superior, which are a object-valued attributes. The attribute values for these attributes are elements of the object sets of Departments and Employees, resp.. Since each Employees is a subclass of Persons, each element of the domain of Student is assigned an age besides the explicitly mentioned attributes. □

The instances of classes, i.e. objects and their states, can equivalently be represented by nested relations with additional surrogate attributes. These nested relations are called *object relations* and are in fact functions from object sets to their states (see [HS93]).

Example 2.4 As an example, we present the object relations for the classes Employees and Departments in Figure 2. The Employees and Departments columns, resp., represent the object identities of both classes. The values of the object-valued attribute department are department objects. Hence, Departments is a component class of Employees and, vice versa, Employees is called an *owner class* of Departments. □

The behavioral component of EXTREM allows the definition of methods for each of the classes and the inheritance of methods from superclasses to subclasses. We distinguish between *query methods* (derived or computed attributes) and *update methods*, where the effect is a state change of objects in the appropriate class. In the interface of methods, attributes used in the implementation of the methods are specified in the `USES`-section of the interface. In the interface of update methods, attributes (used and) modified by the method are specified in the `MODIFIES`-section.

Example 2.5 We have defined an update method adjust_salary in class Employees which decreases the salary of an employee to within a certain reach of social welfare. □

In OSCAR, *derived classes* can be computed by object algebra expressions, queries in the O^2QL language, or programs in the rule-based language LIVING IN A LATTICE (see [HFW90, VdBH93, HS93]). For example, we can derive a subclass of employees employed in the "Toys" department by specifying the O^2QL query

```
DERIVED CLASS Employees_of_Toys_Department
SELECT OBJECT Employees
FROM          Employees
WHERE         department.name = 'Toys' .
```

On the other hand, we also can have (complex) values as a query result. The following query simply computes a set of INTEGERs (ages of employees of the "Drugs" department):

```
SELECT age
FROM   Employees
WHERE  department.name = 'Drugs' .
```

A derived class can be used in the same way as base classes. The following ECA–rules and the techniques to detect non-terminating and non-confluent rules are heavily based on this feature of OSCAR.

3 The OSCAR Trigger System

The OSCAR system offers a trigger system which should be sufficient for most applications. The basis of our triggers are the widely accepted ECA–rules, which were first introduced by HiPAC ([DBM88], [DD89]).

3.1 Syntax

The syntax for the definition of a trigger in OSCAR is as follows:

```
CREATE RULE name
AFTER Event {OR Event|AND THEN Event|XOR Event}
[IF Condition]
THEN DO [INSTEAD] Action [DIRECT|DECOUPLED] [ON INSTANCE LEVEL]
[PRECEDES RuleNameList]
[FOLLOWS RuleNameList]
```

The definition of an `Event` is:

```
  Operations TO
  "("Classexp")"{."("Attribute{, Attribute}")"}
  [WHERE Selectexp]
```

`Operations` is defined as:

```
"("Operation{; Operation}")"
Operation := "Insert" | "Update" | "Increase" | "Decrease" |
             "Delete" | "Retrieve" | Methodname
```

The other expressions are defined as:

`Classexp` is an expression with class names and the set operators $\cup$, $\cap$ and $\setminus$, defining a derived class.

`Attribute` is an attribute which is defined for the derived class.

`Selectexp` is a valid query language selection, defined on the derived class, giving another (specialized) derived class as a result. Thus the semantics of an event is fulfilled.

`Methodname` is a method call which is valid for the derived class.

`Condition` is an existentially qualified O^{2}QL query with the additional keywords `NEW` and `OLD` for referencing the old and new sets of objects before and after the triggering operations (if applicable). Alternatively, we can use the transition classes (defined below) instead of `NEW` and `OLD`.

`Action` is a list of the following expressions:

- An update expression,
- an insert expression,
- a delete expression or
- an expression of the form:

```
Methodname TO Classexp [WHERE Selectexp]
```

3.2 Semantics

The semantics of an OSCAR trigger is pretty straight–forward. The building blocks of rule processing are closed nested transactions. If the event detector signalizes an event which triggers a rule, the query which makes up the `Condition` part will be evaluated at a rule assertion point (i.e. a point in time, where a rule execution cycle is started: end of transaction or user defined). If the result of the query is not empty, the `Action` will be executed. If the keyword `DECOUPLED` is present in the definition of the rule, a new root transaction will be started for the `Action` which runs autonomously from the triggering action, i.e. if the triggering action aborts, the `Action` part of our rule will be committed (if successful). Otherwise (keyword `DIRECT`, this is the default) the `Action` will run as a subtransaction of the transaction that raised the triggering `Event`. All nonfatal error codes will then be forwarded to the parent transaction which is responsible for the treatment of the error condition. If a fatal error occurs, the complete transaction will be aborted.

An `Event` is defined by a set of operations which are performed on a derived class. This derived class is specified by

1. the `Classexp` which consists of an expression with classnames and the set operators, thus defining a dreived class.
2. Then a specialization of this derived class is constructed by the specification of a `Selectexp` which may use attributes of the derived class only. It limits the number of eligible extensions by giving a condition which all extensions have to fulfill. Therefore the `Selectexp` can be viewed as a class invariant, and yields a valid derived class.

Thus the usual semantics of an event, as we know it from other publications, is kept.

The `Condition` and `Action` can refer to transition classes or instances (in the set–oriented or instance–level case respectively) for obtaining informations about the changes which triggered the rule. For each defined rule one set of transition classes is created corresponding to the `Operations` defined in the event part (`rulename_del, rulename_ins, rulename_newupd, rulename_oldupd`). These derived classes are not real base classes. Therefore It is not allowed to define rules on them. If a rule's event is complex, i.e. consists of some basic events connected by `OR` or `AND THEN` the derived transition classes are created as a generalization of the derived classes which are defined in the basic events. The access to these generalizations is managed by an intelligent query processor.

All extensions which have been deleted, inserted and updated by some rule's actions including the effects of the last "regular" DML statement[1], are kept in these change classes. As mentioned above the domain and structure of these transition classes correspond to those of the derived classes which are defined in the event part of the rule `rulename`.

If the keyword `ON INSTANCE LEVEL` is present, the rule will be triggered once for each involved instance. Otherwise the rule will be triggered only once for the complex data manipulation operation.

If the keyword `INSTEAD` is present, the `Action` part will be executed instead of the triggering action as in POSTGRES ([Sto92], [S$^+$90]).

After each DML statement the transition classes are filled with the objects which were deleted, updated, inserted or retrieved by this DML statement if one of the `Operations` occurs for the `Attribute` (if specified) for a nonempty subset of the extensions, described by the `Classexp` and the `Selectexp` in the respective rule definition to which the transition classes belong.[2]

At each rule assertion point (i.e. at each point in time, where the rules will be evaluated and executed: end of transaction or user defined) the rules triggered rules are executed with respect to the following algorithm:

```
WHILE {not_empty_derived_rule_class}
         - {not_yet_completed_composite_event} != {} DO
  BEGIN
  Select one of the rules r with the highest priority
     where        {not_empty_derived_rule_class}
               - {not_yet_completed_composite_event} != {};
  Evaluate its Condition;
  IF Condition = TRUE
    THEN Execute it in the required fashion;
  Delete all objects out of their corresponding
                         derived_rule_classes;
  Check for completion of composite events
  END; {WHILE}
```

The detection of complex events involving `AND THEN` is done by means of Petri nets.

[1] which started the cascading execution of rules

[2] Please note, that in our model objects can exist in more than one class at a time.

Example 3.1 Consider the following rule:

```
CREATE RULE toy_subordinate_del
AFTER (DELETE) TO Employees
  WHERE department.name = "Toys"
IF EXISTS SELECT OBJECT Employees FROM Employees
  WHERE department.name = "Toys"
THEN DO DELETE FROM Employees WHERE department.name = "Toys"
  AND superior IN toy_subordinate_del_del
```

This trigger recursively deletes employees of the toy department if their chief is deleted. The rule's name is `toy_subordinate_del`, it gets triggered by a deletion, and therefore a transition class is created automatically. It's name is constructed according to the rules mentioned above: `toy_subordinate_del_del`. This transition class is then used in the `Action` part of the rule to perform the required deletions. □

4 Termination

4.1 Introduction

We'll first give a brief sketch of the idea behind our analysis algorithm. It is somewhat similar to the termination test in [AWH92]. The improvements in our algorithm consist of

- the incorporation of OO concepts,
- the use of generic operations as well as method calls as `Action`,
- a much richer rule model,
- greatly improved sufficient conditions (even in the first stage), and
- an additional analysis stage, which makes our sufficient conditions even stronger.

Also, we can equally handle set–oriented and instance–level rules. In the remainder of the article we concentrate on set–oriented rules.

Our algorithm consists of two stages. In the **first stage** we construct a triggering graph out of the syntax of our rules. If our triggering graph has no cycles, we can guarantee that our set of rules will terminate for all initial database states.

In the **second stage** we analyze all strong components of the graph separately. We group the actions together to one complex operation which modifies the same objects in the same derived class. If this complex operation satisfies certain criteria like monotonicity, we are able to remove some edges in the respective strong component, thus possibly eliminating some cycles.

The only prerequisite needed for our algorithm is the assumption that the `Action` part of a trigger will always terminate. This is not obvious because as a part of the `Action` there may be a method call. Our methods are written in a Turing–complete programming language. If there are only retrieve, insert, update and delete statements in the `Action` part of each trigger, this property is guaranteed to hold.

Our goal is to compute subsets of triggers $\{t_{i_1}, ..., t_{i_k}\}$ out of a set of triggers $T = \{t_1, ..., t_n\}$ that might give rise to non–terminating cyclic execution of the triggers' actions, or, vice versa, to verify that the triggers in T will terminate for all initial database states d_I.

4.2 Domains as Lattices

For the analysis of termination in the two stages of our algorithm we need a formal foundation for the application of fixed–point theorems. Therefore we apply the notion of lattices to our domains. This enables us to treat all domains in a uniform way.

By a *lattice* we understand a system $\Lambda = (A, \leq)$ formed by a nonempty set A and a binary relation $\leq$, which establishes a partial order in A, and that for any two elements $a, b \in A$ there is a least upper bound $a \cup b$ and a greatest lower bound $a \cap b$.

Every domain in OSCAR[3] consists of such a nonempty set. The partial order is either defined by default or can be defined by overloading the comparison operators (for not set–valued ADTs). For the ordering of all set–valued domains (i.e. Classes, some ADTs, SET and LIST[4]) we use the set inclusion relation. The least upper bound and the greatest lower bound of any two elements of one of our domains can thus be obtained straight–forwardly by applying the above definitions.

We call an operation $o : \Lambda \longmapsto \Lambda$ *increasing* (*decreasing*) if $x \leq o(x)$ $(x \geq o(x))$.[5] For example a deletion of an element of a set is an decreasing operation, because the cardinality of the set before the deletion is greater than the cardinality of the set after the deletion.

4.3 First Stage

In the first stage of our algorithm a directed *triggering graph* $G_T = (V, E)$ is constructed out of the syntax of the trigger definitions in T. Each $t_i \in V$ represents a trigger $t_i \in T$. An edge e from t_i to t_j in E denotes, that t_i might trigger t_j. Therefore, if G_T's strong components consist of isolated edges without loops, we can guarantee the termination of our trigger set.

For the analysis of when to draw an edge from t_i to t_j, we need the notion of an *event*. An event is defined as follows:

Definition 4.1 *An* atomic event e *is a 4–tuple* $e = (O, C, A, S)$ *where*

O *is a set of operations as defined in the syntax description, i.e.* `RETRIEVE, INSERT, UPDATE, INCREASE, DECREASE, DELETE, Methodname`

C *is a derived class, constructed out of the* `Classexp` *defined above,*

[3] Classes, ADTs, INTEGER, REAL, CHAR, STRING, BOOLEAN, SET, TUPLE and LIST

[4] We just neglect the ordering of a list's elements

[5] Note, that our definition differs from the usual definition of *increasing* (*decreasing*) operators in lattices!

A is a set of attributes which are valid for the derived class, defined by C,

S is an instance of the `Selectexp` *defined above.*

For each atomic event this 4–tuple can easily be computed out of its syntax. A composite event, which is constructed out of atomic events, interconnected by the `OR` and `AND THEN` operators and maybe brackets, is a set of atomic events, which can be computed by using the following rules:

1. Get the subexpression `ce1 op ce2` with the two composite events `ce1` and `ce2` and the highest precedence.
2. If `op` = `OR` or `op` = `XOR`, replace the expression by the composite event `ce` = {`ce1, ce2`}, i.e. the rule is triggered by the occurrence of either `ce1` or `ce2`.
3. If `op` = `AND THEN`, replace the expression by the composite event `ce` = {`ce2`}, i.e. if the rule is triggered by the occurrence of `ce2` after `ce1` occurred, we consider the occurrence of `ce2` only.
4. If the remaining expression consists of a single composite event `ce`, then end, else goto step 1.

In almost the same manner it is possible to compute an atomic event out of each statement of a trigger's `Action`–part straight–forwardly. The resulting composite event, which we will call *actionevent* in the sequel, describing the complex event, which is caused by a trigger's `Action`, is obtained by uniting all atomic events. The following algorithm shows the basic steps of this procedure:

Algorithm 4.1 (Action → actionevent) *This algorithm performs a transformation of each statement of an* `Action` *into an event e_i. The resulting actionevent ae then is the union of all e_i:*

```
INPUT: Action with statements si, i = 1...n;
OUTPUT: actionevent ae;
FOR EACH statement si in the Action do
  BEGIN
  IF si IN {insert, delete}
    THEN BEGIN
         compute O, C and S out of the syntax of si;
         A := {all attributes of the derived class C}
         END;
  IF si IN {update}
    then compute O (which could also possibly be increase or
           decrease), C, A and S out of the syntax of si;
  IF si IN {methodcall}
    THEN BEGIN
         IF si is an update method
           THEN IF the method is labelled as being increasing
                  (decreasing) THEN O := {retrieve, increase}
                                   (O := {retrieve, decrease})
                               ELSE O := {retrieve, update}
           ELSE O := {retrieve};
```

```
        compute C and S out of the syntax of the method call;
        compute A out of the USES and MODIFIES list of the
          method;
        compute additional (O,C,A,S) tuples if in the method
          body there are further retrieve operations;
        END
END;
```

We now use these two kinds of events to analyze, whether one rule's action can trigger the activation of another rule, i.e. we have to add an edge (t_i, t_j) to the set of edges E of G_T. We therefore have to check for each rule's (t_i) actionevent, whether it has something in common with the events of all rules, including t_i itself.

A rule's actionevent ae might trigger another rule t_j if none of the following properties hold for each combination of atomic events of ae (e_1) and t_j (e_2):

1. The set of operations, that e_1 performs is disjoint to the set of operations, to which e_2 responds, or e_1 includes an `increase` or `decrease` operation and e_2 responds to arbitrary updates.

2. The least common upper bound in the class lattice of the derived class, on which e_1 performs its action and the derived class which is defined in e_2 is Object, i.e. the most general class in the class hierarchy [6]. In this case, the two events are defined on disjoint derived classes.

3. The two events are defined on disjoint sets of attributes.

4. The set of objects defined by the `SELECT OBJECT` expression of e and the set of objects defined by the `SELECT OBJECT` expression of e_2 is disjoint for all database states.[7]

If all four properties at once hold for at least one arbitrary combination of atomic events e_1 of ae and e_2 of t_j, we have to include the edge (t_i, t_j) into E, thus marking that t_i might trigger t_j.

As stated above, we have to go through these steps for all pairs of rules, which can be constructed out of T. If the strong components of the resulting triggering graph G_T consist of isolated edges without loops only, T is guaranteed to terminate for all initial database states. If this condition does not hold, we now have to launch stage two on G_T, which analyzes all strong components separately, in order to find some edges that can be removed.

We summarize the notions defined above with a little example:

Example 4.1 We introduce two triggers. The first trigger fires if an old employee (age ≥ 40) is inserted into the Employees class. It will then delete the oldest employees so that our company does not have too many old employees. The second trigger fires if a young employee (age < 40) is deleted. It will then insert the youngest employee from the class Applicants into the Employees class.

The two triggers for this purpose are defined as follows:

[6] In the running example, Persons and Departments are automatically disjoint specializations of Object.

[7] This can (for some cases) be verified by testing predicates like in [BJNS94]

```
t1:
AFTER (INSERT) TO Employees WHERE age >= 40
THEN DO DELETE FROM Employees WHERE (age >= 40) AND (age >= ALL
  (SELECT age FROM Employees))

t2:
AFTER (DELETE) TO Employees WHERE age < 40
THEN DO
  INSERT INTO Employees OBJECTS
  SELECT OBJECT Applicants FROM Applicants WHERE age <= ALL
    (SELECT age FROM Applicants);
  DELETE FROM Applicants OBJECTS
  SELECT OBJECT Applicants FROM Applicants WHERE age <= ALL
    (SELECT age FROM Applicants)
```

In the corresponding triggering graph G_T we have edges
$E = \{(e_1, ae_1), (e_2, ae_2), (ae_2, e_1)\}$.
E does *not* contain (ae_1, e_2) because a predicate testing algorithm can verify easily that the set of objects which is touched in the action of t_1 is disjoint to the set of objects to which deletions t_2 reacts for all possible extensions of the class Employees.

Thus G_T does not contain any cycle. Therefore $T = \{t_1, t_2\}$ is guaranteed to terminate for all initial database states d_I. □

4.4 Second Stage

Because of the limited space available, we can just give a brief overview of the main concepts, we are using for the second stage of our algorithm.

We now analyze each strong component of G_T, which does not consist of an isolated edge without loops, separately. It is possible to remove an edge from a strong component of G_T, if one rule some time performs an "empty" operation after a finite number of steps. This is for example the case, if we have a delete operation on a set–valued domain. If there is no other operation in the respective strong component, which inserts a value into the set, repetitive deletion of elements from this set will, after a finite number of steps, yield an empty set. Any further delete operations on this set will therefore be operations, that perform nothing on the database. Thus, the cycle will be interrupted, and we can remove the respective edge from this rule to the next one which is triggered by the respective deletion of objects. We extend this notion to arbitrary types by the extensive use of lattice theory.

If we can't fulfill the conditions which are shown below, the respective edges won't be considered for removal.

We model this behavior by means of lattices. Each possible domain in OSCAR can be interpreted as a lattice.[8]

Lemma 4.1 The repetitive application of an increasing (decreasing) operation on a lattice will terminate after a finite number of steps if:

[8] For ADTs we have to overload the comparison operators. If this is not done, the respective edge will not be considered for removal from G_T.

1. the step size of the operation is non–decreasing (non–increasing)[9],
2. there is an upper (lower) bound for the operation (possibly the "0" and "1" elements of the lattice, if it is complete).

Proof (Sketch):
The termination itself follows out of the application of the contraction theorem. The property that the step size must not be decreasing is used for proving the termination after a finite number of steps for lattices with an infinite number of elements. □

The property, whether an operation is increasing or decreasing can be acquired by analyzing the transitional conditions of methods and applying predicate testing algorithms to update operations. For the deletion and insertion of values from and into set–valued attributes or classes, this property holds obviously: A delete is a decreasing operation, because the cardinality of the set becomes smaller, an insertion is an increasing operation, because the cardinality is increased. The lower bound for operations on sets is the empty set, the step size in a set is at least the increasing or decreasing of the cardinality by one. Therefore these properties hold automatically for most operations on sets.

The upper or lower bounds for the operations can be formulated in either of three ways:

- as integrity constraints on (derived) classes,
- as integrity constraints on methods,
- in the definition of the derived class in the `Selectexp`.

These conditions, if met, won't lead to an transaction abort. Instead the transaction won't be started. Therefore, even in the `DIRECT` mode, a complete rollback of all rule actions will never take place, thus ensuring the wanted semantics.

In order to verify these properties for a selected strong component we group together all operations of this component, that act on the same attributes of the same extensions, into a composite recursive operation. If the above stated properties hold for one of the composite operations, we can remove the edge from G_T which connects the rule with the some time "empty" operation with the next rule (which is triggered by this operation), thus reducing our strong component, which will eventually split into several strong components with isolated edges without loops. In this way it might be possible to eliminate complete cycles, which were created in the first stage of our algorithm. If we can't verify all of the above properties, the strong component remains untouched and is presented to the programmer "as is" for further considerations or remodelling.

Example 4.2 We now consider the rule defined in example 3.1. For this example the first stage of our algorithm would yield a triggering graph consisting of an isolated node (representing the single trigger) with a loop. In the second stage this loop will be removed because in this strong component there

[9]This condition can be relaxed by using research results for real and integer valued functions

are only delete operations and the rule is triggered by this delete operation. In the "worst" case, the cascading triggering will stop when all objects hve been deleted from the class Employees. After the removal of this edge, we have a single node without loops left. Therefore our algorithm can guarantee the termination of our set of rules for all initial database states.

This special case may not be new, but in our approach it is smoothly integrated into a much broader theory of eliminating edges with the help of lattice theory. □

A more complex situation is shown in Example 4.3.

Example 4.3 We now consider the following rule:

```
CREATE RULE decrease_salary
AFTER (DECREASE) TO Employees.salary
                 WHERE decrease_salary_newupd.salary > 1000
DO decrease_salary_newupd.adjust_salary(100.0) DECOUPLED
```

This rule lowers all salaries of employees, who receive a cut of salary, until it is just below \$1000. For removing the concerned edge in G_T the second stage has to verify the following conditions:

- The method adjust_salary() must decrease the salary. This condition is met by a transitional constraint on the method (see below).
- The step size must be non–decreasing. This property is met because we have a constant parameter (100.0) which is used in the transitional constraint

  (decrease_salary_oldupd.salary =
  decrease_salary_newupd.salary - amount_of_decrease).

  Additionally, we need the precondition amount_of_decrease $>$ 0.
- There must be a lower bound for our operation. This condition is met because of the simple (i.e. automatically verifiable) condition in the WHERE–clause. The lower bound could have also been specified as an integrity constraint on the class Employees or as a post condition of the method adjust_salary().

Therefore the second stage would have eliminated the respective edge, thus guaranteeing the absence of non–terminating cyclic execution of our rule. If we had not had the precondition and the transitional constraint on the method, the second stage would not have touched the respective edge in G_T. □

5 Related Work

In the last years there have been a lot of papers about many aspects of ECA–rules starting with their introduction by HiPAC ([DBM88], [DBB+88], [DD89]). The ECA–rules have since then become a kind of standard principle for most active DBMS. Most of the articles about ECA-rules in DBMS are concerned with the general architecture, mechanisms for expressing and detecting events, conditions and actions, their representation within the DBMS, the embedding of triggers in transactions and their applications. Widely known approaches for the general architecture of active DBMS include SAMOS ([GD92], [GD93]) with a very powerful event specification language and refined facilities for the detection of composite events using Petri nets ([GD94]), the event specification language Snoop ([CM93]) with very powerful mechanisms for the expression of complex events and REACH ([BBKZ92]) with the stress on the temporal aspects and performance.

As mentioned above, the second wave of publication about active DBMS is mainly concerned with the applications of ECA-rules for various purposes. E.g. quite a few articles deal with using triggers for the enforcement of constraints. Among them are Starburst ([CW90]), an extension to the latter approach by Ceri et al. ([CFPT92]) and an approach by Lipeck, Gertz et al. which also incorporates temporal integrity constraints ([Ger94]). Other applications for active DBMS include the maintenance of derived data and materialized views.

All these approaches have the problems of termination and confluence in common, and yet there are only very few proposals for solutions. Even in commercial systems which incorporate active features like Ingres and Oracle, these problems are only addressed in a very restrictive way, if at all. These systems deal with termination by enforcing an upper bound for the cascaded triggering of rules during runtime. If this upper bound is met, the whole transaction will be rolled back even if in the next cycle the cascading triggering would terminate. In Oracle one cannnot even set this upper bound. It is hard–wired to 64. Confluence is not addressed at all, it is the user's responsibility to foresee all possible problems.

The only articles dealing with the problem of termination in active DBMS are the following:

- In [KU94] the authors use a different paradigm than in our approach: their approach is based on term-rewriting systems which is hard to compare to the ECA-approach we and others use.

- In [CFPT92] the authors deal with the problem in the limited focus of constraint enforcement. They only mention some heuristic techniques.

- In [VS93] the authors give sufficient conditions to analyze termination of CA–triggers at definition time. Unfortunately they only use a very limited formalism for the definition of triggers. Their triggers consist of a simple query, which, if evaluated to true, can trigger a simple and single attribute assignment. Thus, our rule definition language is much more powerful than theirs (for which termination and confluence are even decidable!) and their approach cannot be easily transferred to ECA–rules.

- [AWH92] deals with the problem of termination in the context of relational databases. In our object–oriented approach we benefit from the richer semantics in two ways:

    1. We give a more powerful language for the expression of triggers and

    2. we can use the richer semantics of the OO data model for a more accurate analysis.

The first stage of our algorithm resembles the one proposed by Widom et al., but even there we benefit from the use of the `INCREASE` and `DECREASE` constructs, which already makes our first stage more powerful than their algorithm. Additionally, we benefit from the more powerful object-oriented concepts like subclassing. For example, it is not possible in Starburst to express and therefore analyze subsets of tuples of relations in their events. Thus, they cannot automatically detect that the rules defined in example 4.1 are guaranteed to terminate after a finite number of steps.

  In the second stage we perform an even deeper analysis, thus making our conditions much weaker. E.g. we can detect in the examples 4.2 and 4.3 that there is no non–terminating cyclic execution. The algorithm in [AWH92] would detect a possible cyclic execution.

- In [BW94], the authors give better conditions for the analysis of termination than in [AWH92] but they also reach only the power of our first stage. Furthermore, they only deal with CA-rules in a relational context.

It is our idea to start out with a comparatively simple language for the expression of rules, for which we can give very strong sufficient conditions. After this step is taken, we can simulate most of the more powerful atomic events and event constructors one can find in other approaches. These can then be made available to the user by the definition of macros which are internally broken down into our basic events and event constructors for the analysis of termination. For example we can simulate

- an absolute time event by defining an event on the system class `time`,
- a periodic time event (e.g. every 30 minutes) by an event like
  `AFTER (UPDATE) TO system_clock.minutes`
  `WHERE minutes mod 30 = 0`,
- e.g. `ANY 2 (E1,E2,E3)` by `(E1 AND THEN E2) OR (E2 AND THEN E1)`
  `OR ... OR (E3 AND THEN E2)`[10],
- other complex event constructors by sequences of triggers which record the needed prerequisites for the signaling of the complex event (e.g. arbitrary intermediate states) in some specially defined classes. This can easily be hidden from the user.

For the detection of complex events at runtime one can of course choose a strategy with a better performance. Nevertheless it is thus possible to verify the property of termination without the need to extend our algorithm on the one hand, or it is possible to extend our algorithm to support other atomic events or event constructors, whichever strategy seems more feasible and adequate.

6 Conclusions and Future Work

We described a set of algorithms that allows the efficient analysis of termination of a set of ECA–rules at definition time. Our algorithm cannot find the exact

[10] This is a generalization of the `AND` operator!

answer for all cases, but it gives some strong sufficient conditions which can isolate all subsets of rules that em might give rise to non–terminating execution. Our conditions are meant to be incorporated into an interactive design tool for the definition of rules which shows the rule designer all subsets of rules which may lead to non–terminating cycles of rule executions. On the other hand we can restrict the number of rules which have to be watched for cycles during runtime.

By using OODB concepts on a large scale we have the opportunity to take advantage of the greater expressiveness of object–oriented databases, e.g. we can use the class lattice for the verification of empty intersections between derived classes which is a decidable problem. Another advantage of our approach is the uniform treatment of generic update operations as well as method calls on the one hand, and of all data types, including abstract data types on the other hand, thus incorporating the object–oriented concepts which our data model defines. The data model is a real object–oriented model with all the required features. On the other hand it is more of an evolutionary approach from the relational world. We don't give up all the "goodies" like set–orientedness, generic updates and associative query languages, which made the relational model as successful as it is nowadays. Therefore our approach can easily be remodeled for the use in relational databases as well as object–oriented databases.

We are currently investigating sufficient conditions for the property of confluence. Also for this aspect one can expect improvements over existing results for active relational databases.

The next step of our approach which is under development, will be a tool that analyzes conflicting triggers during runtime by the use of Petri nets. Only conflicting rules have to be considered. Furthermore more complex composite events can easily be modeled by means of Petri nets.

Other specifics like event inheritance and others play a vital role in other fields like e.g. event detection but fall out of the scope of this paper. These aspects will be treated in a soon–to–appear PHD–thesis by the first author of this article.

Finally, there are quite a number of improvements we are planning for our algorithms, namely:

- extending the `Actions` by procedural constructs like `IF - THEN - ELSE` and loops,
- analyzing the property of confluence ([AWH92]),
- using incremental methods for the construction of G_T,
- using OO concepts like specialization for complex events thus making the detection of events easier,
- developing procedures for the run–time observation of possibly hazardous rules,
- representing our triggers at the meta level of OSCAR,
- implementation of our algorithms to analyze their behavior for large sets of triggers.

Acknowledgements

We would like to thank Holger Riedel for many fruitful discussions and Reinhold Schönefeld for a lot of very useful hints.

References

[AFS89] S. Abiteboul, P.C. Fischer, and H.-J. Schek, editors. *Nested Relations and Complex Objects in Databases*, volume 361 of *Lecture Notes in Computer Science*. Springer, Berlin, 1989.

[AWH92] A. Aiken, J. Widom, and J.M. Hellerstein. Behavior of database production rules: Termination, confluence and observable determinism. In *Proc. ACM-SIGMOD, San Diego*, pages 59–68. ACM, May 1992.

[BBKZ92] A.P. Buchmann, H. Branding, T. Kudrass, and J. Zimmermann. Reach: A real–time active and heterogeneous mediator system. *IEEE Bulletin of the TC on Data Engineering*, 15(1–4):44–47, December 1992.

[BJNS94] M. Buchheit, M.A. Jeusfeld, W. Nutt, and M. Staudt. Subsumption between queries to object-oriented databases. *Information Systems*, 19(1):33–54, 1994.

[BW94] E. Baralis and J. Widom. An algebraic approach to rule analysis in expert database systems. In *Proc. 20th Int. Conf. on Very Large Data Bases, Santiago de Chile*, September 1994.

[CFPT92] S. Ceri, P. Fraternali, S. Paraboschi, and L. Tanca. Constraint enforcement through production rules: Putting active databases to work. *IEEE Bulletin of the TC on Data Engineering*, 15(1–4):10–14, December 1992.

[CM93] S. Chakravarthy and Deepak Mishra. Snoop: An expressive event specification language for active databases. Technical Report UF-CIS-TR-93-007, University of Florida, Gainesville, FL 32611, March 1993.

[CW90] S. Ceri and J. Widom. Deriving production rules for constraint maintenance. In *Proc. 16. Int. Conf. on Very Large Databases*, pages 566–577, August 1990.

[DBB+88] U. Dayal, A. Buchmann B. Blaustein, et al. The HiPAC Project: Combining Active Databases and Timing Constraints. *ACM SIGMOD Record*, 17(1):???, 1988.

[DBM88] U. Dayal, A. Buchmann, and D. McCarthy. Rules are objects too: A knowledge model for an active object–oriented database system. In *Proc. Int. Workshop on Object–Oriented Database Systems*, 1988.

[DD89] U. Dayal and McCarthy D. The architecture of an active database management system. In *Proc. ACM–SIGMOD*, pages 215–224, 1989.

[DNP91] O. Diaz and P. Gray N. Paton. Rule management in object–oriented databases: A uniform approach. In *Proc. 17th International Conf. on Very Large Databases, Barcelona, Spain*, September 1991.

[GD92] S. Gatziu and K.R. Dittrich. Samos: an active object–oriented database system. *IEEE Bulletin of the TC on Data Engineering*, 15(1–4):23–26, December 1992.

[GD93] S. Gatziu and K.R. Dittrich. Eine Ereignissprache f"ur das aktive, objektorientierte Datenbanksystem SAMOS. In *Proc. BTW '93*, 1993.

[GD94] S. Gatziu and K.R. Dittrich. Detecting composite events in active database systems using petri nets. In *Proc. 4th Int. Workshop on Research Issues in Data Eng.: Active DBS*, 1994.

[Ger94] M. Gertz. Specifying reactive integrity control for active databases. In *Proc. 4th Int. Workshop on Research Issues in Data Eng.: Active DBS*, 1994.

[GGD91] S. Gatziu, A. Geppert, and K.R. Dittrich. Integrating active concepts into an object–oriented database system. In *Proc. DBPL–3 workshop*, August 1991.

[GJ92] N.H. Gehani and H.V. Jagadish. Active database facilities in ODE. *IEEE Bulletin of the TC on Data Engineering*, 15(1–4):19–22, December 1992.

[Heu89] A. Heuer. A data model for complex objects based on a semantic database model and nested relations. In *[AFS89]*, pages 297–312, 1989.

[HFW90] A. Heuer, J. Fuchs, and U. Wiebking. OSCAR: An object-oriented database system with a nested relational kernel. In *Proc. of the 9th Int. Conf. on Entity-Relationship Approach, Lausanne*, pages 95–110. Elsevier, October 1990.

[HH91] C. Hörner and A. Heuer. EXTREM — The structural part of an object–oriented database model. Informatik-Bericht 91/5, Institut für Informatik, TU Clausthal, 1991.

[HS93] A. Heuer and P. Sander. The LIVING IN A LATTICE rule language. *Data and Knowledge Engineering*, 9(3):249–286, 1993.

[KU94] Anton P. Karadimce and S. D. Urban. Conditional term rewriting as a formal basis for analysis of active database rules. In *Proc. RIDE '94 4th Int. Workshop on Research Issues in Data Engineering, Houston, Texas, USA*, pages 156–162, February 1994.

[S+90] M. Stonebraker et al. On rules, procedures, cashing and views in database systems. In *Proc. ACM SIGMOD Conference on Management of Data*, pages 281–290. ACM New York, 1990.

[Sto92] M. Stonebraker. The integration of rule systems and database systems. *IEEE Transactions on Knowledge and Data Engineering*, 4(5):415–423, October 1992.

[VdBH93] J. Van den Bussche and A. Heuer. Using SQL with object-oriented databases. *Information Systems*, 18(7):461–487, 1993.

[VS93] Leonie van der Voort and A. Siebes. Termination and confluence of rule execution. *Proceedings of the 2nd Int. Conf. on Information and Knowledge Management*, November 1993.

[Wid92] J. Widom. The Starburst rule system: Language design, implementation, and applications. *IEEE Bulletin of the TC on Data Engineering*, 15(1–4):15–18, December 1992.