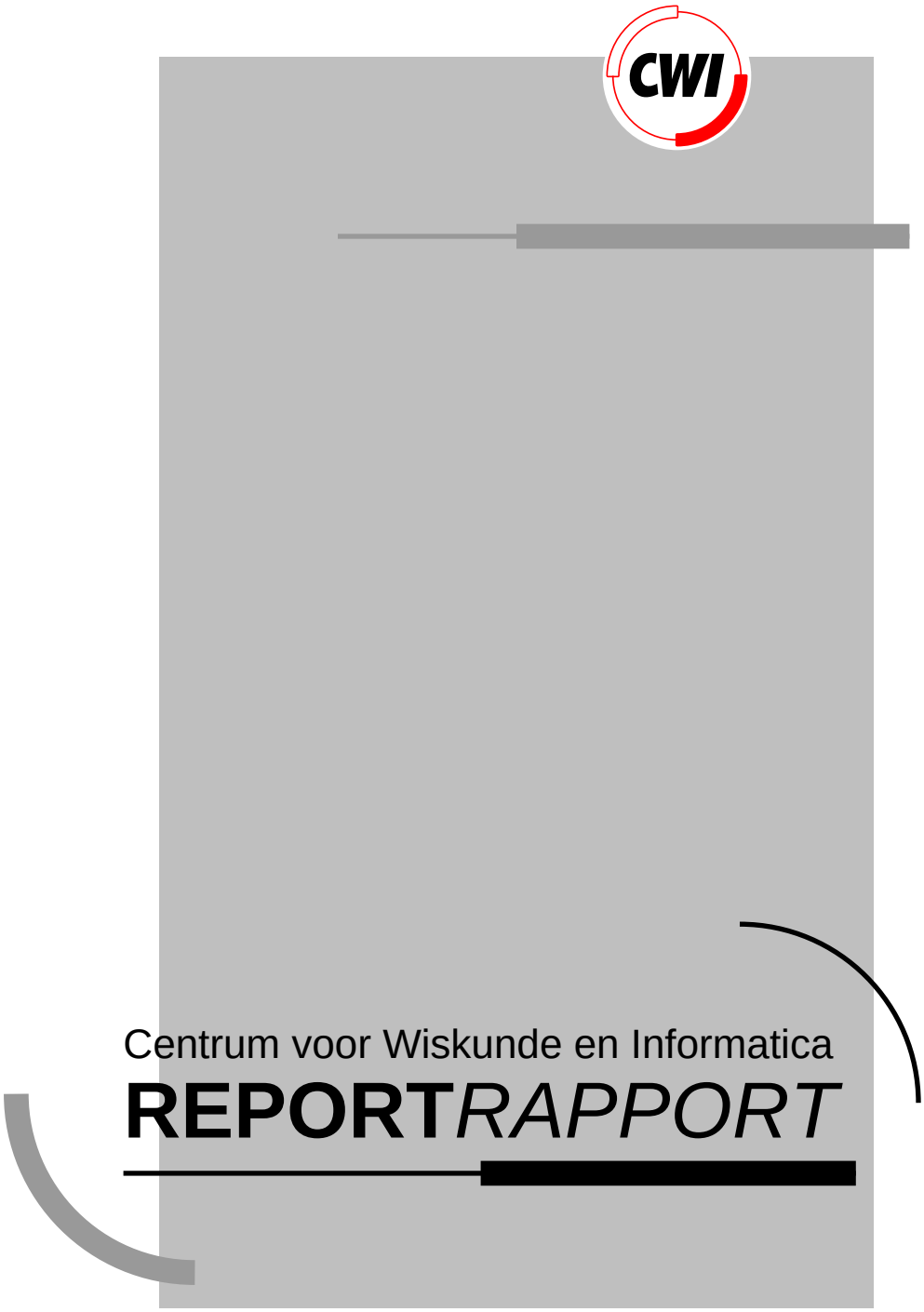

Centrum voor Wiskunde en Informatica

REPORT*RAPPORT*

Termination and confluence of rule execution

M.H. van der Voort, A.P.J.M. Siebes

Computer Science/Department of Algorithmics and Architecture

CS-R9309 1993

Termination and Confluence of Rule Execution

Leonie van der Voort, Arno Siebes
CWI
P.O. Box 4079, 1009 AB Amsterdam, The Netherlands
{leonie,arno}@cwi.nl

Abstract

Rules provide the functionality for constraint enforcement and view maintenance. A provably correct implementation of both issues based on rules, requires confluent and terminating behaviour of the rule set. However, little work has been done so far on the static detection of these properties. In this paper, a *design theory* for rule sets in an OODBMS is developed.

This paper introduces two predicates, viz., *Terminate(n)* and *Independent*, which imply respectively termination and confluency. Both predicates are characterised under two kinds of rule execution semantics: set and instance based. The decidability of the predicates is proven and it is shown that set and instance based semantics coincide whenever the rule set is independent and terminates. Moreover, sufficient conditions of low algorithmic complexity for both Terminate(n) and Independent under both kinds of semantics are given.

1991 CR Categories: H.2.1[Information Systems]: Logical Design- *data models*; H.2.3[Information Systems]: Languages- *data description languages(DDL); data manipulation languages(DML)*;

Additional Keywords and Phrases: Rule design theory, Triggers, Active Databases

1 Introduction

A DBMS becomes active through the addition of rules or triggers. Rules allow specification of data manipulation operations that are executed automatically when certain conditions are met. They offer a flexible, unifying mechanism for common database management tasks, like constraint enforcement and view maintenance. As a consequence, a number of proposals for incorporating rules into DBMS's appeared recently [16, 4, 6, 3, 9, 5, 2, 11, 12].

The correct implementation of constraint enforcement and view maintenance based on rules requires that the set is confluent and terminates [15]. A rule set terminates if its execution terminates on all database states. A terminating rule set is confluent if for each initial database state db_0 the order of rule execution does not influence the final database state db_1. That is, db_1 is uniquely determined by db_0 and the rule set. Confluency of rule sets is thus similar to confluency of rewrite systems [7] in that the execution order is immaterial. The difference, however, is that a rule sets behaviour is affected by the underlying database state, and a rule set is confluent if it is confluent on all database states.

Whether a rule set is confluent or terminates depends, of course, on the rule execution semantics. There are two pre-dominant models in the literature: set and instance based semantics. Set based semantics means processing all qualifying objects at a time, while instance based semantics means processing one qualifying object at a time. The choice between either one is mainly based on the computational model of the underlying DBMS and on the functionality of anticipated applications.

Rule interaction can be quite intricate. For example, rules may mutually activate or deactivate each other. This complicates static detection of properties such as termination and confluency. It is therefore necessary to develop a design theory which simplifies their analysis at design time. The relevance of such a theory is also endorsed by [1, 2, 5, 8, 13]. However, little work has been done so far on their development.

This paper introduces a design theory for rule sets in the context of of an object oriented data model. It is focussed on two predicates, i.e. *Independent* and *Terminate(n)*, under both pre-dominant execution semantics.

Rule sets are *Independent*, if their members are pairwise independent. Two rules R_1 and R_2 are independent if the sequential execution of R_1 after R_2 results in the same database state as the execution of R_2 after R_1. In other words, the executions of R_1 and R_2 commute. Terminating independent rule sets are confluent.

For example, consider a database populated by *cells* together with rules *paint_red* and *paint_blue*. Rule *paint_red* selects red cells from the database and paints them orange and rule *paint_blue* selects blue cells from the database and paints them green. Initially *paint_red* and *paint_blue* select different cells. Furthermore, painting a cell by *paint_red* never causes a cell to be repainted by *paint_blue* and vice versa. Thus, for each database the execution of *paint_red* followed by *paint_blue* results in the same database as when the execution order was the other way around and consequently they are independent.

Termination of rule execution is enforced by *Terminate(n)* to express that rule execution terminates in n steps. The steps are related how often a rule executes (set semantics) or how often an object is subject to rule execution (instance semantics). For example, the execution of *paint_red* and *paint_blue* terminates in one step under both set and instance based semantics because each selected cell is painted with a color such that it is never again selected by either *paint_red* or *paint_blue*.

The prime result of this paper is that recognition of both Independent and Terminate(n) is decidable under both rule execution semantics. Moreover, it is shown that for terminating, confluent rule sets set and instance based semantics coincide. Finally, as the algorithmic complexity of our decidability proof is unpractically high, sufficient conditions of low complexity are provided for both predicates.

The decidability proofs are based on *typical database states* (*tdb*). A tdb is a database state such that whenever a condition holds after rule execution in *tdb*, it holds in every database state after rule execution.

Whereas the decidability proofs are of a model-theoretic nature, our sufficient conditions are proof-theoretic. The sufficient condition for independence of two rules is based on mutual respect of each others select set and updates. That is, whenever the updates of a rule R_1 do not affect the select set and the updates of a rule R_2 and vice versa, R_1 and R_2 are independent. For example, reconsider the rules *paint_red* and *paint_blue*. The execution of *paint_red* never removes a cell from or adds a cell to the select set of *paint_blue* and vice versa. Furthermore, they always have a disjoint select set. Thus they are independent.

The sufficient condition for Terminate(n) is based on the notion of a flag. Intuitively, objects with a raised flag are selected by a rule's query. The execution of the rule's action gradually lowers the flag. Whenever a flag is lowered in less than n action executions, Terminate(n) holds. For a set of rules, terminate(n) holds if it holds for all rules and the flags are all private. Rule R has a private flag, if only rule R can (un)set the flag. For example, the red color of a cell is the flag of *paint_red*. As this color is changed by one action execution *paint(new_color=orange)*, Terminate(1) holds for *paint_red*.

Related work

Preliminary work on a design theory for rules have been published recently [11, 17, 15, 10]. In the context of the RDL rule system, Simon and deMaindreville [11] formulate a condition under which set and instance based rule execution coincide. Zhou and Hsu [17] describe a trigger (= rule) definition language and give it semantics such that conflicts in trigger execution results in an execution-abort. Under these semantics, a condition for confluency of trigger behaviour is formulated. Aiken, Widom, and Hellerstein [15] discuss the confluence and termination of rules in the context of the STARBURST rule system, which has set based rule execution semantics. For both properties they formulate sufficient conditions based on the static analysis of read and write sets of rules.

Our design theory differs in several ways from the work mentioned above. The prime difference is that we proof the static detection of two important predicates to be decidable.

Furthermore, the difference between set and instance based semantics as described in [11] is taken into account in our design theory as well. But besides establishing a condition under which both coincide, we also consider termination and confluency. Our theory aims at the static detection of these properties to

guarantee well-behaved execution, which in our opinion excludes execution-aborts. In this respect we differ from [17].

Our work is closest to that of Aiken, Widom, and Hellerstein. Similar, we discuss confluence and termination, but we deal with both execution semantics. Moreover, for the rules as defined in Section 2, we refine detection of these properties in two ways. Not only do we prove that our, weaker, predicates are decidable. Our resulting sufficient conditions for termination and confluency are also sharper.

Outline

Section 2 introduces a simple object oriented data model together with our rules. The definition of Independent and Terminate(n) together with a condition under which both semantics coincide are given in Section 3. Section 4 discusses the decidability of the predicates and section 5 describes the sufficient conditions. Finally, in section 6 we conclude and discuss future work.

2 Data and rule model

This section provides a brief overview of the simple object oriented data model and rule language used in this paper through some examples; for a full report see [14]. The data model is not meant to be yet another object oriented data model, it is just a reflection of the common concepts in this area.

Data model

A **Class** definition describes the properties and behaviour of its objects through **Attributes** and **Methods**. The attributes consist of a name and a type. Each class induces a simple type and these can be combined inductively with tuple and set constructors to obtain complex types. The methods are of a simple nature; they assign new values to attributes of an object. An example is the class *cell* defined by:

Class cell
Attributes
no: Integer
color: String
neighbors: (left: cell, right: cell)
Methods
paint_cell(new_color: String) = **self except** color := new_color
Endclass

As usual, a hierarchy is as usual a set of classes, defined directly or through (multiple) inheritance. A database state is a set of objects. Each object belongs to one or more classes. In this paper, we assume some fixed hierarchy. The universe of its database states is denoted by DB, with typical elements $db, db_0, db_1, db_2, \cdots$.

Let m be a method from a class C with header $m(l_1 : \tau_1, \cdots l_n : \tau_n)$ A method-call for m is an expression of the form: $m(l_1 = e_1, \cdots l_n = e_n)$ with each $e_i \in F\_expr$ and $e_i : C \rightarrow \tau_i$. The execution of a method-call m by an object o is denoted by $o(m)$.

Queries are formulated through the definition of query classes. A query class is a class, derived from a superclass using a selection condition. This condition identifies the objects from the superclass that are member of the query class.

An example of a query is *neighbor*, which selects cells with equal colored neighbors.

Qclass neighbor **isa** cell
Where
color = color $\circ$ neighbors.left $\wedge$ color = color $\circ$ neighbors.right
Endqclass

The selection condition is defined using a simple functional language, generated with the following grammar:

$$Expr = basic\_expr \mid Expr \circ Expr \mid (l_1 = Expr, \cdots, l_n = Expr) \mid$$
$$Expr.l \mid \{Expr, \cdots, Expr\} \mid Expr \cap Expr \mid Expr \cup Expr$$

Basic expressions are the (polymorphic) identity function *id*, the class attributes, and constants. The constants are objects in basic classes such as *Int* and *String*. The $\circ$ denotes function-composition, i.e., $e_1 \circ e_2(x) = e_1(e_2(x))$. This slightly unconventional notation is chosen because it simplifies the reasoning later in this paper. $e.l$ denotes the projection of a tuple-type expression e on it's l component. And finally, $\{e_1, \cdots, e_n\}$ denotes a set with elements $e_1, \cdots, e_n$. Both $\cap$ and $\cup$ have their usual set-theoretic interpretation

F_expr is the set of well-typed expressions generated with *Expr*, with the obvious typing rules [14]. Some examples are: *color*, *neighbors.left*, *(left_color = color* $\circ$ *neighbors.left, right_color = color* $\circ$ *neighbors.right)*, and $\{color \circ neighbors.left\} \cup \{color \circ neighbors.right\}$.

The selection condition of a query Q, denoted by C_Q, is of the form $\bigwedge_i \bigvee_j (e\,\omega f)_{ij}$ with $\omega \in \{=, \neq, \subset, \not\subset, \in, \notin\}$, $e, f \in F\_expr$, and $(e\,\omega f)_{ij}$ well-typed. Furthermore, $C_Q(o, db)$ means that the condition C holds for object o in database state *db*. The select set, $S_{db}(Q)$ of the query Q in database state *db* is defined by $\{o \in db | C_Q(o, db)\}$.

For the decidability results, we need the notions of the length of a condition, and the length of a method-call. These are defined using the length of a functional expression, which is the depth of its parse tree. For example, $length(color) = 1$ and $length(neighbors.left) = 2$. Then:

1. $length(\bigwedge_i \bigvee_j (e\,\omega f)_{ij}) = max(\{max(length(e), length(f))\}_{ij})$
2. $length(m(l_1 = e_1, \cdots l_n = e_n)) = max(\{length(e_1), \cdots, length(e_n)\})$

Rule model

A rule is a (query, method_call) pair where the query selects the objects that have to execute the method-call. The method has, of course, to be one of the methods of the class underlying the query-class. The syntax for rule definition is:

Rule rule_name = (query, method_call)

An example of a rule is the familiar **Rule** *paint_red = (red,paint_cell(new_color=orange))* where the query *red* is defined by **Qclass** *red* **isa** cell **Where** color = red **Enqclass**.

The execution of a rule $R = (Q, M)$ in a database *db* is represented by either $execute(R, db, set)$ or $execute(R, db, instance)$ depending under what semantics R is executed. Under set semantics the execution of R results in the execution of M by all objects that satisfy the selection condition of R in database *db*. With the provision that first the select set is determined, then for each object that satisfies the query the method-header is evaluated, and then the objects execute the method_call. This provision guarantees that the resulting state does not depend on a particular order. So, it is defined by:

execute(R,db,set) = {
  forall $o \in S_{db}(Q)$ **do**
    evaluate$(o(M))$ **od**
  forall $o \in S_{db}(Q)$ **do**
    $db := db \setminus \{o\} \cup \{o(M)\}$ **od**
  return db }

Under instance semantics the execution of R results the execution of M by one randomly chosen object that satisfies the selection condition of R in database *db*. This is defined by:

execute(R,db,instance) = {
  if $S_{db}(Q) \neq \emptyset$ **then**
    $o := choose(S_{db}(Q))$
    $db := db \setminus \{o\} \cup \{o(M)\}$
  fi
  return db }

We will sometimes use the, auxiliary, notation *execute(R(o),db,instance)*. It denotes the execution of rule R with subject o under instance semantics. If o is in the select set of R in db, it results in the execution of M by o otherwise it is a no-operation.

Rules are meant to respond automatical to interesting database states. As we only consider rule sets in isolation, i.e. there are no other queries or transactions, this behaviour can be represented by a repeating execution cycle, which executes a random selected, activated rule. The cycle stops when all rules have empty select sets. The execution, denoted by *E(Rules,db,sem)*, of a rule set *Rules* on a database db under semantics *sem* is thus defined by:

E(Rules,db,sem) = {
  while $\exists R \in Rules : S_{db}(Q_R) \neq \emptyset$ **do**
    $R := choose(\{R | R \in Rules \wedge S_{db}(Q_R) \neq \emptyset\})$
    $db := execute(R, db, sem)$
  od
  return db }

Given an initial database db, a set of rules *Rules* and a semantics *sem*, *E(Rules,db,sem)* induces the set *Seq(Rules,db,sem)* of *execution sequences*. Under set semantics, such an execution sequence registers which rule was chosen by the *choose* command. That is, an execution sequence is a, possibly infinite, list of rules, $Sq = [R_5, R_1, R_3, \cdots]$. Sq_i denotes the rule on the i-th position in the execution sequence.

Under instance semantics, an execution sequence not only registers which rule was chosen, but also which object was chosen to execute the method. So, in this case, an execution sequence is a, possibly infinite, list of the form $Sq = [R_5(o_1), R_1(o_4), \cdots]$.

The predicate *finite(Sq)* returns true if Sq is a finite list. So, if for $Sq \in$ *Seq(Rules,db,sem)* *finite(Sq)* holds, the execution of the rule set on the database terminates in a stable database state. This final state will be denoted by *Ex(Sq,db,sem)*.

3 Definition of the predicates

As mentioned in the introduction, confluence and termination are important properties for rule sets that are used for constraint enforcement or view maintenance. These properties are defined formally as follows:

Definition 1: Let *Rules* be a rule set and let *sem* denote either set or instance based semantics:

1) *Terminate(Rules, sem)* $\stackrel{\text{def}}{=}$ $\forall db \in DB \, \forall Sq \in Seq(Rules, db, sem) : finite(Sq)$
2) *Confluent(Rules, sem)* $\stackrel{\text{def}}{=}$ *Terminate(Rules, sem)* $\wedge$
   $\forall db \in DB \, \forall Sq_1, Sq_2 \in Seq(Rules, db, sem) : Ex(Sq_1, db, sem) = Ex(Sq_2, db, sem)$

Terminate(n)

In this paper, we restrict our attention to termination in n steps. The most obvious definition of which would be to restrict the length of all execution sequences to n. Under instance semantics, however, this implies that no *non-trivial* rule terminates in n steps, as n puts a limit on the size of the database state.

Therefore, under set based semantics n denotes the maximum number of times a rule may execute and under instance based semantics, it denotes the maximum number of times a rule may execute on a particular object:

Definition 2: Let *Rules* be a rule set:

1) $Terminate(n,\ Rules,\ set) \stackrel{\text{def}}{=}$
$\forall db \in DB\ \forall Sq \in Seq(Rules,db,set)\ \forall R \in Rules : |\{i \mid Sq_i = R\}| \leq n$
2) $Terminate(n,\ Rules,\ instance) \stackrel{\text{def}}{=}$
$\forall db \in DB\ \forall o \in db\ \forall Sq \in Seq(Rules,db,instance)\ \forall R \in Rules : |\{i \mid Sq_i = R \wedge o_i = o\}| \leq n$

Because all rule sets and all databases are finite, both definitions imply that all execution sequences are finite. That is, they imply termination of the rule-sets[1]:

Theorem 1: Let *Rules* be a rule set and let *sem* denote either set or instance based semantics:

$Terminate(n,\ Rules,\ sem) \rightarrow Terminate(Rules,\ sem)$

Independence

Our detection mechanism for confluency is like that of [15] based on commutativity of rule execution, which is called *Independent*:

Definition 3: Let *Rules* be a rule set:

1) $Independent(Rules,\ set) \stackrel{\text{def}}{=}$
$\forall R_i, R_j \in Rules\ \forall db \in DB :$
$execute(R_i, execute(R_j, db, set), set) = execute(R_j, execute(R_i, db, set), set)$
2) $Independent(Rules,\ instance) \stackrel{\text{def}}{=}$
$\forall R_i, R_j \in Rules\ \forall db \in DB\ \forall o_k, o_l \in db :$
$execute(R_i(o_k), execute(R_j(o_l), db, instance), instance) =$
$execute(R_j(o_l), execute(R_i(o_k), db, instance), instance)$

So, to prove independence, it is sufficient to prove pair-wise independence. In particular, that a rule is independent of itself. Under set based semantics, this is obvious. Under instance based semantics it is not.

Definition 4: Let R be a rule,

$Self\text{-}independent(R) \stackrel{\text{def}}{=} \forall db \in DB : \forall o_i, o_j \in db :$
$execute(R(o_i), execute(R(o_j), db, instance), instance) =$
$execute(R(o_j), execute(R(o_i), db, instance), instance)$

Under both kinds of semantics, independence of rule sets implies the re-arrangability of execution sequences. This re-arrangability is a strong property. First, it enables a straight-forward proof that independence implies confluency for terminating rule sets (see also [15]):

Theorem 2: Let *Rules* be rule set and let *sem* denote either set or instance based semantics:

$Terminate(Rules,\ sem) \wedge Independent(Rules,\ sem) \rightarrow Confluent(Rules,\ sem)$

Secondly, re-arrangability allows for a characterisation of a class of rule sets for which set and instance based semantics coincide, viz., those rule set which are independent and terminate under both semantics:

Theorem 3: Let *Rules* be a rule set. Then

$Terminate(Rules,\ set) \wedge Terminate(Rules,\ instance) \wedge$
$Independent(Rules,\ set) \wedge Independent(Rules,\ instance)$
$\rightarrow$
$\forall db \in DB : E(Rules,db,set) = E(Rules,db,instance)$

[1] All formal proofs can be found in [14].

4 Decidability of the predicates

In this section it is proven that the termination and independence predicates, defined in the previous section, are decidable. That is, it is proven that there exists an effective algorithm that given a rule set *Rules* as input decides the truth value of any of these predicates for *Rules*. In other words, for each rule set it can be determined whether, e.g., it terminates under set based semantics.

The decidability proof is based on reasoning with *pre-* and *post-conditions*. The pre-condition of a rule is given by the selection condition of its query. Its post-condition is determined by its action. To illustrate the reasoning, assume that the post-condition of a rule R implies the negation of its pre-condition. Moreover, assume that if an object o does not satisfy R's pre-condition, then no execution of R by any (set of) object(s) in the database can make o satisfy R's pre-condition. Then $\{R\}$ is clearly a terminating set of rules.

It is well-known that implications between logical expressions can be proven either proof-theoretically or model-theoretically. That is, either syntactically or semantically. Our decidability proof relies on model-theoretic reasoning through the notion of a *typical database state* (tdb).

Such a tdb can be compared to the minimal models of logic programming. Informally, if a certain condition holds in the typical database state after the execution of a rule R, it holds in all database states after the execution of R.

In contrast with minimal models, however, tdbs are parameterized by L_c and L_m. L_c and L_m denote respectively the maximal length of the conditions and the maximal length of the method-calls for which it is typical. The reason for this restriction is that L_c and L_m determine a strictly increasing lowerbound on the number of objects in the typical database state. Because neither conditions nor method-calls are restricted in length, no finite database state can be typical for *all* conditions and method-calls. The formal definition is as follows:

Definition 5: Let tdb be a database state, let $\mathcal{R}$ be the set of all possible rules, and let sem denote either set or instance based semantics. Then tdb is a typical database state for conditions of length L_c and methods of length L_m if

$$\begin{array}{l}\forall C_1, C_2 \in \{C \mid length(C) \leq L_c\}: \\ \forall R = (Q, M) \in \{R \in \mathcal{R} \mid length(C_Q) \leq L_c \wedge\ length(M) \leq L_m\}: \\ \quad \forall db \in DB: ([\forall o \in tdb: C_1(o, tdb) \rightarrow C_2(o, execute(R, tdb, sem))] \\ \qquad \rightarrow [\forall o \in db: C_1(o, db) \rightarrow C_2(o, execute(R, db, sem))])\end{array}$$

This definition is obviously non-constructive. The first result of this section is that there exists a constructive characterisation of typical database states. That is, there exists an effective algorithm for the construction of typical database states. The algorithm is only sketched in this paper, a full description can be found in [14].

The construction of typical database states is based on three observations. The first observation is that a condition induces an equivalence relation on a database state. Reconsider for example the rule *paint-red*, with selection condition *color = red*. For this rule, the actual color of an object is immaterial. All one needs to proof termination is the distinction between *color = red* and *color ≠ red*. So, a database with only two objects, one whose color is red and the other whose color is not, is typical for this particular problem.

The second observation is that the truth of a post-condition depends on the truth of a, related, pre-condition. Consider for example the rule *(red, paint-cell(color = color ∘ neighbor.left)*, which gives red cells the color of their left neighbor. The evaluation of the condition *color = red* for an object *o after* execution of the rule equals the evaluation of *color ∘ neighbor.left = red* for that same object *o before* execution of the rule.

The final observation is that consistency is decidable for our class of conditions. That is, given a condition C it is possible to determine whether there exists a database state db with an object o, such that $C(o, db)$. In fact, if the condition is consistent, the algorithm returns such a pair (o, db), otherwise it fails. The pair (o, db) is called a witness for C.

The consistency check goes roughly as follows. First the condition C is re-ordered according to some suitable order (basically in ascending complexity). Then one starts building a witness database state around an object o, using the elementary conditions of C in order. If at any point the construction of the state cannot be continued C is inconsistent, otherwise it is consistent.

With these three observations the construction of the typical database is roughly as follows:

1. Generate all consistent conditions of length $L_c \times L_m$, together with their witnesses.
2. Merge the witness database states into a final database state.

This final database state is typical for conditions of length L_c and methods of length L_m.

Theorem 4: There exists an effective algorithm for the construction of a typical database for conditions of length L_c and methods of length L_m [2].

Now that typical database states can be constructed, it is relatively straightforward to prove that our predicates are indeed decidable. The basic idea is that one simply executes all possible execution sequences upto a certain maximal length on a typical database state of suitable lengths and checks if the necessary conditions are satisfied. The burden of the proofs lies in determining combinatorial expressions for the lengths of typical database states and the maximally needed length of the execution sequences.

For example, if one wants to check whether two rules R_1 and R_2 commute under set based semantics, one simply executes $R_1; R_2$ and $R_2; R_1$ on a suitable typical database and checks whether the resulting database states are the same.

Theorem 5: Let *Rules* be a rule set, $R \in$ *Rules*, and let $n \in \mathcal{N}$. The predicates *Terminate(n, Rules, set)*, *Terminate(n, Rules, instance)*, *Independent(Rules, set)*, *Independent(Rules, instance)* and *Self_independent(R)* are decidable.

5 Sufficient conditions for the predicates

In the previous section, it is shown that our predicates are decidable. However, the decision algorithm has an unpractically high complexity. Even after the considerable improvements that can be made on the sketch presented above. Therefore, we formulate sufficient conditions of low algorithmic complexity for the predicates in this section.

The decidability algorithm is *model-theoretic*, first a typical database state is generated, and then conditions are checked by execution of all possible execution sequences. That is, conditions are semantically checked by forward reasoning and model-checking. Both tdb-generation and condition checking are of considerable complexity in this approach.

The approach in this section is *proof-theoretic*. It is again performed in two steps. The first step is based on the observation that a post-condition can often be transformed into a pre-condition using the method-call. An example of this phenomenon was already given in the previous section, where it was shown that if a rule gives red cells the color of their left neighbor and we want to evaluate the condition *color* = *red* for an object *o after* it executed the rule, it suffices to evaluate *color* $\circ$ *neighbor.left* = *red* for that same object *o before* execution of the rule.

If the post-condition and the method-call do not lead to a well-defined pre-condition, this simply means that the required post-condition may or may not hold after execution of the rule. Which means that in such cases our predicates are simply not true.

So, the first step is transforming post-conditions into pre-conditions. The second step is checking whether these pre-conditions hold by checking their implication from given knowledge (selection conditions of rules). In fact, both steps rely on *syntactical* derivation of implications.

A priori, there are no evident performance gains in going from model-theoretic reasoning to proof-theoretic reasoning. The low complexity of our sufficient conditions is reached by two relaxations in the proof-theoretic scheme.

[2] Under the assumption that the set of constants is finite which it is in practice.

The first is that we do not pursue completeness for implication proving. Rather, we define a relation $\rightsquigarrow$ of low algorithmic complexity such that:

$$(C(o, db) \rightsquigarrow C'(o, db)) \rightarrow (C(o, db) \rightarrow C'(o, db))$$

The second is that we prove our predicates indirectly. That is, our sufficient conditions imply our predicates but not necessarily vice versa.

The $\rightsquigarrow$ is defined as follows:

Definition 6: Let $e_1, \cdots, e_n$ be functional expressions and let C_1, C_2, C_3 be conditions. The $\rightsquigarrow$ is defined inductively by:

1. the logical rules:
   (a) $C_1 \rightsquigarrow C_1$;
   (b) $C_1 \wedge C_2 \rightsquigarrow C_1$;
   (c) $C_1 \rightsquigarrow C_2 \wedge C_2 \rightsquigarrow C_3 \rightarrow C_1 \rightsquigarrow C_3$
   (d) $C_1 \rightsquigarrow C_2 \rightarrow \neg C_2 \rightsquigarrow \neg C_1$
2. the rules for equalities:
   (a) $e_1 = e_2 \wedge e_2 = e_3 \rightsquigarrow e_1 = e_3$
   (b) Leibnitz restricted to:
      i. $e_1 = e_2 \rightsquigarrow e_3 \circ e_1 = e_3 \circ e_2$;
      ii. $e_1 = e_2 \rightsquigarrow e_1.l = e_2.l$
      iii. $e_1 = e_2 \rightsquigarrow \{e_1, e_3, \cdots e_n\} = \{e_2, e_3, \cdots e_n\}$
      iv. $e_1 = e_2 \rightsquigarrow e_1 \cup e_3 = e_2 \cup e_3$
      v. $e_1 = e_2 \rightsquigarrow e_1 \cap e_3 = e_2 \cap e_3$
3. the rules for sets:
   (a) $e_1 \in e_2 \wedge e_2 \subseteq e_3 \rightsquigarrow e_1 \in e_3$;
   (b) $e_1 \subseteq e_2 \wedge e_2 \subseteq e_3 \rightsquigarrow e_1 \subseteq e_3$;

For the deduction algorithm based on $\rightsquigarrow$, see [14].

The transformation of postconditions into preconditions is described with three substitutions. The first, $|R^+|$, for objects that satisfy the selection condition of rule R and execute its method. The second, $|R^\pm|$, for objects that satisfy the selection condition of rule R but do not execute its method. The third, $|R^-|$, for objects that do not satisfy the selection condition. The notation Sub^n is used to denote n repeated applications of substitution Sub. To exemplify the substitutions, let o be an object in the database and consider the expression $b_n \circ \cdots \circ b_1(o)$, in which all b_i are attributes. Assume that a rule R is executed, then clearly the value of $b_n \circ \cdots \circ b_1(o)$ is invariant if R does not assign a new value to any of the b_i attributes. In all other cases the value of $b_n \circ \cdots \circ b_1(o)$ may change. The actual change depends on whether the argument of an b_i, thus the object $b_{i-1} \circ \cdots \circ b_1(o)$ was subject to the execution of R. The latter can often be derived from the (negated) selection condition of R.

We exemplify the substitutions on expressions. From this, their effect on conditions can be derived straightforwardly.

Let rule R only assign the expression c to the attribute b, let the selection condition of R be $C \equiv e = f \wedge e \circ c \neq f \circ c$ and assume o satisfies it. If we want to know the value of $b(o)$ after execution of R, then we have to know the value before execution of R of either $c(o)$ if o was subject to the execution of R (set or

instance semantics) or of $b(o)$ if o was not subject to the execution of R (instance semantics). Therefore, $b|R|^+ = c$ and $b|R|^\pm = b$.

If we want to know the value of $b \circ c(o)$ after execution of R, then we have to know the value of $b \circ c(o)$ before execution of R. The b in the expression $b \circ c$ is not substituted as we know that this attribute has not been assigned another value as the object $c(o)$ for an arbitrary object o is not selected by a. This can be seen from the selection condition $C \equiv e = f \wedge e \circ c \neq f \circ c$. Whenever an object o satisfies C we know that $e(c(o)) \neq f(c(o))$ holds and thus $c(o)$ does not satisfy the first conjunct of C. Therefore, $b \circ c|R|^+ = b \circ c$ and $b \circ c|R|^\pm = b \circ c$.

For an example of derivation from negated selection conditions, let R still only assign the expression c to attribute b, but let the selection condition be $C \equiv e \neq f \vee e \circ c = f \circ c$ and assume o does not satisfy C. If we want to know the value of $b(o)$ after execution of R, then we only have to know the value of $b(o)$ before execution of R as o is not selected by R and therefore its attributes are not changed. Therefore, $b|R|^- = b$.

If we want to know the value of $b \circ c(o)$ after the execution of R, then we have to know the value of $c \circ c(o)$ before execution of R. The b in the expression $b \circ c(o)$ is substituted by c as we know that the object $c(o)$ is selected by A for all objects o. This can be seen from the negated selection condition $\neg C \equiv e = f \wedge e \circ c \neq f \circ c$. As object o satisfies $\neg C$ we know that $e(c(o)) \neq f(c(o))$ and so $c(o)$ does not satisfy the first conjunct of $\neg C$ and therefore satisfies C. So its attribute b is surely updated with c. Therefore, $b \circ c|R|^- = c \circ c$.

To formalise the transformation of post-conditions, we define some functions. The functions *Sattr*, *Wattr*, and *Uattr* that return the select-attributes, write-attributes and the update-attributes of a rule respectively. The latter is the set of all attributes a rule uses to update its write-attributes. Furthermore, the function *attr* that returns the attributes used in an expression. This function is used for the definition of the other three.

Definition 7:

1. The function *attr.expr* for functional expressions to attribute sets is defined inductively as follows:
   (a) $attr.expr(a) = \{a\}$ if a is an attribute;
   (b) $attr.expr(e_1 \circ e_2) = attr.expr(e_1) \cup attr.expr(e_2)$;
   (c) $attr.expr(e.l) = attr.expr(e)$;
   (d) $attr.expr((l_1 = e_1, \cdots, l_n = e_n)) = \bigcup_{i=1}^{n} attr.expr(e_i)$;
   (e) $attr.expr(\{e_1, \cdots e_n\})) = \bigcup_{i=1}^{n} attr.expr(e_i)$;
   (f) $attr.expr(e_1 \cup e_2) = attr.expr(e_1) \cup attr.expr(e_2)$;
   (g) $attr.expr(e_1 \cap e_2) = attr.expr(e_1) \cup attr.expr(e_2)$.
2. The function *attr* for conditions to attribute sets is defined inductively by:
   (a) $attr(e_1 \omega e_2) = attr.expr(e_1) \cup attr.expr(e_2)$;
   (b) $attr(c_1 \wedge c_2) = attr(c_1) \cup attr(c_2)$;
   (c) $attr(c_1 \vee c_2) = attr(c_1) \cup attr(c_2)$.
3. Let $R = (Q, m(l_1 = e_1, \cdots, l_m = e_m)$ be a rule, with m defined by $m(l_1 : \tau_1, \cdots, l_m : \tau_m) =$ **self except** $a_1 := l_1, \cdots, a_m := l_m$ then:
   (a) $Sattr(R) = attr(C_Q)$;
   (b) $Wattr(R) = \{a_1, \cdots, a_m\}$;
   (c) $Uattr(R) = \bigcup_{i=1}^{m} attr(e_i)$

To simplify the definition of these substitutions, we first introduce the function *link* that is used to determine whether an expression could have been changed by rule execution:

Definition 8: Let C be a condition and let e be a functional expression. The function *link* is inductively defined by:

1. if $C \equiv e_1 \omega e_2$, then $link(C,e) = e_1 \circ e\, \omega\, e_2 \circ e$;
2. if $C \equiv C_1 \wedge C_2$, then $link(C,e) = link(C_1,e) \wedge link(C_2,e)$;
3. if $C \equiv C_1 \vee C_2$, then $link(C,e) = link(C_1,e) \vee link(C_2,e)$;
4. if $C \equiv \neg C_1$, then $link(C,e) = \neg link(C_1,e)$

With this function the substitution rules are defined as follows:

Definition 9: Let $R = (Q, m(l_1 = e_1, \cdots, l_m = e_m)$ be a rule, with m defined by $m(l_1 : \tau_1, \cdots, l_m : \tau_m) =$ **self except** $a_1 := l_1, \cdots, a_m := l_m$ The substitutions $|R^+|$, $|R^-|$, and $|R^\pm|$ on expressions of the form $b_n \circ \cdots \circ b_1$ where the b_i are attributes are defined by:

1. $b_n \circ \cdots \circ b_1 |R^+| = c_n \circ \cdots \circ c_1$ with
$$\forall i \in \{2, \cdots, n\} : c_i = \begin{cases} b_i & \text{if } \forall a \in Wattr(R) : b_i \neq a \vee \\ & \exists C' : C \rightsquigarrow C' \wedge C \rightsquigarrow \neg link(C', b_{i-1} \circ \cdots \circ b_1) \\ ? & otherwise \end{cases}$$
and
$$c_1 = \begin{cases} b_1 & \text{if } \forall a \in Wattr(R) : b_1 \neq a \\ e_i & \text{if } \exists i \in \{1, \cdots, m\} : a_i = b_1 \end{cases}$$
2. $b_n \circ \cdots \circ b_1 |R^-| = c_n \circ \cdots \circ c_2 \circ b_1$ with
$$\forall i \in \{2, \cdots, n\} : c_i = \begin{cases} e_j & \text{if } \exists j \in \{1, \cdots, m\} : a_j = b_i \wedge \\ & \exists C' : \neg C \rightsquigarrow C' \wedge \neg C \rightsquigarrow \neg link(C', b_{i-1} \circ \cdots b_1)) \\ b_i & \text{if } \forall a \in Wattr(R) : a \neq b_i \\ ? & otherwise \end{cases}$$
3. $b_n \circ \cdots \circ b_1 |R^\pm| = c_n \circ \cdots \circ c_2 \circ b_1$ with
$$\forall i \in \{2, \cdots, n\} : c_i = \begin{cases} b_i & \text{if } \forall a \in Wattr(R) : b_i \neq a \vee \\ & \exists C' : C \rightsquigarrow C' \wedge C \rightsquigarrow \neg link(C', b_{i-1} \circ \cdots \circ b_1)) \\ ? & otherwise \end{cases}$$

The question marks indicate that in these cases, it is impossible to derive the related condition. The substitutions on other expressions and conditions are defined by:

Definition 10: Let Sub be one of the three substitutions defined above.

1. $((e_1)\phi(e_2))Sub = (e_1)Sub\ \phi\ (e_2)Sub$ with $\phi \in \{\cap, \cup\}$;
2. $(\{e\})Sub = \{(e)Sub\}$;
3. $(e.l)Sub = ((e)Sub).l$;
4. $(l_1 = e_1, \cdots, l_n = e_n)Sub = (l_1 = (e_1)Sub, \cdots, l_n = (e_n)Sub)$
5. $(e_1 \omega e_2)Sub = (e_1)Sub\ \omega\ (e_2)Sub$ with $\omega \in \{=, \neq, \subset, \not\subset, \in, \notin\}$.

if no question marks arise, otherwise the substitution process fails.

Finally, we have come to the formulation of the sufficient conditions. Rules are independent if they do not change each others select set, do not overwrite each others updates, and do not use attributes for their own updates if they are updated by others.

Theorem 6: Let R_1 and R_2 be two rules with selection condition C_1 and C_2 respectively. Then, under the assumption that the substitutions succeed:

1. *Independent*($\{R_1, R_2\}$, *set*) holds if

   (a):
   $(C_1 \wedge C_2 \rightsquigarrow C_1|R_2^+| \wedge C_2|R_1^+|) \wedge (\neg C_1 \wedge \neg C_2 \rightsquigarrow \neg C_1|R_2^-| \wedge \neg C_2|R_1^-|) \wedge$
   $(C_1 \wedge \neg C_2 \rightsquigarrow C_1|R_2^-| \wedge \neg C_2|R_1^+|) \wedge (\neg C_1 \wedge C_2 \rightsquigarrow \neg C_1|R_2^+| \wedge C_2|R_1^-|)$
   (b) :
   $Wattr(R_1) \cap Wattr(R_2) = \emptyset \wedge Uattr(R_1) \cap Wattr(R_2) = \emptyset \wedge Uattr(R_2) \cap Wattr(R_1) = \emptyset$

2. Let the method call of R_1 be $M_1(l_1 = b_1, \cdots, l_n = b_n)$ then *Self_independent*(R_1) holds if

   (a):
   $(C_1 \rightsquigarrow C_1|R_1^{\pm}|) \wedge (\neg C_1 \rightsquigarrow \neg C_1|R_1^-|)$
   (b):
   $Wattr(R) \cap Uattr = \emptyset \ \vee \ \forall i \in \{1, \cdots, n\} : length(b_i) = 1$

3. *Independent*($\{R_1, R_2\}$, *instance*) holds if conditions of item 1 hold and if for both R_1 and R_2 the conditions of item 2 hold.

The first condition (a) of item one describes the stability of select sets and the second condition (b) of item one expresses that otherones updates are not used and not overwritten.

A simple corolarry of this theorem is based on the disjointness of attribute sets.

Corolarry 1: Let R_1 and R_2 be two rules, then:

1. *Independent*($\{R_1$,$R_2\}$,*set*) holds if

   $(Sattr(R_1) \cup Uattr(R_1) \cup Wattr(R_1)) \cap Wattr(R_2) = \emptyset \wedge$
   $(Sattr(R_2) \cup Uattr(R_2) \cup Wattr(R_2)) \cap Wattr(R_1) = \emptyset \wedge$

2. *Self_independent*(R_1) holds if

   $(Sattr(R_1) \cup Uattr(R_1)) \cap Wattr(R_1) = \emptyset$

3. *Independent*($\{R_1, R_2\}$, *instance*) holds if both R_1 and R_2 satisfy the criteria of items 1 and 2 above.

A set of rules terminates in n steps if the selection condition of each rule R implies a condition C that is falsified in at most n executions of R and if the truth of this condition for an object o can only be changed by the execution of R with subject o. Such a condition C is called a private flag.

Theorem 7: Let $Rules = \{R_1, \cdots, R_n\}$, and let C_i denote the selection condition of R_i. Assume that for each C_i there exists a C_i' such that:

1. C_i' is a conjunct of C_i;
2. $C_i' \rightsquigarrow (\neg C_i'|R_i^+| \vee \cdots \vee \neg C_i'|R_i^+|^n)$
3. $\forall R_j \neq R_i : \text{attr}(C_i') \cap Wattr(R_j) = \emptyset \wedge \text{length}(C_i') = 1$

and all substitutions succeed. Then *Terminate*(*n, Rules, instance*) and *Terminate*(*Rules, set*) hold.

The first item identifies the flag, the second item describes the lowering of the flag in at most n steps, and the third item expresses the privateness of each flag.

To guarantee termination in n steps under set semantics, we need an addititonal restriction which states that whenever a negated selection condition holds for an object this can never be changed by the execution of any active object. For the formulation of this condition, we define the set $Sub\_seq(A, n)$ of repreated substitutions.

Definition: Let *Rules* be a set of rules. Then $\forall i \in \{1, \cdots, n\} : |R_1^{\phi_1}| \cdots |R_n^{\phi_n}| \in Sub\_seq(R, n)$ where for all $j \in \{1, \cdots, i\}$, $\phi_i = -$ if $R_i = R$ and $\phi_i \in \{-, +\}$ if $R_i \neq R$.

The sufficient condition for *Terminate(n, Rules, set)*.

Theorem 8: Let $Rules = \{R_1, \cdots, R_n\}$, and let C_i denote the selection condition of R_i. Assume that for each C_i

1 $\forall j \in \{1, \cdots, n\}\, \forall Sub \in Sub\_seq(A, j) : \neg C_i \rightsquigarrow \neg C_i Sub$;

and for each C_i there exists a flaf C_i' such that:

2 C_i' is a conjunct of C_i;

3 $C_i' \rightsquigarrow (\neg C_i' | R_i^+ | \vee \cdots \vee \neg C_i' | R_i^+ |^n)$

4 $\forall R_j \neq R_i : \text{attr}(C_i') \cap Wattr(R_j) = \emptyset \wedge \text{length}(C_i') = 1$

and all substitutions succeed. Then *Terminate(n, Rules, instance)* and *Terminate(Rules, set)* hold.

6 Conclusions and future work

We have described a powerful design theory to facilitates the static detection of confluency and termination of rule sets under the two pre-dominant rule execution semantics. For this, we introduced several predicates that capture important properties of a rule set and that can be staticly inferred.

For a rule set *Rules*, *Terminate(n, Rules, set)* and *Terminate(n, Rules, instance)* imply termination under set and instance based semantics respectively. While *Independent(Rules, set)* and *Independent(Rules, instance)* imply confluency for terminating rule sets.

We have shown that these predicates are decidable, and we have given sufficient conditions of low algorithmic complexity for all the predicates.

Finally, we have shown that for terminating, independent rule sets the two pre-dominant rule execution semantics coincide.

The most important extension we are planning is to consider complex rules. For example, the rule $R_3 = R_1; R_2$ would first execute R_1 and then R_2 as one atomic action.

Acknowledgments

Thanks to Martin Kersten for helpful comments on an initial draft.

References

[1] S. Ceri and J. Widom. Deriving production rules for incremental view maintenance. In *Proceedings of the 17th International Conference on VLDB*, pages 577–589, 1991.

[2] U. Dayal, A. Buchmann, and D.R. McCarthy. Rules are objects too: a knowledge model for an active object oriented dbms. In *Proceedings of the Second International Workshop on Object-Oriented Database Systems*, pages 129–143, 1988.

[3] O. Diaz, N. Paton, and P. Gray. Rule management in object oriented databases a uniform approach. In *Proceedings of the 17th International Conference on VLDB*, pages 317–326, 1991.

[4] S. Gatziu, A. Geppert, and K.R. Dittrich. Integrating active concepts into an object-oriented database system. In *Proceedings of the 3th International Workshop on DBPL*, pages 341–357, 1991.

[5] E.N. Hanson. An initial report on the design of ariel: A dbms with an integrated production rule system. In *SIGMOD RECORD*, volume 18, pages 12–19, 1989.

[6] R. Hull and D. Jacobs. Language constructs for programming active databases. In *Proceedings of the 17th International Conference on VLDB*, pages 455–467, 1991.

[7] J.W. Klop. Term rewriting systems: A tutorial. In *Bull. European Assoc. Theoretical Computer Science*, volume 32, pages 143–183, 1987.

[8] A.M. Kotz, K.R. Dittrich, and J.A. Mulle. Supporting semantics rules by a generalized event/trigger mechanism. In *Advances in Database Technology: EDBT 90, LNCS 416*, pages 76–91, 1990.

[9] U. Schreier, H. Pirahesh, R. Agrawal, and C. Mohan. Alert, an architecture for transforming a passive dbms into an active dbms. In *Proceedings of the 17th International Conference on VLDB*, pages 469–478, 1991.

[10] A.P.J.M. Siebes, M.H. van der Voort, and M.L. Kersten. Towards a design theory for database triggers. In *Database and Expert Systems Applications*, pages 338–344, 1992.

[11] E. Simon and C. deMaindreville. Deciding whether a production rule is relational computable. In *Proceedings of the ICDT 88, LNCS 326*, pages 205–222, 1988.

[12] M. Stonebraker, E. Hanson, and C.H. Hong. The design of the postgres rule system. In *Readings in Database Systems, eds. M. Stonebraker*, pages 556–565, 1988.

[13] M. Stonebraker, A. Jhingran, J. Goh, and S. Potamianos. On rules, procedures, caching and views in database systems. In *Proceedings of the ACM SIGMOD conference*, pages 281–290, 1990.

[14] M.H. van der Voort and A.P.J.M. Siebes. A design theory for active objects. Technical report, CWI, 1993.

[15] A. Aiken J. Widom and J.M. Hellerstein. Behavior of database production rules: Termination, confluence, and observable determinism. In *SIGMOD RECORD*, volume 21, pages 59–68, 1992.

[16] J. Widom and S.J. Finkelstein. Set-oriented production rules in relational database systems. In *Proceedings of the ACM SIGMOD conference*, pages 259–270, 1990.

[17] Y. Zhou and M. Hsu. A theory for rule triggering systems. In *Advances in Database Technology: EDBT 90, LNCS 416*, pages 407–422, 1990.