Università degli Studi di Roma Tre
Dipartimento di Informatica e Automazione
Via della Vasca Navale, 84 – 00146 Roma, Italy

Analysis and Optimization of Active Databases

Danilo Montesi†, Riccardo Torlone‡

RT-INF-12-96 **October 1996**

† Dip. di Scienze dell'Informazione
Universita' di Milano
Via Comelico 39/41
20135 Milano, Italy

‡ Dip. di Informatica e Automazione
Università di Roma Tre
Via della Vasca Navale, 84
00146 Roma, Italy

A preliminary version of this paper appeared in [17]. This work was partially supported by Consiglio Nazionale delle Ricerche *and by* NATO.

ABSTRACT

We introduce a new formal semantics for active databases that relies on a transaction rewriting technique. A user defined transaction, which is viewed here as a sequence of atomic database updates forming a semantic unit, is translated by means of active rules into induced one(s). Those transactions embody active rule semantics which can be either immediate or deferred. Rule semantics, confluence, equivalence and optimization are then formally investigated and characterized in a solid framework that naturally extends a known model for relational database transactions.

Keywords: Active databases, rule semantics, transaction equivalence, confluence, optimization.

1 Introduction

Active databases are based on rules that allow us to specify actions to be taken by the system automatically, when certain events occur and some conditions are met. It is widely recognized that these *active* rules provide a powerful mechanism for the management of several important database activities (e.g., constraint maintenance and view materialization [6, 7]), and for this reason, they are now largely used in modern database applications and have been extensively studied in the last years [2, 4, 5, 9, 12, 14, 21, 22, 23]. However, in the various approaches, active rule execution is generally specified only by informal, natural-language descriptions. It follows that very often, when the number of rules increases, active rule processing becomes quickly complex and unpredictable, even for relatively small rule sets [23].

The goal of this paper is to provide a formal approach to active rule processing that relies on a method for rewriting user defined transactions to reflect the behavior of a set of active rules, and to show how known results for transaction equivalence can be extended in this framework to pre-analyze properties of transactions and rules.

We start by introducing a simple transaction language, based on a well known model for relational databases [1] in which a transaction is viewed as a collection of basic update operations forming a semantic unit, and a quite general active rule language, whose computational model is set-oriented (like in [23] and differently from other approaches [22]). We consider two different execution models for active rules: immediate and deferred (or delayed) [8, 14]. The former has no temporal decoupling between the event, condition and action parts. The latter has a temporal decoupling between the event part on one side and the condition and action parts on the other side. We then define in this context a rewriting process that takes as input a user defined transaction t and a set of active rules and produces a new transaction t' that "embodies" active rule semantics, in the sense that t' explicitly includes the additional updates due to active processing. Under the deferred modality, the new transaction is the original one augmented with some induced actions, whereas, under the immediate modality, the new transaction interleaves original updates and actions defined in active rules. It follows that the execution of the new transaction in a passive environment corresponds to the execution of the original transaction within the active environment defined by the given rules. Other approaches consider rewriting techniques [11, 22], but usually they apply in a restrictive context or are not formal. Conversely, we believe that this formal and simple approach can improve the understanding of several active concepts and make it easier to show results.

As we have said, the execution model of our transactions extends a relational transaction model which has been extensively investigated [1]. The reason for this choice is twofold. Firstly, we wish to use a well known framework having a formal setting and a solid transaction execution model. Secondly, we wish to take full advantage of the results already available on transaction equivalence and optimization [1, 13]. In this way, we are able to formally investigate statically several interesting properties of active rule processing. First, we can check whether two transactions are equivalent in an active database. Then, due to the results on transaction equivalence, we are also able to provide results on confluence. Finally, optimization issues can be addressed. As a final remark, we note that, with this approach, active rule processing does not require any specific run-time support, and so it is simpler to implement than others which are built from scratch [10].

The remainder of this paper is organized as follows. In Section 2, a detailed overview

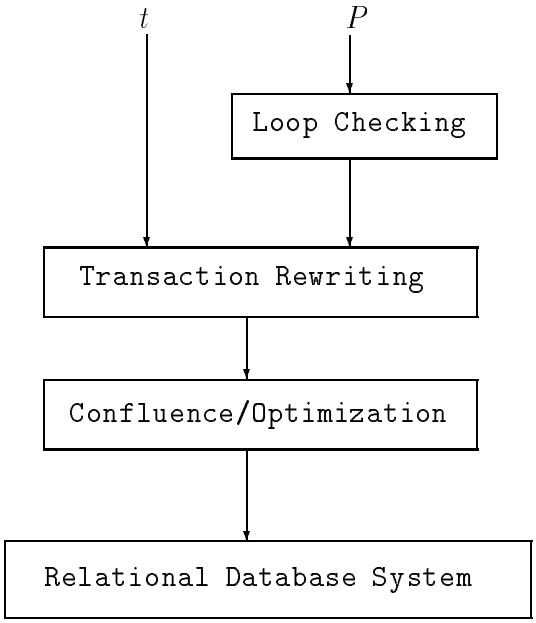

Figure 1: Components of the approach

of the approach is presented by using several practical examples. In Section 3 we define the basic framework. In Sections 4 and 5, we introduce, in a systematic way, the notion of active database and the rewriting transaction technique, respectively. The property of equivalence is investigated in Section 6. From this study, several results on active rule processing are derived in Section 7. Finally, in Section 8, we draw some conclusions.

2 An overview of the approach

In this section we informally present our approach. As described in Figure 1, the basic idea is to express active rule processing as a four step computation. Given a user defined transaction t and a set of active rules P, the first step checks whether P presents some kind of recursion. For the time being, we present a simple characterization and we will not address this issue in detail in the present paper. The second step takes P and t, and transforms the transaction t into an induced one(s) that "embodies" the semantics of the rules in P. In general, during this step several transactions can be generated. These different induced transactions take into account the fact that an update of the original transaction may trigger several rules at the same time, and so the corresponding actions can be executed in different orders yielding different results. In the third step, confluence and optimization issues of active rule processing are investigated by analyzing the transactions computed during the second step. This is done by extending known techniques for testing equivalence of database transactions [1, 13]. Then, in the last step, according to the results of this analysis, one transaction is finally submitted to a relational database management system.

We point out two important aspects of this approach. Firstly, it relies on a formal basis

that allows us to derive solid results. Secondly, the rewriting and confluence/optimization steps can be done statically, without accessing the underlying database, and therefore they can be performed very efficiently at compile time.

As we have said, we will consider the immediate and deferred active rule execution models: the immediate modality reflects the intuition that rules are processed as soon as they are triggered, while deferred modality suggests that a rule is evaluated and executed after the end of the original transaction [14]. Thus, two different rewriting procedures will be given. Specifically, consider a user defined transaction as a sequence of updates: $t = u_1; \ldots; u_n$. This transaction is transformed under the immediate modality into an *induced* one:

$$t_I = u_1; \mu_1^P; \ldots; u_n; \mu_n^P.$$

where μ_i^P denotes the sequence of updates computed as *immediate reaction* of the update u_i with respect to a set of active rules P. This reaction can be derived by "matching" the update u_i with the event part of the active rules. Clearly the obtained updates can themselves trigger other rules, hence this reaction is computed recursively. As noted above, several transactions can be obtained in this way. Note that under the immediate modality the induced transaction is an interleaving of user defined updates with rule actions.

Under the deferred modality, the induced transaction has the form:

$$t_D = u_1; \ldots; u_n; \mu_1^P; \ldots; \mu_n^P.$$

Hence the *reaction is deferred* (or postponed) until the end of the user transaction. Here again the induced updates can themselves trigger other rules, and so the reactions of the original updates are recursively computed, but using the immediate modality.

We now give a number of practical examples to clarify the above discussion. The following active rules react to updates to a personnel database composed by two relations: `emp(name,dname,sal)` and `dep(dname,mgr)`. Rules are expressed here in a generic language that does not refer to any specific system but whose intended meaning should be evident (indeed, those rules can be easily expressed in any practical active rule language).

```
r1: When DELETED d FROM dep
    Then DELETE FROM emp WHERE dname=d.dname
r2: When INSERTED new-e INTO emp
    Then DELETE FROM emp
         WHERE name=new-e.name AND dname<>new-e.dname
r3: When INSERTED new-e INTO emp
    If   new-e.sal > 50K
    Then INSERT INTO dep
         VALUES (dname=new-e.dname, mgr=new-e.name)
```

Intuitively, the first rule states that when a department is deleted then all the employees working in such a department must be removed (cascading delete). The second one serves to enforce the constraint that an employee can work in one department only, and states that when an employee tuple, say `(john,toy,40K)` is inserted into the relation `emp`, then the old tuples where `john` is associated with a department different from `toy` must be deleted. Finally, the last rule states that if an inserted employee has a salary

greater than 50k then he is eligible to be a manager of the department in which he works and so, according to that, a tuple with the name of the department and the new employee is inserted in the relation dep.

Now, we provide the following simple user defined transaction where first the toy department is removed and then an employee is added to this department with a salary of 60K.

```
t1: DELETE FROM dep WHERE dname='toy';
    INSERT INTO emp VALUES (name='bill',dname='toy',sal=60K);
```

By inspecting the given active rules, we can easily realize that, at run time, the first update in t1 will trigger rule r1, whereas the second update will trigger rules r2 and r3. Therefore, under immediate modality, t1 can be rewritten, at compile time, into the following transaction t1I (where I denotes immediate modality), by "unfolding" t1 with respect to the active rules. In this new transaction, the prefix * denotes an induced update.

```
t1I: DELETE FROM dep WHERE dname='toy';
    *DELETE FROM emp WHERE dname='toy';
     INSERT INTO emp VALUES (name='bill',dname='toy',sal=60K);
    *DELETE FROM emp WHERE name='bill' AND dname<>'toy';
    *INSERT INTO dep VALUES (dname='toy',name='bill');
```

The obtained transaction describes the behavior of the transaction t1 taking into account the active rules under the immediate modality. Note that there is another possible translation in which the last two updates are switched. This is because the second update of the original transaction triggers two rules at the same time (namely r2 and r3) and therefore we have two possible execution orders of the effects of these rules. It follows that, in general, a user defined transaction actually induces a *set* of transactions. One of the goals of this paper is to show that, in many cases, it is possible to statically check whether these transactions are equivalent. If all the induced transactions are equivalent we can state that the active program is "confluent" with respect to the transaction t1. In this case the execution of one of the obtained transactions implements the expected behavior of the user defined transaction within the active framework. Note that we do not assume the presence of a (partial) ordering on the rules, but the framework can be easily extended to take it into account.

Let us now turn our attention to the deferred execution model. Assume that we want to move the employee John from the toy to the book department. This can be implemented by means of the following transaction.

```
t2: INSERT INTO emp VALUES (name='john',dname='book',sal=50K);
    DELETE FROM emp WHERE name='john' AND dname='toy' AND sal=50K;
```

By inspecting transaction and rules, we can statically decide that the first update in t2 will not trigger rule r3 since its condition will not be satisfied (the salary of the new employee is not greater than 50K). Thus, if we rewrite this transaction taking into account our active rules under the deferred modality, we have the following possible translation t2D (D denotes deferred modality), in which the effect of the rules is postponed to the end of the transaction.

```
t2D: INSERT INTO emp VALUES (name='john',dname='book',sal=50K);
     DELETE FROM emp WHERE name='john' AND dname='toy' AND sal=50K;
    *DELETE FROM emp WHERE name='john' AND dname<>'book';
```

Now, before executing the above transaction, we can observe the fact that the second update can be discarded without altering the overall effect of the transaction, since its effect is "included" in the effect of the third update. This shows how some optimization can be performed on those induced transactions. The transaction that implements the expected behavior is then as follows.

```
t2D': INSERT INTO emp VALUES (name='john',dname='book',sal=50K);
     *DELETE FROM emp WHERE name='john' AND dname<>'book';
```

In contrast to a user defined transaction, the updates in the induced transactions are not independent, as some updates are indeed "induced" by others. This fact has a consequence on the execution semantics of an induced transaction. Assume for instance that at run-time the execution of an update u in a induced transaction t has a null effect on the database (because, for example, its condition does not hold or its effect is invalidated by a subsequent update). Then, it is reasonable that the updates in t induced (directly or indirectly) by u are not executed as well. Under this interpretation, we need to define a new transaction semantics that takes into account the inducer/induced relationship among updates. Clearly, the techniques to achieve confluence and optimization must take into account this fact.

To clarify the point, consider the transformation of the transaction `t1` under the deferred modality. According to the previous discussion, the rewriting process should generate the following transaction.

```
t1D: DELETE FROM dep WHERE dname='toy';
     INSERT INTO emp VALUES (name='bill',dname='toy',sal=60K);
    *DELETE FROM emp WHERE dname='toy';
    *DELETE FROM emp WHERE name='bill' AND dname<>'toy';
    *INSERT INTO dep VALUES (dname='toy',name='bill');
```

However, it is easy to see that the third update invalidates the effect of second one. It follows that the last two updates of the transaction `t1D`, which are induced by such an update, must not be executed at run time. So, the rewriting of the transaction `t1` under the deferred modality can be simplified as follows:

```
t1D': DELETE FROM dep WHERE dname='toy';
     *DELETE FROM emp WHERE dname='toy';
```

Thus, we need to develop specific techniques to check equivalence and to optimize *induced* transactions. This will be done by extending an already existing framework for equivalence and optimization in relational databases.

The rest of the paper is devoted to the formalization and characterization of the issues discussed in this section.

3 A model for Transactions

The notion of transaction we use in this paper is based on a model for relational transactions introduced by Abiteboul and Vianu [1]. Informally, for transaction we mean a sequence of basic update operations (namely, insertions and deletions of tuples) viewed as a semantic unit. Specifically, we will restrict our attention to the important class of "domain-based" transactions, where the selection of tuples involves the inspection of individual values for each tuple. Differently from the model described in [1], we also allow arithmetic comparisons predicates.

3.1 Preliminaries

Let U be a finite set of symbols called *attributes* and, for each $A \in U$, let $dom(A)$ be an infinite set of constants called the *domain of* A. As usual, we use the same notation A to indicate both the single attribute A and the singleton $\{A\}$. Also, we indicate the union of attributes (or sets thereof) by means of the juxtaposition of their names. Moreover, we assume, for technical reasons, that the domains are disjoint and totally ordered. A *relation scheme* is an object $R(X)$ where R is the name of the relation and X is a subset of U. A *database scheme* S over U is a collection of relation schemes $\{R_1(X_1), \ldots, R_n(X_n)\}$ with distinct relation names such that the union of the X_i's is U.

A tuple v over a set of attributes X is a function from X to the union of all the domains such that, for each $A \in X$, $v(A)$ is in $dom(A)$. A relation over a relation scheme $R(X)$ is a finite set of tuples over X. A *database instance* s over a database scheme S is a function from S such that, for each $R(X) \in S$, $s(R(X))$ is a relation over $R(X)$. We will denote by $Tup(X)$ the set of all tuples over a set of attributes X and by $Inst(S)$ the set of all database instances over a database scheme S.

Throughout the rest of the paper, we will always refer to a fixed database scheme $S = \{R_1(X_1), \ldots, R_n(X_n)\}$ over a set of attributes U.

3.2 Conditions

Let X be a set of attributes and A be an attribute in X. An *atomic condition* over X is an expression of the form: (1) $A\theta c$, where $c \in dom(A)$ and θ is a comparison predicate $(=, \neq, <, \leq, \geq, >)$, or (2) $A \in (c_1, c_2)$, where $c_1 \in dom(A) \cup \{-\infty\}$, $c_2 \in dom(A) \cup \{+\infty\}$ and $c_1 < c_2$. The meaning of the symbols $-\infty$ and $+\infty$ is evident: $A \in (-\infty, c_2)$ is equivalent to $A < c_2$. The reason for allowing this form of atomic condition will be clarified shortly.

Definition 3.1 (Condition) *A* complex condition *(or simply a* condition*) over a set of attributes X is a finite set of atomic conditions over X. A tuple v over X satisfies an atomic condition $A\theta c$ ($A \in (c_1, c_2)$) if $v(A)\theta c$ ($c_1 < v(A) < c_2$). A tuple satisfies a condition C if it satisfies every atomic condition occurring in C.*

We assume that conditions are always *satisfiable*, that is, they do not contain atomic conditions that are always false (e.g., $A \in (c_1, c_2)$ and there is no $c \in dom(A)$ such that $c \in (c_1, c_2)$) or atomic conditions that are mutually exclusive (e.g., both $A = c$ and $A \neq c$).

A condition C over a set of attributes X uniquely identifies a set of tuples over X: those satisfying the condition. This set is called the *target* of C.

Definition 3.2 (Target of a condition) *The* target *of a condition C over a set of attributes X, denoted by $Targ(C)$, is the set of tuples $\{v \in Tup(X) \mid v \text{ satisfies } C\}$.*

Note that $Targ(C)$ is not empty if and only if C is satisfiable. We say that a condition C over $X = A_1 \ldots A_n$ specifies a complete tuple if $C = \{A_1 = a_1, \ldots, A_n = a_n\}$, where $a_i \in dom(A_i)$ for $i = 1, \ldots, n$.

3.3 Transactions

Let us first introduce the basic update operations.

Definition 3.3 (Insertion) *An* insertion *over a relation scheme $R_j(X_j) \in S$ is an expression of the form $+R_j[C]$, where C is a condition over X_j that specifies a complete tuple. The effect of an insertion $+R_j[C]$ is a mapping $Eff(+R_j[C])$ from $Inst(S)$ to $Inst(S)$ defined, for each $R_i(X_i) \in S$, by:*

$$Eff(+R_j[C])(s)(R_i(X_i)) = \begin{cases} s(R_i(X_i)) \cup Targ(C) & \text{if } i = j \\ s(R_i(X_i)) & \text{if } i \neq j \end{cases}$$

Definition 3.4 (Deletion) *A* deletion *over a relation scheme $R_j(X_j) \in S$ is an expression of the form $-R_j[C]$, where C is a condition over X_j. The effect of a deletion $-R_j[C]$ is a mapping $Eff(-R_j[C])$ from $Inst(S)$ to $Inst(S)$ defined, for each $R_i(X_i) \in S$, by:*

$$Eff(-R_j[C])(s)(R_i(X_i)) = \begin{cases} s(R_i(X_i)) - Targ(C) & \text{if } i = j \\ s(R_i(X_i)) & \text{if } i \neq j \end{cases}$$

An *update* over a relation scheme is an insertion or a deletion. Note that, for sake of simplicity, we do not consider modify operations here. Actually, similarly to [1], modifications can be accommodated in our framework but the complexity of notation would increase dramatically.

Update operations are generally executed within *transactions*, that is, collections of data manipulation operations viewed as a semantic atomic unit.

Definition 3.5 (User transaction) *A* user transaction *is a finite sequence of updates. The* effect *of a transaction $t = u_1, \ldots, u_n$ is the composition of the effects of the updates it contains, that is, is the mapping: $Eff(t) = Eff(u_1) \circ \ldots \circ Eff(u_n)$.*

Example 3.1 *The SQL transactions described in Section 2 can be easily expressed using the notation introduced above. For instance, transaction* `t1` *at page 6, can be expressed as follows:*

$$t_1 = -\texttt{dep[dname=toy]};+\texttt{emp[name=bill,dname=toy,sal=60k]}$$

Two user transactions are equivalent when they always produce the same result if applied to the same database instance, that is, when they have the same effect.

Definition 3.6 (Equivalence of user transactions) *Two user transaction t_1 and t_2 are* equivalent *(denoted $t_1 \sim t_2$) if it is the case that $Eff(t_1) = Eff(t_2)$.*

3.4 Normalization of transactions

According to [1], we describe and characterize in this section transactions satisfying a property called *normal form*. In such transactions, syntactically distinct updates have disjoint targets (and therefore do not interfere). This is a very convenient form since it simplifies results and algorithms. Moreover, it will make easier the specification of the reaction of active rules to updates involved in a transaction. We also show that any transaction can be brought to this special form by means of a "preprocessing" phase called *normalization*, and that this operation can be performed in polynomial time.

Definition 3.7 (Normal form) *A transaction t is in* normal form *if, for each pair of updates u_i and u_j in t that are over the same relation and have conditions C_i and C_j such that $C_i \neq C_j$, it is the case that $Targ(C_i) \cap Targ(C_j) = \emptyset$.*

The following result easily follows by definitions and states that, in a transaction in normal form: (1) if two updates have different targets, then these targets have an empty intersection, and (2) if two updates have the same target, then they have the same condition.

Lemma 3.1 *In a transaction in normal form: (1) the targets of a pair of updates are either identical or disjoint, and (2) the conditions of a pair of updates having the same target are syntactically equal.*

Proof. (1) Assume by way of contradiction that in a transaction in normal form there are two updates u_i and u_j with conditions C_i and C_j such that $Targ(C_i) \neq Targ(C_j)$ and $Targ(C_i) \cap Targ(C_j) \neq \emptyset$. Since the targets of u_i and u_j are different, $C_i \neq C_j$, but by Definition 3.7, this implies that their targets are disjoint — a contradiction.
(2) Assume that in a transaction in normal form two updates with the same target have different conditions. By definition of transaction in normal form, this implies that their targets are indeed disjoint — a contradiction. □

Each transaction can be transformed into an equivalent transaction in normal form by "splitting" the target of each condition into sufficiently many targets. To this end, we now introduce a number of axioms, called *Split Axioms*, that can be used to transform a transaction into an equivalent transaction in normal form. Intuitively, these axioms show: (1) how we can transform a transaction in an equivalent transactions that contains only atomic conditions of the form $\{A = c\}$ or $\{A \in (c_1, c_2)\}$ (axioms SA2—SA4), and (2) how we can further transform a transaction in such a way that possible interferences between pair of updates of these two forms can be avoided (axioms SA5 and SA6).[1] Axioms SA1 and SA7 are useful in order to apply the others. Specifically, the former shows how we can generate conditions composed by singletons over the various attributes, the latter shows how we can add a condition over an attribute to a condition that does not mention it.

Definition 3.8 (Split Axioms) *In the following axioms, called the* Split Axioms, *C is a condition over X, $A \in X$, and $C|_Y$ denotes the set of atomic conditions in C that are over Y:*

[1] Clearly, this is just one of the possible way to enforce the normal form.

(SA1) $-R[C] \approx_{\text{split}} -R[C|_{X-A} \cup \{C_A^{(1)}\}]; \ldots; -R[C|_{X-A} \cup \{C_A^{(k)}\}]$
where $C|_A = \{C_A^{(1)}, \ldots, C_A^{(k)}\}$ *and* $C_A^{(i)}$, *for* $i = 1, \ldots, k$, *is an atomic condition.*

(SA2) $-R[C] \approx_{\text{split}} -R[C|_{X-A} \cup \{A \in (-\infty, c)\}]; -R[C|_{X-A} \cup \{A \in (c, +\infty)\}]$
where $C|_A = \{A \neq c\}$.

(SA3) $-R[C] \approx_{\text{split}} -R[C|_{X-A} \cup \{A = c\}]; -R[C|_{X-A} \cup \{A \in (c, +\infty)\}]$
where $C|_A = \{A \geq c\}$.

(SA4) $-R[C] \approx_{\text{split}} -R[C|_{X-A} \cup \{A = c\}]; -R[C|_{X-A} \cup \{A \in (-\infty, c)\}]$
where $C|_A = \{A \leq c\}$.

(SA5) $-R[C] \approx_{\text{split}} -R[C|_{X-A} \cup \{A = c\}]$;
where $C|_A = \{A \in (c_1, c_2)\}$ *and* c *is the only element in* $dom(A)$ *such that* $c_1 < c < c_2$.

(SA6) $-R[C] \approx_{\text{split}} -R[C|_{X-A} \cup \{A = c\}]; -R[C|_{X-A} \cup \{A \in (c_1, c)\}]; - R[C|_{X-A} \cup \{A \in (c, c_2)\}]$
where $C|_A = \{A \in (c_1, c_2)\}$, $c \in dom(A)$ *and* $c_1 < c < c_2$.

(SA7) $-R[C] \approx_{\text{split}} -R[C|_{X-A} \cup \{A = c\}]; -R[C|_{X-A} \cup \{A \in (-\infty, c)\}]; - R[C|_{X-A} \cup \{A \in (c, +\infty)\}]$
where $C|_A = \emptyset$ *and* $c \in dom(A)$.

The following result can be easily proved.

Lemma 3.2 *The split axioms are* sound, *that is,* $t_1 \approx_{\text{split}} t_2$ *implies* $t_1 \sim t_2$.

Proof. Let us consider, for instance, axiom SA2. Let $u = -R[C]$ where $C|_A = \{A \neq c\}$, and let $u_1 = -R[C_1] = -R[C|_{X-A} \cup \{A \in (-\infty, c)\}]$ and $u_2 = -R[C_2] = -R[C|_{X-A} \cup \{A \in (c, +\infty)\}]$. It is easy to see that $Targ(C) = Targ(C_1) \cup Targ(C_2)$ and so, by Definitions 3.4 and 3.5, we have that $\mathit{Eff}(u) = \mathit{Eff}(u_1) \circ \mathit{Eff}(u_2)$, that is, $u \sim u_1; u_2$. Similar considerations apply to the other axioms. □

We now show how these axioms can be practically used to normalize transactions. Let us first introduce a property of conditions to be used in the algorithm that follows.

Property 3.1 *Let* C *be a condition over* X, Z *be a set of attributes and* $\mathcal{C}$ *be a finite set of constants. Then, for each attribute* $A \in X \cap Z$ *(1)* $C|_A$ *has the form* $\{A = c\}$ *or* $\{A \in (c_1, c_2)\}$, *and (2) if* $C|_A = \{A \in (c_1, c_2)\}$, *then there is no* $c \in dom(A) \cap \mathcal{C}$ *such that* $c \in (c_1, c_2)$.

The split algorithm that can be used to normalize transactions is reported in Figure 2. We have the following result.

Theorem 3.1 *Let* t *be a transaction,* $\mathcal{C}$ *be the set of all constants appearing in* t, *and* Z *be the set of all attributes mentioned in* t. *Then, (1) Algorithm* SPLIT *terminates over* t, $\mathcal{C}$ *and* Z *and generates a transaction* t_{split} *in polynomial time,*[2] *(2)* $t_{\text{split}} \sim t$, *and (3)* t_{split} *is in normal form.*

[2]Hereinafter, polynomial time means time polynomial with respect to the length of the transaction.

Algorithm SPLIT
Input: A transaction t, a set of constants $\mathcal{C}$ and a set of attributes Z;
Output: A new transaction t_{split};
begin
  $i := 1$;
  $t_i := t$;
  while (there is an update u over $R_i(X_i)$ in t_i whose condition does not satisfies Property 3.1 for the sets of attributes X_i and Z and the set of constants $\mathcal{C}$)
    t_{i+1} := the transaction obtained from t_i by splitting u according to some split axiom;
    $i := i + 1$;
  endwhile;
  $t_{\text{split}} := t_i - \{$updates with unsatisfiable conditions$\}$
end.

Figure 2: Algorithm SPLIT

Proof. (1) Assume that $|t| = 1$ (that is, t contains just one update). By the structure of the split axioms, at each iteration of the loop in Algorithm SPLIT, we have $|t_{i+1}| > |t_i|$. Moreover, the algorithm tries to enforce Property 3.1 that allows only updates having atomic conditions of the form $A = c$ and $A \in (c_1, c_2)$. It follows that the number of different forms that each atomic condition in t_{i+1} can take during the execution of the algorithm is bounded by:

$$m = \left(\begin{array}{c} k \\ 2 \end{array} \right) + |\mathcal{C}|.$$

where $k = |\mathcal{C}| + 2$. This corresponds to the number of ordered pairs of k symbols (for the atomic conditions of the form $A \in (c_1, c_2)$) plus the cardinality of $\mathcal{C}$ (for the atomic conditions of the form $A = c$). Since each update is not split more than once with respect to the same constant, it follows that for every i, $|t_i|$ is bounded by $m^{|U|}$, that is, by the number of complex conditions over the universe U of attributes that can be formed with m different atomic conditions. Thus, the sequence of the t_i's is strictly increasing and bounded and therefore the algorithm terminates. If t contains multiple updates, the SPLIT algorithm can be applied separately to each update in t and the results can be then concatenated to obtain t_{split}. It follows that Algorithm SPLIT terminates over any transaction and generates the output transaction in polynomial time.
(2) This part can be easily shown on the basis of Lemma 3.2, by induction on the number of transformations applied to t by Algorithm SPLIT.
(3) By way of contradiction, assume that t_{split} is not in normal form, that is, there is a pair of updates over the same set of attributes X with syntactically different conditions, say C_i and C_j, whose targets are not disjoint. Now let A be an attribute in X such that $C_i|_A \neq C_j|_A$. Since t_{split} satisfies Condition (1) of Property 3.1 (being the output of the algorithm SPLIT) we have that both $C_i|_A$ and $C_j|_A$ are of the form $A = c$ or $A \in (c_1, c_2)$. Then, we have two possible cases : (a) $C_i|_A$ has the form $A = c$, $C_j|_A$ has the form

$A \in (c_1, c_2)$ and $c \in (c_1, c_2)$ (otherwise the targets would be disjoint), and (b) $C_i|_A$ has the form $A \in (c_{i,1}, c_{i,2})$, $C_j|_A$ has the form $A \in (c_{j,1}, c_{j,2})$ and $c_{j,1} < c_{i,2}$ (as above). In both cases, at least one of the two conditions does not satisfy Condition (2) of Property 3.1, and this contradicts that t_{split} is the output of Algorithm SPLIT. □

Example 3.2 *Let $S = \{R_1(A), R_2(AB)\}$ with domains over the integers and consider the transaction $t = +R_1[A = 2]; -R_1[A \neq 1]$. This transaction is not in normal form since the tuple $v = (2)$ over $R_1(A)$ is in both the targets of the two updates it contains. By applying Algorithm* SPLIT *we obtain:*

1. $t_1 = +R_1[A = 2]; -R_1[A < 1]; -R_1[A > 1]$[3] *(by axiom SA2),*
2. $t_2 = +R_1[A = 2]; -R_1[A < 1]; -R_1[A \in (1, 2)]; -R_1[A = 2]; -R_1[A > 2]$ *(by axiom SA6)*

The algorithm terminates at the second step and outputs

$$t_{\text{split}} = +R_1[A = 2]; -R_1[A < 1]; -R_1[A = 2]; -R_1[A > 2]$$

which is in normal form.

Let us now consider the transaction $t' = -R_2[A = 1]; +R_2[A = 1, B = 2]$, which is not in normal form. Note that t' does not satisfies Property 3.1 since the first update does not even mention attribute B. By applying Algorithm SPLIT *we then obtain:*

1. $t'_1 = -R_2[A = 1, B < 2]; -R_2[A = 1, B = 2]; -R_2[A = 1, B > 2]; +R_2[A = 1, B = 2]$ *(by axiom SA7)*

The algorithm terminates at the first step and outputs $t'_{\text{split}} = t'_1$ which is indeed in normal form.

4 Active Databases

In this section we introduce formally the notions of active rule and program. We will consider a simple form of active rules that however captures a considerable portion of rules described in the literature and implemented in the systems. In particular, in [19] we have considered the core of several concrete active rule languages whose rule execution is specified only by informal descriptions, and have shown that with our framework it is possible to describe the main features of these languages.

4.1 Active Rules and Programs

We represent an active rule by using the same notation introduced to express transactions. This allows us to easily describe the way in which updates and active rules interact. Specifically, the event and the condition parts of a rule are described by an update whereas the action part is described by a transaction according to the definitions of Section 3.3, with the only difference that variables can be used in the place of constants. These variables are used to describe bindings that are passed from the event and the condition to the action of a rule.

[3] For simplicity, in this example we write $A < 1$ ($A > 1$) instead of $A \in (-\infty, 1)$ ($A \in (1, +\infty)$).

Thus, let us fix a set of symbols called *variables*. We call *generalized update* an update having a condition in which variables can occur in the place of constants, and *generalized transaction* a transaction composed by generalized updates. Active rules are then defined as follows.

Definition 4.1 (Active rule) *An* active rule *has the form:*

$$u_e \Rightarrow t_a$$

where: (1) u_e is a generalized update such that, in the case of an insertion, the condition does not necessarily specify a complete tuple; and (2) t_a is a generalized transaction such that each variable occurring in t_a also occurs in u_e.

The left hand side and the right hand side of the rule are also called the *event part* and the *action part*, respectively.

Definition 4.2 (Active database) *An* active program *P is a set of active rules. An* active database *is a pair (s, P) where s is a database instance and P is an active program.*

Note that, the event part of an active rule allows us to specify both the update triggering the rule and the condition to be met for the effective execution of the rule [23]. The intuitive semantics of a rule as above is then as follows: if an update u "matching" with u_e is executed on the database, then perform the transaction t_a using the bindings of the matching between u and u_e.

Example 4.1 *The active rules described in Section 2 (page 5) can be easily expressed using the notation introduced above:*

r_1 : $-$`dep[dname=D]` $\Rightarrow$ $-$`emp[dname=D]`
r_2 : $+$`emp[name=N,dname=D]` $\Rightarrow$ $-$`emp[name=N,dname<>D]`
r_3 : $+$`emp[name=N,dname=D,sal>50k]` $\Rightarrow$ $+$`dep[dname=D,mgr=N]`

As we have said, one important point here is the temporal relationship between the execution of the components of a rule. The event and the action have a temporal decoupling under the deferred execution model, whereas, under the immediate execution model there is no temporal decoupling. In our approach, the semantics of an active database with respect to a transaction t is given in terms of the execution of a new transaction t' induced by t, and so it will be defined in Section 5 along with the definition of the rewriting technique.

4.2 Triggering of rules

We now describe how updates and active rules interact. We first introduce some preliminary notions.

Let D be the union of all the domains of the attributes in U and V be the set of variables. A *substitution* σ is a function from $D \cup V$ to $D \cup V$ that is the identity on constants. Then, the *matching* between atomic conditions is defined as follows: a ground atomic condition C_A (that is, an atomic condition without variables) over A *matches* with a generalized atomic conditions C'_A over the same attribute if: either (1) C'_A contains

variables and there is a substitution σ, called *binding*, such that $C_A = \sigma(C'_A)$, or (2) C'_A is ground and there is at least one tuple v over X, such that v satisfies both C_A and C'_A (that is, $Targ(C_A) \cap Targ(C'_A) \neq \emptyset$).

Let C_1 be a (ground) condition and C_2 be a generalized condition over the same set of attributes X. We say that C_1 matches with C_2 if, for each attribute $A \in X$ occurring in both of them, $C_1|_A$ matches with $C_2|_A$. If so, the composition of the bindings (if any) of the various atomic conditions forms the binding of C_1 and C_2. It is easy to show that if two condition match, then the matching is unique up to renaming of variables.

Definition 4.3 (Triggering) *Let $u = a[C]$ be an update and r be an active rule $u_e \Rightarrow t_a$, where $u_e = a_e[C_e]$. Then, we say that u* triggers *r if: (1) $a = a_e$ (that is, u and u_e perform the same type of operation on the same relation), and (2) C matches with C_e. If an update u triggers a rule $r : u_e \Rightarrow t_a$ and σ is their binding, then we say that u* induces *the sequence of updates $\sigma(t_a)$ because of the rule r.*

Note that because of the condition on the variables in an active rule (see Definition 4.1), a ground update always induces ground updates.

Example 4.2 *The update* `+emp[name=bill,dname=toy,sal=60k]` *triggers the following active rule:*

`+emp[name=N,dname=D,sal>50k]`$\Rightarrow$ `+dep[dname=D,mgr=N]`

because of the binding that associates `bill` *to the variable* `N` *and* `toy` *to the variable* `D` *(note that* `sal=60k` *matches with* `sal>50k`*). It follows that the update induces the update* `+dep[dname=toy,mgr=bill]`.

5 Transaction transformation

In this section we present the algorithms that transform a user defined transaction into an induced one that embodies the active rules behavior. We consider both the immediate and deferred cases.

5.1 Transaction transformations

In Figure 3 is reported the recursive algorithm that computes the reaction of a single update. In the algorithm, the symbol $\cdot$ denotes the concatenation operator of sequences.

Note that, in general, different outputs can be generated by this algorithm depending on the order in which triggered rules are selected in the first step of the while loop. Clearly the algorithm can be generalized in such a way that all the possible reactions of an update are generated. Moreover, according to several approaches described in the literature, the algorithm can me modified (first step of the while loop) in order to take into account a (partial) order on rules.

Unfortunately, the algorithm is not guaranteed to terminate over any possible input since some kind of recursion can occur in the active program. However, syntactical restriction can be given so that Algorithm REACTION is guaranteed to terminate. The result that follows is based on the construction of a special graph G_P describing the relationship between the rules of P. The construction of this graph is based on the notion of "unification" between updates that generalizes the notion of matching as follows. We

Algorithm REACTION
Input: An update u, an active program P, a set of constants $\mathcal{C}$, and
a set of attributes Z.
Output: A sequence μ^P of updates induced directly or indirectly by u and P.
begin
$\mu :=<>$;
Triggered $(u, P) := \{r \in P : r$ is triggered by $u\}$;
while Triggered (u, P) is not empty **do**
pick a rule $r_j : u_e \Rightarrow t_a$ from Triggered (u, P) and let σ be
the binding of u and u_e;
$t :=$ SPLIT$(\sigma(t_a), \mathcal{C}, Z)$;
for each u_i in t **do** $\mu := \mu \cdot <u_i> \cdot$ REACTION(u_i, P)
endwhile;
$\mu^P := \mu$
end.

Figure 3: Algorithm REACTION

say that two generalized updates u_1 and u_2 (possibly both containing variables) *unify* if there is a ground substitution σ (called *unifier*) such that $Targ(\sigma(u_1)) \cap Targ(\sigma(u_2)) \neq \emptyset$. Then, in the graph G_P the nodes represent the rules in P and there is an edge from a rule $r : u_e \Rightarrow t_a$ to a rule $r' : u'_e \Rightarrow t'_a$ if there is an update in t_a that unifies with u'_e.

Lemma 5.1 *If the graph G_P is acyclic then the algorithm* REACTION *is guaranteed to terminate over P and any update u_k.*

Proof. Algorithm REACTION performs a recursive call for each update $\sigma(u)$, where u in an update that occurs in the action part of a triggered rule r and σ is the matching that causes the triggering. This call causes in turn the triggering of a set of rules and, for each of these rules, a number of further recursive calls of the Algorithm REACTION. Let $r' : u'_e \Rightarrow t'_a$ be a rule triggered by $\sigma(u)$. By Definition 4.3, this means that there is a substitution σ' such that the targets of $\sigma(u)$ and $\sigma'(u'_e)$ have a nonempty intersection. Since we can assume that all the rules have different variables, we have that $\sigma \circ \sigma'$ is a unifier of r and r'. Therefore, two rules of P cause a recursive call if there is an edge from r to r' in G_P. Since G_P is acyclic, it follows that the number of recursive calls is always finite and so the algorithm terminates. □

Hereinafter, we consider only active program P such that the graph G_P is acyclic. Indeed, less restrictive conditions can be given to achieve termination. Also, the algorithm can be modified in order to take into account the presence of some kind of recursion. We have discussed these issues elsewhere [18].

We are now ready to present the notion of induced transactions.

Definition 5.1 (Induced transaction) *Let t be a user defined transaction, P be an active program, $\mathcal{C}$ be a set of constants that includes the constants occurring in t and the constants occurring in P, Z be a set of attributes that includes the attributes mentioned*

in t and the attributes mentioned in P, and $t' = u_1, \ldots, u_n$ be the output of Algorithm SPLIT *over t, $\mathcal{C}$ and Z. Then, consider the following transactions:*

- $t_I = u_1; \text{REACTION}(u_1, P, \mathcal{C}, Z); \ldots; u_n; \text{REACTION}(u_n, P, \mathcal{C}, Z)$,
- $t_D = u_1; \ldots; u_n; \text{REACTION}(u_1, P, \mathcal{C}, Z); \ldots; \text{REACTION}(u_n, P, \mathcal{C}, Z)$.

We say that t_I and t_D are induced *by t because of P, under the immediate and deferred modality respectively.*

Actually, in the following we will refer to induced transactions without making any explicit reference to the modality under which the transaction transformation has been computed since the various results hold independently from this aspect.

For induced transactions, the following property holds.

Lemma 5.2 *Let t be a user defined transaction and P be an active program. Then, any transaction induced by t because of P is in normal form and can be computed in polynomial time.*

Proof. By Definition 5.1, each update occurring in a transaction t' induced by t is split, using Algorithm SPLIT, either in the preprocessing step or during the execution of Algorithm REACTION, with respect to a set of constants and a set of attributes that include those occurring in t. Therefore, by Theorem 3.1, t' is in normal form. Let us now consider the complexity of the construction of t'. By the hypothesis on the acyclicity of the graph G_P, it easily follows that one execution of Algorithm REACTION requires, in the worst case, a number of recursive calls equals to $|P| + |P - 1| + \ldots + 1$, where $|P|$ denotes the number of rules in P, and so bounded by $|P|^2$. Moreover, each call of Algorithm REACTION involves one execution of Algorithm SPLIT, which requires polynomial time by Theorem 3.1, and a number of concatenation operations bounded by the maximum number of updates occurring in the action part of a rule of P. It follows that Algorithm REACTION requires polynomial time and, since this algorithm is used once for each update occurring in the original transaction, we have that an induced transaction can be computed in polynomial time. □

We point out that given a user defined transaction and an active program, we may have several different induced transactions, depending on the possible different outputs of Algorithm REACTION, and even if the number of those induced transactions is always finite, it may be very large. However, this number can be reduced by checking for instance when certain ones are "obviously" equivalent, e.g., when certain rules trivially commute. The problem of the efficient generation of induced transactions and their management is beyond the goal of this paper and has been addressed elsewhere [18].

5.2 Semantics of induced transaction

As we have said in Section 2, an induced update in an induced transaction is executed only if: (1) the inducing update has been effectively executed or (2) it has not been invalidated afterwards. Then, a new notion of effect of a transaction needs to be defined according to that. We call this new semantics the *active effect* of an induced transaction since it takes into account the relationship between "inducing" updates and the "induced" ones due

to the active rules. This relationship has to be known always and can be made explicit, during the generation of the induced transaction, in several ways, for instance, by means of a labeling technique, as described in [18].

Now, let u be an update and s be a database instance. We say that the effect of u is *visible* in s if:

- $u = +R[C]$ for some relation $R(X)$ and $Targ(C) \subseteq s(R(X))$, or
- $u = -R[C]$ for some relation $R(X)$ and $Targ(C) \cap s(R(X)) = \emptyset$.

Also, let $t = u_1, \ldots, u_n$ be a transaction and $1 \leq j \leq n$. We denote by $t|_j$ the transaction $u_1, \ldots, u_j$ composed by the first j components of t.

Definition 5.2 (Active effect) *The* active effect Eff_α *of an induced transaction* $t = u_1, \ldots, u_n$ *is a mapping* $\mathit{Eff}_\alpha(t)$ *from* $Inst(S)$ *to* $Inst(S)$*, recursively defined as follows, for* $1 \leq i < j \leq n$*.*

- $\mathit{Eff}_\alpha(t|_1)(s) = \mathit{Eff}(u_1)(s)$,
- $\mathit{Eff}_\alpha(t|_j)(s) = \begin{cases} \mathit{Eff}_\alpha(t|_{j-1})(s) & \text{if } u_j \text{ is induced by an update } u_i \\ & \text{in } t|_{j-1} \text{ and the effect of } u_i \text{ is} \\ & \text{not visible in } \mathit{Eff}_\alpha(t|_{j-1})(s), \\ \mathit{Eff}(u_j) \circ \mathit{Eff}_\alpha(t|_{j-1})(s) & \text{otherwise.} \end{cases}$

We are finally ready to define the semantics of a transaction with respect to an active database.

Definition 5.3 (Effect of a transaction in an active database) *A* potential effect *of a user transaction* t *in an active database* (s, P) *coincides with* $\mathit{Eff}_\alpha(t')(s)$*, where* t' *is a transaction induced by* t *because of* P*.*

6 Equivalence of active databases

Many interesting problems can be systematically studied in the formal framework we have defined. Among them: equivalence, optimization and confluence of active databases. In this section we shall consider equivalence and show that this property is decidable in polynomial time.

6.1 Equivalence of induced transactions

Transaction equivalence has been extensively investigated in the relational model [1, 13]. The major results of this study concern deciding whether two transactions are equivalent and transforming a transaction into an equivalent, but less expensive one. Unfortunately, these results cannot be directly used within our framework because of the different semantics defined for transactions. So, let us introduce a new definition of equivalence that refers to induced transactions.

Definition 6.1 (Equivalence of induced transactions) *Two induced transactions* t_1 *and* t_2 *are* equivalent *(denoted* $t_1 \sim_\alpha t_2$*) if it is the case that* $\mathit{Eff}_\alpha(t_1) = \mathit{Eff}_\alpha(t_2)$*.*

Algorithm SUMMARY
Input: An induced transaction $t = u_1, \ldots, u_n$.
Output: The summary $\Sigma_t = (\Sigma_t^+, \Sigma_t^-)$ of t.
begin
  $\Sigma_0 := (\emptyset, \emptyset)$;
  for each i, $1 \leq i \leq n$ **do**
    if (u_i is induced by u_j, $j < i$, and u_j is not embedded in Σ_{i-1})
    then $\Sigma_i := \Sigma_{i-1}$;
    else
      case u_i **of**
        $+R[C] : \Sigma_i := (\Sigma_{i-1}^+ \cup \{\langle R, C\rangle\}, \Sigma_{i-1}^- - \{\langle R, C\rangle\})$;
        $-R[C] : \Sigma_i := (\Sigma_{i-1}^+ - \{\langle R, C\rangle\}, \Sigma_{i-1}^- \cup \{\langle R, C\rangle\})$;
      endcase;
  $\Sigma_t := \Sigma_n$;
end.

Figure 4: Algorithm SUMMARY

We now present a simple method for testing equivalence of induced transaction. Actually, the method works for any transaction in normal form and is based on a representation of the behavior of a transaction that we call *summary*.

An *annotated condition* has the following syntax: $\langle R, C\rangle$, where R is a relation and C is a complex condition. Then, the *summary* of an induced transaction t is a pair $\Sigma_t = (\Sigma_t^+, \Sigma_t^-)$ of sets of annotated conditions generated by the SUMMARY algorithm reported in Figure 4.

In the algorithm we make use of the following notion: given a summary $\Sigma_t = (\Sigma_t^+, \Sigma_t^-)$, we say that an update u with condition C *is embedded* in Σ_t if either $u = +R[C]$ and $\langle R, C\rangle \in \Sigma_t^+$ or $u = -R[C]$ and $\langle R, C\rangle \in \Sigma_t^-$.

The summary $\Sigma_t = (\Sigma_t^+, \Sigma_t^-)$ describes, in a succinct way, the behavior of an induced transaction t. Specifically, let m and n be the cardinalities of Σ_t^+ and Σ_t^- respectively and consider a transaction $\widehat{t}$ defined as follows: for each $\langle R, C\rangle \in \Sigma_t^-$, $\widehat{t}$ has an update $-R[C]$ in one of its first n positions, and for each $\langle R, C\rangle \in \Sigma_t^+$, $\widehat{t}$ has an update $+R[C]$ in one of the positions from $n+1$ to $n+m$. Actually, since both Σ_t^+ and Σ_t^- are sets, we have several different way to build $\widehat{t}$; however, since the transactions we obtain have always the same deletions followed by the same insertions, their (non-active) effects are always the same. Thus, the order in which the updates occur in $\widehat{t}$ is immaterial and we can consider deterministic its construction. We now have the following.

Lemma 6.1 *Eff*$_\alpha(t) =$ *Eff*$(\widehat{t}\,)$.

Proof. The proof proceeds by induction on the length n of t. The basis ($n = 1$) is trivial since in this case it is easy to see that $t = \widehat{t}$ and so, by Definition 5.2, *Eff*$_\alpha(t) =$ *Eff*$(\widehat{t}\,)$. With regard to the induction step, assume that *Eff*$_\alpha(t|_{j-1}) =$ *Eff*$(\widehat{t}\,|_{j-1})$. Then, by Definition 5.2, for any instance s, the update u_j is not executed on *Eff*$_\alpha(t|_{j-1})(s)$ if and only if it is induced by an update u_i occurring in $t|_{j-1}$ that is not visible on *Eff*$_\alpha(t|_{j-1})(s)$. Under

this condition, by the inductive hypothesis u_i is not visible also on $\mathit{Eff}(\widehat{t}\,|_{j-1})(s)$ and, by construction, this simply implies that u_i does not occur in $\widehat{t}|_{j-1}$. But this means that u_i is not embedded in Σ_{j-1} and so, in the execution of the Algorithm SUMMARY, the annotated condition corresponding to u_j is not included in Σ_j and therefore u_j does not occur in $\widehat{t}|_j$. It follows that, for any instance s, u_j is effectively executed on $\mathit{Eff}_\alpha(t|_{j-1})(s)$ under the active effect semantics if and only if u_j occurs in $\widehat{t}|_j$, and so we have: $\mathit{Eff}_\alpha(t|_j) = \mathit{Eff}(\widehat{t}|_j)$; this completes the induction part. For $j = n$ we have $\mathit{Eff}_\alpha(t) = \mathit{Eff}(\widehat{t}\,)$. □

Theorem 6.1 *Let t_1 and t_2 be two induced transaction over the same set of constants and the same set of attributes. Then, we have that $t_1 \sim_\alpha t_2$ if and only if $\Sigma_{t_1} = \Sigma_{t_2}$.*

Proof. (If) Let $\Sigma_{t_1} = \Sigma_{t_2}$. We can assume, without loss of generality, that $\widehat{t_1} = \widehat{t_2}$, and so, by Lemma 6.1, $\mathit{Eff}_\alpha(t_1) = \mathit{Eff}(\widehat{t_1}) = \mathit{Eff}(\widehat{t_2}) = \mathit{Eff}_\alpha(t_2)$.
(Only if) Let $\mathit{Eff}_\alpha(t_1) = \mathit{Eff}_\alpha(t_2)$, s be an instance and, for some relation scheme $R(X)$, let v be a tuple in $s(R(X))$ such that $v \notin \mathit{Eff}_\alpha(t_1)(s)(R(X))$. Clearly, we have also that $v \notin \mathit{Eff}_\alpha(t_2)(s)(R(X))$. Now consider the transactions $\widehat{t_1}$ and $\widehat{t_2}$: by Lemma 6.1, $\mathit{Eff}_\alpha(t_1) = \mathit{Eff}(\widehat{t_1})$ and $\mathit{Eff}_\alpha(t_2) = \mathit{Eff}(\widehat{t_2})$, and so there is an update $u_1 = -R(C_1)$ in $\widehat{t_1}$ and an update $u_2 = -R(C_2)$ in $\widehat{t_2}$ such that $v \in \mathit{Targ}(C_1)$ and $v \in \mathit{Targ}(C_2)$. But t_1 and t_2 are induced transaction over the same set of constants and the same set of attributes and therefore their concatenation t_1t_2 is a transaction in normal form. Since $\mathit{Targ}(C_1)$ and $\mathit{Targ}(C_2)$ have a non empty intersection, by Lemma 3.1, it follows that $\mathit{Targ}(C_1) = \mathit{Targ}(C_2)$ and that $C_1 = C_2$. Thus, the annotated condition $\langle R, C_1\rangle$ occurs in both $\Sigma^-_{t_1}$ and $\Sigma^-_{t_2}$. The same considerations apply for insertions. It follows that, by construction, $\Sigma_{t_1} = \Sigma_{t_2}$. □

An interesting aspect to point out is that the notion of active effect of an induced transaction indeed generalizes the notion of effect of a user-defined transaction. This implies that the above characterization of equivalence also hold for ordinary transactions in passive environments.

6.2 Equivalence of user transactions

The notion of equivalence of user transactions, can be naturally extended in an active environment. Since we have seen that a transaction can potentially produces different results on an active database (depending on the different induced transactions that can be generated from it), we can assume that two transactions are equivalent when they are able to produce always the same results on any database instance.

Definition 6.2 (Equivalence of transactions in an active framework) *Two user transactions t_1 and t_2 are* equivalent with respect to an active program P *if for each transaction t'_1 induced by t_1 because of P there is a transaction t'_2 induced by t_2 because of P such that $t'_1 \sim_\alpha t'_2$, and vice versa.*

By the results of the above section, we can state the following result.

Theorem 6.2 *The equivalence of two user transactions in an active framework can be decided in polynomial time.*

Proof. Given a user transaction t, by Lemma 5.2 we can construct the transactions it induces in polynomial time. The number of transactions that can be induced by a single update because of an active program P is bounded by a constant that depends only on the size of P. Specifically, it is bounded by $k = (|P|)! \times (|P-1|)! \times \ldots \times 2$, where $|P|$ denotes the cardinality of P. Then, the number of transactions that can be induced by t is bounded by $k \times |t|$. By Definition 6.2, the equivalence of two user transactions t_1 and t_2 requires a test for the equivalence of each pair of transactions induced by t_1 and t_2 respectively, that is, a number of tests bounded by $k^2 \times |t_1| \times |t_2|$. Now, by Theorem 6.1, equivalence of two induced transactions requires: (1) the construction of their summaries by means of Algorithm SUMMARY, which requires time linear with respect of the length of the transaction (bounded by $|t| \times |P|^2$), and (2) the test for the equality of two summaries, which requires time proportional to $|t_1| \times |t_2| \times |P|^4$. It follows that the equivalence of t_1 and t_2 can be decided in polynomial time. □

6.3 Axiomatization of transaction equivalence

Before closing this section, we present two simple and intuitive axioms for proving equivalence of induced transactions, which provide much insight into the relationship between updates in an induced transaction. Since these axioms show how transactions can be manipulated without altering their overall effect, we call them *Manipulation Axioms*. We will also show that these axioms suggest a way to optimize efficiently transactions in an active environment.

In order to present those axioms, we need to introduce some preliminary notions. First, we use the notation $Induced(u)$ to denote the updates induced directly or indirectly by the update u in an induced transaction. Moreover, we say that two updates *collide* if they are over the same relation scheme and their conditions are identical. Finally, we introduce the notion of *validity* of an update in a transaction, which is the syntactical counterpart of the notion of visibility and is recursively defined as follow: an update u_i in an induced transaction t is *valid* if: (1) it is not induced, or (2) it is induced directly by an update u_j preceding u_i in t, and there is not a valid update u_k in t between u_j and u_i that collides with u_j.

Definition 6.3 (Manipulation Axioms) *The following axioms are called* Manipulation Axioms *and involve induced transactions in which μ_1 and μ_2 are (eventually empty) subtransactions:*

(MA1) $\mu_1; u_i; u_j; \mu_2 \approx_{\text{man}} \mu_1; u_j; u_i; \mu_2$ [Switching Axiom]
where: (1) u_i and u_j do not collide, (2) u_i is a valid update, and (3) u_j does not collide with any update u_k occurring in μ_1 such that $u_i \in Induced(u_k)$.

(MA2) $\mu_1; u_i; u_j; \mu_2 \approx_{\text{man}} \mu_1; u_j; \mu_2'$ [Merging Axiom]
where: (1) u_i and u_j collide, (2) u_j is a valid update, and (3) μ_2' equals μ_2 without the updates in $Induced(u_i) - Induced(u_j)$.

Intuitively, axiom (MA1) states that if the updates u_i and u_j do not collide, and u_j cannot invalidate the effect of the updates inducing u_i (if any), then they can be switched. Instead, Axiom (MA2) states that if u_i collides with u_j then their execution is equivalent to execute only u_j, provided that: (1) all the updates induced (directly or indirectly) by

u_i are not executed as well, and (2) the updates induced (directly or indirectly) by both u_i and u_j (if any) are however executed (this happens when u_j is induced by u_i and so $Induced(u_j) \subseteq Induced(u_i)$).

Lemma 6.2 *The manipulation axioms are sound, that is* $t_1 \approx_{\text{man}} t_2$ *implies* $t_1 \sim_\alpha t_2$.

Proof. Let us first consider MA1: since u_i and u_j do not collide, u_i cannot invalidate u_j if we execute the latter first. Moreover, by Definition 5.2, before the switching the effective execution of u_i depends on the updates occurring in μ_1 and if u_j does not collide with any update inducing u_i, we have that after the switching the execution of u_i still depends on the updates occurring in μ_1. Finally, before the switching the effective execution of u_j depends on the updates occurring in $\mu_1; u_i$. Therefore, if u_j is not induced by u_i, after the switching the effective execution of u_j still depends on the updates occurring in μ_1. On the other hand, if u_j is induced by u_i, no problem arises since u_i is valid and so u_j has to be executed anyway. It follows that, under the given conditions, the active effect of $\mu_1; u_i; u_j; \mu_2$ coincides with the active effect $\mu_1; u_j; u_i; \mu_2$.

Let us now consider MA2: since u_i and u_j collide and u_j is valid, we have that u_j invalidates u_i and therefore u_i can be deleted without altering the effect of the transaction $\mu_1; u_i; u_j; \mu_2$. Moreover, since u_i is invalidated by u_j, its effect is not visible during the execution of μ_2, and so all the updates induced directly or indirectly by u_i can be deleted from μ_2, except those induced also by u_j (in the case in which u_j is induced by u_i). Hence, under the given conditions, we have that the active effect of $\mu_1; u_i; u_j; \mu_2$ coincides with the active effect $\mu_1; u_j; \mu_2'$. □

Theorem 6.3 *MA is a sound and complete set of axioms for proving equivalence of induced transactions.*

Proof. By Lemma 6.2, MA is sound. To prove the completeness of these axioms, consider two induced transaction t_1 and t_2 such that $t_1 \sim_\alpha t_2$. Then, using the manipulation axioms, we can transform these transactions in two new transactions t_1' and t_2' where all the deletions are performed before all the insertions. This can be done in two steps: in the first one all the updates in the transaction that collide with another update are deleted and in the second one all the deletions are moved to the front of the transaction, before all the insertions. The first step can be performed by using the manipulation axioms as follows. Starting from the second update and iterating over the updates in the transaction, we move forward, using axiom MA1, the updates preceding the one currently under consideration that collide with it. Then, these updates are deleted using axiom MA2. It is easy to show that this work can always be done. After the first step, there is no pair of updates in the transaction that collide and so, in the second step, we can easily move the deletions to the front of the transaction using axiom MA1. Since the axioms are sound by Lemma 6.2, we have $t_1 \sim_\alpha t_1'$ and $t_2 \sim_\alpha t_2'$ and so $t_1' \sim_\alpha t_2'$. Also, since no pair of updates in both t_1' and t_2' collide, it easily follows that $t_1' = \widehat{t_1'}$ and $t_2' = \widehat{t_2'}$, and since $t_1' \sim_\alpha t_2'$, by Theorem 6.1 we have $\Sigma_{t_1'} = \Sigma_{t_2'}$. Then, by construction, we have that $\widehat{t_1'}$ and $\widehat{t_1'}$ coincide up to a permutation of the deletions and a permutation of the insertions. It follows that $\widehat{t_1'}$ can be transformed into $\widehat{t_2'}$ using repeatly axiom MA2. In sum we have: $t_1 \approx_{\text{man}} t_1' = \widehat{t_1'} \approx_{\text{man}} \widehat{t_2'} = t_2' \approx_{\text{man}} t_2$. □

7 Analysis of active rule processing

On the basis of the results on transaction equivalence, we derive in this section a number of results about important properties of active databases.

7.1 Confluence

Confluence is a strong property and some applications may actually need a weaker notion [2]. We then propose two notions of confluence. The former is weaker than the latter since refers to a specific transaction. However, this notion can be nicely characterized and turns out to be of practical importance.

Definition 7.1 (Weak confluence) *An active program P is* confluent with respect to a user transaction t *if all the transactions induced by t because of P are equivalent.*

Definition 7.2 (Strong confluence) *An active program P is* (strongly) confluent *if it is confluent with respect to any user transaction.*

The following result show that there is a practical method for testing weak confluence.

Theorem 7.1 *Weak confluence is decidable and can be tested in polynomial time.*

Proof. Given a user transaction t, by Lemma 5.2, we can construct the transactions induced by t in polynomial time. By Definition 7.1, the confluence of t with respect to P, requires to test for equivalence of each pair of transactions induced by t. The number of tests to be done is bounded by the square of the maximum number of transactions that can be induced by t, that is, by $k^2 \times |t|^2$, where k is a constant that depends only on the size of P (see the proof of Theorem 6.2). Since, by Theorem 6.1, testing for equivalence of two induced transactions requires polynomial time, it follows that the confluence of t with respect to P can be also performed in polynomial time. □

We now introduce another interesting notion of confluence that is independent of a specific transaction.

Let P be an active program, $\mathcal{C}$ be the set of constants occurring in P and $r : u_e \Rightarrow t_a$ be a rule of P. We denote by $\mathcal{U}_r$ the set of updates obtained from r as follows. For each atomic condition C_A in u_e involving a variable x, let $\mathcal{C}_A = dom(A) \cap \mathcal{C}$ and Ψ_A be the set of intervals $\psi = (c_1, c_2)$ such that: (1) $c_1, c_2 \in \mathcal{C}_A \cup \{-\infty, +\infty\}$, and (2) there is no constant $c \in \mathcal{C}_A$ such that $c \in (c_1, c_2)$. Note that, since $\mathcal{C}_A$ is finite, Ψ_A is actually a finite partition of $dom(A)$. Now, let K_ψ be a set of constants that contains one element (whichever it be) in every $(c_1, c_2) \in \Psi_A$, and let $K_A = K_\psi \cup \mathcal{C}_A$. Note that since Ψ_A is a finite partition of $dom(A)$, K_A is always finite. Then, the set $\mathcal{U}_r$ contains all the possible updates that can be obtained by applying to u_e a substitution that, for each attribute A occurring in u_e, maps variables of u_e occurring in C_A to constants in K_A.

Intuitively, the set $\mathcal{U}_r$ contains all the "representatives" of triggering updates for the rule r, and specify the different ways in which the rule r can be triggered by an update.

Definition 7.3 (Local confluence) *An active program P is* locally confluent on a rule $r \in P$ *if P is confluent with respect to any update in $\mathcal{U}_r$. An active program P is* locally confluent *if it is confluent on every rule in P.*

Note that, by Theorem 7.1, it follows that we can check for local confluence of an active program in polynomial time. The following result states that local confluence, although restrictive, is a desirable property for an active program.

Theorem 7.2 *If an active program is locally confluent then it is strongly confluent.*

Proof. Let t be a user transaction in normal form with respect to the set of constants occurring in t and P and the set of attributes mentioned in t and P. Note that this is not a restrictive hypothesis since, by Theorem 3.1, any transaction can be transformed in a transaction satisfying this property using Algorithm SPLIT. The proof proceeds by showing that, for each update u in t triggering a rule $r \in P$ and each sequence of updates μ induced by u, there is a mapping over constants θ such that: (1) $\theta(u) \in \mathcal{U}_r$, and (2) $\theta(u)$ induces a sequence of updates μ' such that $\mu = \theta(\mu')$. Specifically, this mapping is defined as follows: for each atomic condition C_A occurring in u, θ is the identity on the constants in K_A and maps each constant $c \notin K_A$ to the constant $c' \in K_A$ that belongs to the interval $\psi \in \Psi_A$ containing c. Clearly, $\theta(u) \in K_A$. Moreover, by Definition 4.3 (triggering) and by Algorithm SPLIT, it is easy to show, by induction on the number of step of Algorithm REACTION, that for each sequence of updates μ generated by this algorithm starting from u, the same algorithm is able to generate the sequence $\theta(\mu)$ starting from $\theta(u)$. But, by definition of local confluence, we have that all the sequences of updates induced by $\theta(u)$ are equivalent. It easily follows that all the sequences of updates induced by u are also equivalent. Thus, we have that P is confluent with respect to each sequence of updates induced by an update in t (if any). Now it can be easily shown that, given a transaction t, if there is a partition of t in sequences of adjacent updates $t = \mu_1, \ldots, \mu_k$ such that P is confluent with respect to each μ_i, $i = 1, \ldots, n$, then P is confluent with respect to t. Therefore, by Definition 5.1 of induced transaction, it follows that, independently from the modality, for any transaction t, P is confluent with respect to t and so P is strongly confluent. □

It is possible to show that, while local confluence implies strong confluence, the converse in not true in general even for weak confluence. That is, there are active programs that are not locally confluent but are confluent with respect to certain transactions.

The notion of local confluence gives us a sufficient condition for confluence that can be checked very efficiently. Let P be an active program and P_{conf} be the set of rules on which P is locally confluent. Note that this set can be derived one for all, at definition time. The following characterization of weak confluence simply requires, for each update in a transaction, one test of matching with the event part of the rules in P.

Corollary 7.1 *Let P be an active program and t be a user defined transaction. Then, P is confluent with respect to t if each update in t triggers only rules in P_{conf}.*

Proof. P_{conf} is indeed a strongly confluent program and therefore, by Theorem 7.2, t is confluent with respect to P_{conf} and so with respect to P. □

7.2 Optimization

A major objective of our research is to provide tools for optimizing induced transactions. This is particularly important since, in our approach, an optimization technique

for induced transactions yields a method for optimizing the overall activity of active rule processing.

According to [1], two types of optimization criteria for transactions can be considered. The first is related to syntactic aspects (e.g., length and complexity of updates) of a transaction, whereas the second is related to operational criteria such as the number of atomic updates performed by a transaction. Both criteria are formally investigated in this section.

Let us first introduce a preliminary notion. Let $\mathcal{P}$ be a partition of the tuple space, that is, a partition of the set of all tuples $v \in Tup(X)$ for every $R(X)$ in the scheme S: we say that a transaction t *is based on* $\mathcal{P}$ if, for each condition C occurring in an update of t, $Targ(C) \in \mathcal{P}$.

According to most implementations, we assume that a deletion operation is more complex than an insertion operation (denoted $u_i \leq u_j$). Clearly, this ordering may be invalid for certain implementation of the updates. However, changing the ordering does not affect the results that follow.

Definition 7.4 *A transaction $t = u_1; \ldots; u_n$ based on $\mathcal{P}$ is* syntactically optimal *if for every transaction t' based on $\mathcal{P}$ equivalent to t, $t' = u_1; \ldots; u_m$, where $m \geq n$, and there exists a permutation π of $\{1, \ldots, m\}$ such that $u_i \leq u_{\pi(i)}$, $1 \leq i \leq n$*

Given a transaction t, we denote by $Nop(t)$ a mapping from $Inst(S)$ to $\mathbb{N} \times \mathbb{N}$ that associates to an instance s the pair (i, d) where i is the number of tuples inserted by t into s, and d is the number of tuples deleted by t from s. Moreover, we denote by $\preceq$ the order relation on $\mathbb{N} \times \mathbb{N}$ defined as follows:

$$(i_1, d_1) \preceq (i_2, d_2) \text{ if and only if } i_1 + kd_1 \leq i_2 + kd_2$$

where k is the ratio between the cost of an insertion operation and the cost of a deletion operation. Intuitively, the order relation $\preceq$ takes into account the number of update operations, together with the preference attributed to insertions over deletions.

Definition 7.5 *A transaction t based on $\mathcal{P}$ is* operationally optimal *if for every transaction t' based on $\mathcal{P}$ equivalent to t, $Nop(t)(s) \preceq Nop(t')(s)$ for each instance $s \in Inst(S)$.*

Note that the above definitions do not refer to any possible pair of equivalent transactions but rather to transactions that are based on the same partition of the tuple space. This however is a more convenient form since it is possible to show that if the transactions are not based on the same partition, syntactically and operationally optimality cannot be attained simultaneously in general.

Definition 7.6 *A transaction t is* optimal *if it is operationally and syntactically optimal.*

Let us consider the manipulation axioms introduced in the previous section (Definition 6.3). It is easy to see that one application of the Merging Axiom yields a strictly simpler transaction, whereas the Switching Axiom does not affect the complexity of the translation, but is however useful in order to apply the Merging Axiom. This simple observation leads to a method for optimizing induced transactions. Intuitively, this method consists of applying a number of times the Switching Axiom followed by an application

Algorithm OPTIMIZE
Input: An induced transaction $t = u_1, \ldots, u_n$.
Output: A new transaction t_{opt}.
begin
    $i = 1$;
    $t_i := t$;
    repeat
        if (u_i collides with some update u_j ($j < i$) that precedes u_i in t_i)
        then t_{i+1} := the transaction obtained from t_i by deleting u_j and
                all the updates in $Induced(u_j) - Induced(u_i)$
        else $t_{i+1} := t_i$;
        $i = i + 1$;
    until (all the updates in t_i have been examined)
    $t_{\text{opt}} := t_i$;
end.

Figure 5: Algorithm OPTIMIZE

of the Merging Axiom, until no modification can be performed. The method can be effectively implemented in a very simple way by means of the algorithm in Figure 5 (recall that $Induced(u)$ denotes, in an induced transaction, all the updates induced directly or indirectly by the update u).

The following theorem confirms that the algorithm always terminates (in polynomial time) and produces an optimal transaction.

Theorem 7.3 *Let t be an induced transaction. Then, (1) Algorithm* OPTIMIZE *terminates over t and generates a transaction t_{opt} in polynomial time, (2) $t_{\text{opt}} \sim_\alpha t$, and (3) t_{opt} is optimal.*

Proof. (1) The algorithm simply involves an iteration over the updates in t and so requires, in the worst case, time linear in the length of the transaction. Note however that some step can involve a reduction of t and so, on the average, the execution of the algorithm is even more efficient.
(2) This part can be proved by showing that, at each step of the loop in the algorithm, when the new transaction (t_{i+1}) it is not equated to transaction of the previous step (t_i), it can be obtained from it through a number of applications of the Switching Axiom (to move u_j close to u_i) followed by one application of the Merging Axiom. To show this, we note that the algorithm eliminates collisions between updates as soon as they are encountered while iterating over the updates of the transaction. This implies that, at each step, in the sequence of updates μ_1 preceding the update u_i under consideration, there is no pair of updates that collide and so all the updates are valid. Therefore, at each step, we can freely use the Switching Axiom in μ_1 (see Definition 6.3) and, in particular, if there is an update u_j in μ_1 that collides with u_i, we can move u_j close to u_i. Then, we can apply the Merging Axiom thus obtaining exactly the transaction t_{i+1}. By Lemma 6.2, it follows that, for each i, $t_i \sim_\alpha t$ and so, at termination, $t_{\text{opt}} \sim_\alpha t$.

(3) First note that, as a consequence of what shown in part (2), in t_{opt} there is no pair of updates that collide. Now assume, by way of contradiction, that t_{opt} is not syntactically optimal and let t' be a transaction based on the same partition $\mathcal{P}$ and equivalent to t_{opt} that has less update operations than t_{opt} (according to the order $\leq$). This implies that, for each update u of t_{opt} with condition C that is visible on $\mathit{Eff}_\alpha(t_{\text{opt}})(s)$, for some instance s, there must be an update u' with condition C' in t' such that $\mathit{Targ}(C) = \mathit{Targ}(C')$. Therefore, since t_{opt} has more updates than t', there is at least an update u_x in t_{opt} that is not visible on s, and this is possible only if u_x is invalidated by another update in t_{opt}. But t_{opt} is in normal form as it is obtained by just deleting updates from a transaction in normal form. By Lemma 3.1, this implies that there are two updates in t_{opt} that collide — a contradiction.
Assume now, again by way of contradiction, t_{opt} is not operationally optimal and let t' be a transaction based on the same partition $\mathcal{P}$ and equivalent to t_{opt} such that $\mathit{Nop}(t')(s) \preceq \mathit{Nop}(t_{\text{opt}})(s)$ for some instance $s \in \mathit{Inst}(S)$. This implies that t_{opt} either perform two times the insertion/deletion of the same tuple or a tuple is first inserted (or deleted) and then deleted (inserted). But t_{opt} is in normal and so, by Lemma 3.1, this is possible only if there are two updates in t_{opt} that collide — again a contradiction. □

8 Conclusions

We have presented a formal technique that allows us to reduce, in several important cases, active rule processing to passive transaction execution. Specifically, user defined transactions are translated into new transactions that embody the expected rule semantics under the immediate and deferred execution modalities. We have shown that many problems are easier to understand and to investigate from this point of view, as they can be tackled in a formal setting that naturally extends an already established framework for relational transactions. In fact, it turns out that several important results derived for transactions in a passive environment can be taken across to an active one. Firstly, we have been able to formally investigate transaction equivalence in the framework of an active database. Secondly, results on transaction equivalence have been used to check for interesting and practically useful notions of confluence. Finally, optimization issues have been addressed.

We believe that this approach to active rule processing is suitable for further interesting investigations. From a practical point of view, we have studied efficient ways to generate and keep induced transactions, in the context of an implementation of the method on the top of a commercial relational DBMS [18]. From a theoretical point of view, we believe that the various results can be extended in several ways to take into account more general frameworks. Finally, the rewriting technique can be applied to other data models such as one based on objects [16].

References

[1] S. Abiteboul and V. Vianu. Equivalence and optimization of relational transactions. *Journal of the ACM*, 35(1):70–120, January 1988.

[2] A. Aiken, J. M. Hellerstein, and J. Widom. Static analysis techniques for predicting the behavior of active database rules. In *ACM Transaction on Database Systems*, 20(1):3–41, March 1995.

[3] C. Beeri and T. Milo. A model for active object-oriented database. In *Seventeenth International Conf. on Very Large Data Bases, Barcelona*, pages 337–349, 1991.

[4] S. Ceri , P. Fraternali, S. Paraboschi, and L. Tanca. Automatic generation of production rules for integrity maintenanace. In *ACM Transaction on Database Systems*, 19(3):367–422, September 1994.

[5] S. Ceri and R. Manthey. Chimera: a model and language for active DOOD Systems. In *Extending Information Systems Technology – Second International East-West Database Workshop, Klagenfurt*, pages 9–21, 1994.

[6] S. Ceri and J. Widom. Deriving production rules for constraint maintenance. In *Proc. of the Sixteenth International Conf. on Very Large Data Bases, Brisbane*, pages 566–577, 1990.

[7] S. Ceri and J. Widom. Deriving production rules for incremental view maintenance. In *Proc. of the Seventeenth International Conf. on Very Large Data Bases, Barcelona*, pages 577–589, 1991.

[8] P. Fraternali and L. Tanca. A structured approach for the definition of the semantics of active databases. In *ACM Transaction on Database Systems*, 20(4):414–471, December 1995.

[9] N. Gehani and H. V. Jagadish. ODE as an active database: constraints and triggers. In *Proc. of the Seventeenth International Conf. on Very Large Data Bases, Barcelona*, pages 327–336, 1991.

[10] G. Guerrini and D. Montesi. Design and implementation of Chimera's active rule language. *Data & Knowledge Engineering*, North-Holland, 1997. To appear.

[11] P. W. P. J. Grefen. Combining theory and practice in integrity control: a declarative approach to the specification of a transaction modification subsystem. In *Proc. of the Nineteenth International Conf. on Very Large Data Bases, Dublin*, pages 581–591, 1993.

[12] E. N. Hanson and J. Widom. Rule processing in active database systems. In *International Journal of Expert Systems*, 6(1):83–119, 1993.

[13] D. Karabeg and V. Vianu. Simplification rules and complete axiomatization for relational update transactions. In *Proc. of the ACM Transactions on Database Systems*, 16(3):439–475, September 1991.

[14] D.R. McCarthy and U. Dayal. The architecture of an active data base management system. In *Proc. of the ACM SIGMOD International Conf. on Management of Data*, pages 215–224, 1989.

[15] M. Morgenstern. Active databases as a paradigm for enhanced computing environments. In *Ninth International Conf. on Very Large Data Bases, Florence*, pages 34–42, 1983.

[16] D. Montesi and R. Torlone. A rewriting technique for implementing active object systems. In *Proc. of the International Symposium on Object-Oriented Methodologies and Systems, LNCS 858*, Springer-Verlag, pages 171–188, 1994.

[17] D. Montesi and R. Torlone. A rewriting technique for the analysis and the optimization of active databases. In *Proc. of the International Conference on Data Base Theory (ICDT'95), Praga, LNCS 893*, Springer–Verlag, pages 238–251, 1995.

[18] D. Montesi and R. Torlone. A transaction transformation approach to active rule processing. In *Proc. of the Eleventh International Conference on Data Engineering (ICDE'95), Taipei, Taiwan*, pages 109–116, 1995.

[19] D. Montesi and R. Torlone. A framework for the specification of active rule language semantics. In *Proc. of Fifth International Workshop on Database Programming Languages (DBPL5), Gubbio, Italia*, Workshop in Computing, Springer–Verlag, 1995.

[20] T. Sellis, C.C. Lin, and Raschid L. Implementing large production systems in a DBMS environment: concepts and algorithms. In *ACM SIGMOD International Conf. on Management of Data*, pages 404–412, 1988.

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

[22] M. Stonebraker, A. Jhingran, J. Goh, and S. Potamianos. On rules, procedures, caching, and views in data base systems. In *Proc. of the ACM SIGMOD International Conf. on Management of Data*, pages 281–290, 1990.

[23] J. Widom and S. J. Finkelstein. Set-Oriented production rules in relational databases systems. In *Proc. of the ACM SIGMOD International Conf. on Management of Data*, pages 259–270, 1990.