

Analysis and Optimization of Active Databases

Danilo Montesi

DSI - Università di Bologna, Mura Anteo Zamboni, 7 — 40127 Bologna, Italy
montesi@cs.unibo.it

Riccardo Torlone

DIA - Università Roma Tre, Via della Vasca Navale, 79 — 00146 Roma, Italy
torlone@dia.uniroma3.it

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 atomic 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.

Key words: 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 [8,9]), and for this reason, they are now largely used in modern database applications and have been extensively studied in the last years [3,6,7,15,19,22,30,31,34]. However, in the various approaches, active rule execution is generally specified only informally

or through a firing algorithm [10]. 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 [34].

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 an abstract active rule language, whose computational model is set-oriented (like in [34]) while other approaches are tuple-oriented [31]. The rule language chosen is a proper subset of the languages that can be found on real active database systems. Specifically, we just consider simple events and basic queries as rule conditions. In this way however we are able to provide effective results about important properties of active databases, which are difficult or impossible to test in general. Moreover, the language of reference can be easily translated into any practical active language: this makes the results we provide general and independent of the specific system used.

We consider the most popular execution models for active rules: immediate and deferred (or delayed) [14,22]. In the former model, rules are executed as soon as they are triggered. In the latter, rule execution is delayed to the end of the transaction. We 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, but usually they apply in a restrictive context [17] or are not formal [31]. Conversely, we believe that this formal and simple approach can improve the understanding of several active concepts and can 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. First, we wish to use a well known framework that is quite simple but have a formal setting and a solid transaction execution model. Second, we wish to take full advantage of the results already available

on transaction equivalence and optimization [1,20]. 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 [18]. It should be said that we will take into account a quite simple form of active rules (basic events and only projection/selection conditions) which is coherent with the transaction language. We stress on the fact that this choice take into account important applications and allows us to provide effective and efficient solutions for problems that are impossible or very difficult to solve in general [10].

Some preliminary results presented here, have been introduced in [25,26]. This paper extends these works in several directions. The original transaction language considered only equality predicates; now comparison predicates have been taken into account. The rule language has been extended by allowing a sequence of update operations in the action part instead of a single operation. Detailed proofs of equivalence, optimization and confluence are given. A new tool, the *summary* of an active transaction, has been introduced for testing equivalence in a more practical and efficient way. Finally, the cost model for optimization has been refined and a new notion of weak confluence has been provided.

The remainder of this paper is organized as follows. Section 2 discuss the related works. In Section 3, a detailed overview of the approach is presented by using several practical examples. In Section 4 we define the framework and derive a number of basic results. In Section 5, we introduce the rewriting transaction technique and describe the behavior of an induced transaction. 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 Related Works

There are many works considering several aspects of active databases (see [10,28] for an extensive treatment of this topic). Many papers investigate rule termination analysis [3,4,21,32], which is not addressed in this paper. Other papers study the semantics of active databases [29,13,35]. Specifically, in [29] active rule execution is modeled by means of sequences of database states. The expressive power of different active rule languages is then studied by reducing them to known query languages, based on relational algebra and Datalog, for

which the expressive power has already been investigated. In [13] the authors propose an approach for integrating active and deductive databases into a unified semantic framework. In [35] the semantics of active rule systems is investigated through the notion of production rule where there is no event part. The computational model is based on a unique fixpoint computational model that does not allow to generate contradictory updates. With this approach confluence is always guaranteed. In our paper, we provide a formal description of active behaviour that is based on a rewriting technique and is therefore basically different from these approaches. Moreover, we provide this description with the main goal of studying certain properties (equivalence, optimization and confluence) of active databases, which are not addressed in these works.

Some papers study confluence of active databases [3,4,21,33], but none of them also investigate equivalence and optimization. In [3] a static analysis technique is proposed in the context of the *Starburst Rule System* to test termination and deterministic behavior. This technique relies on a shallow comparison of the actions performed by one rule and the events and conditions of another rule. In [4] the authors analyze confluence and termination of Condition-Action rule execution using an algebraic framework. An algorithm is provided for determining when the action of one rule can affect the condition of another rule. This algorithm is used to provide sufficient conditions for confluence based on the commutativity of rule pairs. Our approach for testing confluence is different from both [3,4] since it is based on a method for testing equivalence of transactions in an active framework, which takes into account a set of active rules as a whole. Our rule model is sometimes more restrictive than others, but this allows us to provide effective and efficient characterizations. In [21] a term rewriting technique is adopted for describing active rule behaviour and known analysis techniques for termination and confluence of term rewriting systems are applied. The approach differs notably from ours and cannot be directly compared with it. Moreover confluence analysis is based on finding pairs of terms that can be rewritten in two different ways: this is similar in principle to the approach of [3,4]. Decidability properties of rule analysis in the context of object-oriented databases are investigated in [33]. The properties of confluence and of termination are studied using an approach based on a *typical database* containing all possible objects that could affect the outcome of rule processing. This approach is clearly infeasible in practical applications.

Very few paper investigate optimization of an active database [5,11,12]. In [5], a method is proposed for improving the efficiency of condition evaluation during rule processing in active database systems. The method replaces rule conditions with new conditions that are generally more efficient to evaluate, since they refer to incremental relations instead of entire database relations. In [11] rule analysis is used to detect opportunities for parallel evaluation of rules. In particular, a notion of compatibility extending that of commutativity defined

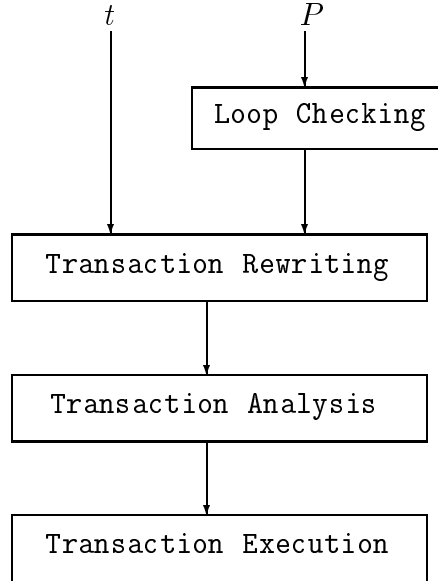


Fig. 1. Components of the approach

in [3], is used as the basis of a parallel execution strategy. In [12] the query optimizer used in the underlying object-oriented database system is adapted and extended for use with active rules. In particular, rule conditions are optimized by propagating constraints from event parameters into the query execution plan and by merging rules whose conditions contain the same or very similar goals. All these optimization methods are quite different from our approach that is based on a transaction rewriting technique. This technique tries to reduce the number of update operations needed to perform a user transaction in an active database system. Specifically, a transaction is first rewritten in a new transaction that embeds active rule behaviour and is then transformed into an equivalent transaction where redundant updates are deleted.

3 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. If the program has recursion we do not proceed with our approach. 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 [1,20]. Then, in the last step, according to the results of the analysis, one transaction is finally executed.

We point out two important aspects of this approach. First, it relies on a formal basis that allows us to derive solid results. Second, the rewriting and analysis/optimization steps can be done without accessing the underlying database, and therefore they can be performed very efficiently before the actual execution of the transaction.

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 [22]. The decoupled mode, in which triggered rules are executed in a separate transaction, can be also accommodated in the framework, but this approach will not be addressed in the paper to simplify the presentation. Thus, two different rewriting procedures will be given. Specifically, consider a user defined transaction as a sequence of updates: $t = u_1; \dots; u_n$. This transaction is transformed under the immediate modality into an *induced* one, which we will denote by t_I (where I denotes immediate modality):

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

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, which we will denote by t_D (where D denotes deferred modality):

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

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. Note that a specific interpretation of both immediate and deferred rule execution modality has been followed and several variants are possible. The approach however

is somehow parametric with respect to the modality and the interpretation chosen as it will be shown in the following.

To make the approach independent of any specific system, we will refer in this paper to generic languages for expressing transactions and active rules. With respect to the expressive power, it should be said that we will consider a quite simple transaction language involving update operations and fairly basic active rules. We believe however that we are able to capture in this way meaningful cases, often occurring in practical applications. Moreover and more important, this allows us to provide effective and efficient solutions for problems that are impossible or very difficult to solve in general. Thus, according to the approach described in Figure 1, the technique proposed in this paper can act as a filter over a real system, which works on certain cases and simply transfers to the underlying system cases which are outside of the framework. Moreover, our technique can be used in a CASE tool that helps database designers in the development of rules that are guaranteed to have the desired properties.

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 is 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
     With 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 $t1I$, 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 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 t_{2D} (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 an 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 usually happens that the updates in t induced (directly or indirectly) by u are not executed as well. This is called the “net effect” semantics of active databases [19]. Under this interpretation, we need to define a new model for transaction execution 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 t_1 under the deferred modality. According to the previous discussion, the rewriting process generates 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 the second one. It follows that the last two updates of the transaction t_{1D} , which are induced by such an update, must not be executed at run time. So, the rewriting of the transaction t_1 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.

4 Transactions and Active Rules

The notion of transaction we use in this paper extends a model for relational transactions introduced by Abiteboul and Vianu [1]. Informally, a transaction is viewed as a sequence of basic update operations (namely, insertions and deletions of tuples) viewed as a semantic atomic unit. More 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 comparison predicates to appear in update conditions.

With respect to the active component, we consider in this paper a simple form of active rules coherent with the transaction language. These rules however captures meaningful cases often occurring in practice and can be easily translated into most of the real systems. In particular, in [27] we have considered the core of several concrete active rule languages and have shown that it is possible to describe with our framework the main features of these languages.

4.1 Basics

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 [1]. 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 $\{R_1(X_1), \dots, R_n(X_n)\}$ 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 cartesian product 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 .

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 4.1 (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 $A\theta c$ if $v(A)\theta c$. Similarly, a tuple satisfies an atomic condition $(A \in (c_1, c_2))$ if $(c_1 < v(A) < c_2)$. 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 4.2 (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 $X = A_1 \dots A_n$ specifies a tuple if $C = \{A_1 = a_1, \dots, A_n = a_n\}$, where $a_i \in dom(A_i)$ for $i = 1, \dots, n$.

4.2 Transactions

Throughout the rest of the paper, we will always refer to a fixed database scheme $S = \{R_1(X_1), \dots, R_n(X_n)\}$ over a set of attributes U . A transaction is based on the basic update operations. The following insertion and deletion operations have, as usual, a set oriented behaviour [2].

Definition 4.3 (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 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 4.4 (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 4.5 (User transaction) A user transaction is a finite and non-empty sequence of updates. The effect of a transaction $t = u_1; \dots; u_n$ is the composition of the effects of the updates it contains, that is, is the mapping: $Eff(t) = Eff(u_1) \circ \dots \circ Eff(u_n)$.

Example 1 The SQL transactions described in Section 3 can be easily expressed using the notation introduced above. For instance, transaction t_1 at page 8, can be expressed as follows:

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

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

4.3 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 4.7 (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 $\text{Targ}(C_i) \cap \text{Targ}(C_j) = \emptyset$.*

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.

Proposition 4.1 *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.*

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 induce an equivalence relationship (denoted by \approx_{split}) between transactions and can be used to transform a non-normalized transaction into a transaction in normal form. Intuitively, the Split Axioms show: (1) how we can transform a transaction in an equivalent transactions that only contains 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 order to avoid a possible overlapping between a pair of conditions of the above forms (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 an attribute to a condition that does not mention it. As an example, axiom (SA2) specifies that an update condition of the form $\{A \neq c\}$ can be translated into a condition of the form $\{A \in (-\infty, c), A \in (c, +\infty)\}$ whereas axiom (SA6) specifies that an update condition of the form $\{A \in (c_1, c_2)\}$ can be translated into a condition of the

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

form $\{A \in (c_1, c), A = c, A \in (c, c_2)\}$ where c is a constant between c_1 and c_2 .

Definition 4.8 (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 :*

- (SA1) $-R[C] \approx_{\text{split}} -R[C|_{X-A} \cup \{C_A^{(1)}\}]; \dots; -R[C|_{X-A} \cup \{C_A^{(k)}\}]$
where $C|_A = \{C_A^{(1)}, \dots, C_A^{(k)}\}$ and $C_A^{(i)}$, for $i = 1, \dots, k$, 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 $\text{dom}(A)$ 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 $C|_A = \{A \in (c_1, c_2)\}$, $c \in \text{dom}(A)$ 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 $c \in \text{dom}(A)$.

Note how Split Axioms induce a relationship (denoted by \approx_{split}) on transactions. Note also that the axioms are defined only on delete operations since it turns out that insertion operations do not need to be transformed to enforce normal form (see Definition 4.3).

The following result can be easily proved.

Lemma 4.1 *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 $\text{Targ}(C) = \text{Targ}(C_1) \cup \text{Targ}(C_2)$ and so, by Definitions 4.4 and 4.5, we have that $\text{Eff}(u) = \text{Eff}(u_1) \circ \text{Eff}(u_2)$, that is, $u \sim u_1; u_2$. Similar considerations apply to the other axioms. \square

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.

<p>Algorithm SPLIT</p> <p>Input: A transaction t, a set of constants \mathcal{C} and a set of attributes Z;</p> <p>Output: A new transaction t_{split};</p> <p>begin</p> <p style="padding-left: 20px;">$i := 1$;</p> <p style="padding-left: 20px;">$t_i := t$;</p> <p style="padding-left: 20px;">while (there is an update u over $R_i(X_i)$ in t_i whose condition does not satisfies Property 4.1 for the sets of attributes X_i and Z and the set of constants \mathcal{C})</p> <p style="padding-left: 40px;">$t_{i+1} :=$ the transaction obtained from t_i by splitting u according to some split axiom;</p> <p style="padding-left: 20px;">$i := i + 1$;</p> <p style="padding-left: 20px;">endwhile;</p> <p style="padding-left: 20px;">$t_{\text{split}} := t_i - \{\text{updates with unsatisfiable conditions}\}$</p> <p>end.</p>
--

Fig. 2. Algorithm SPLIT

Property 4.1 *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 $\{A \in (c_1, c_2)\}$, where $c, c_1, c_2 \in \mathcal{C}$, and (2) if $C|_A = \{A \in (c_1, c_2)\}$, then there is no $c \in \text{dom}(A) \cap \mathcal{C}$ such that $c \in (c_1, c_2)$.*

The split algorithm that can be used to normalize transactions is reported in Figure 2.

Theorem 4.1 *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.*

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 4.1 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:

$$m = \binom{k}{2} + |\mathcal{C}|.$$

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

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 4.1, 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 4.1 (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 4.1, and this contradicts that t_{split} is the output of Algorithm SPLIT. \square

Example 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 $v = (A : 2)$ 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, after the deletion of the unsatisfiable update $-R_1[A \in (1, 2)]$, it 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.

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

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 satisfy Property 4.1 since the first update does not even mention attribute B . By applying Algorithm SPLIT we then obtain:

$$(1) \ t'_1 = -R_2[A = 1, B < 2]; -R_2[A = 1, B = 2]; -R_2[A = 1, B > 2]; \\ +R_2[A = 1, B = 2] \text{ (by axiom SA7)}$$

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

4.4 Active rules and programs

An active rule is represented 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 various parts of a rule are described by updates, according to the definitions of Section 4.2, with the only difference that variables can be used in place of constants. These variables serve to describe bindings that are passed from the event part to the rest of a rule.

Thus, let us fix a set of symbols called *variables*. We call *generalized condition* a condition in which variables can occur in place of constants. A *generalized update* is an update with a generalized condition. An *event specification* is a generalized update in which variables occur only in atomic conditions involving an equality predicate. Active rules are then defined as follows.

Definition 4.9 (Active rule) *An active rule has the form:*

$$u_e \Rightarrow t_a$$

where: (1) u_e is an event specification, and (2) t_a is a transaction composed by generalized updates involving only variables that also occur 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. Note that the condition part of an active rule is not explicitly represented, but it is incorporated in the action part. The intuitive semantics of a rule as above is then as follows: if an update u “matching” with u_e is effectively executed on the database, then perform the (conditional) updates in t_a using the bindings of the matching between u and u_e .

We note once more that we consider a limited notion of active rule but: (1) is coherent with the transaction language introduced early, (2) captures triggers often used in practice: those involving a sequence of “repairing” updates based

on a simple events consisting of an update (insertion or deletion) on a relational table, and (3) they can be translated into any actual real system. We stress on the fact that this choice allows us to provide effective solutions for problems that are untractable or even undecidable in general.

Definition 4.10 (Active database scheme) *An active program P is a set of active rules. An active database scheme is a pair (S, P) where S is a database scheme and P is an active program.*

In the following, in addition to a fixed scheme S , we assume that P is constant.

Example 3 *The active rules described in Section 3 (page 7) can be easily expressed using the notation introduced above:*

$$r_1 : -\text{dep}[\text{dname}=\text{D}] \Rightarrow -\text{emp}[\text{dname}=\text{D}]$$

$$r_2 : +\text{emp}[\text{name}=\text{N}, \text{dname}=\text{D}] \Rightarrow -\text{emp}[\text{name}=\text{N}, \text{dname} \neq \text{D}]$$

$$r_3 : +\text{emp}[\text{name}=\text{N}, \text{dname}=\text{D}, \text{sal} > 50\text{k}] \Rightarrow +\text{dep}[\text{dname}=\text{D}, \text{mgr}=\text{N}]$$

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 the next section, after with the definition of the rewriting technique.

5 Induced transactions

Throughout the rest of the paper, we always refer to a fixed active database scheme (S, P) . In this section we first present the way in which a user defined transaction is translated, using P , into an induced one that embodies the active rules behavior. We then define the semantics of an induced transaction.

5.1 Induced updates

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 mapping from variables in V to variables and constants in $D \cup V$. This notion can be naturally extended to generalized conditions [2].

Now, let C_1 be a ground condition (that is, a condition without variables) and C_2 be a generalized condition over the same set of attributes X . We say that C_1 matches with C_2 if there is a substitution σ such that $Targ(C_1) \cap Targ(\sigma(C_2)) \neq \emptyset$ (that is, there is at least one tuple over X that satisfies both C_1 and $\sigma(C_2)$). If so, σ is called the binding of C_1 and C_2 . It is easy to show that if two conditions match, then the binding is unique up to renaming of variables.

Definition 5.1 (Triggering and induced updates) *Let u be an update and r be an active rule $u_e \Rightarrow t_a$. Then, we say that u triggers r if: (1) u and u_e denote the same update operation on the same relation, and (2) the condition of u matches with the (generalized) condition of u_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 $\sigma(t_a)$.*

Note that because of the condition on the variables in an active rule (see Definition 4.9), a ground update always induces ground updates.

Example 4 *The update `+emp[name=bill,dname=toy,sal=60k]` triggers the 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 the atomic condition `sal=60k` matches with `sal>50k` since their targets have a non empty intersection. It follows that the update induces the update `+dep[dname=toy,mgr=bill]`).

We point out that the triggering of a rule is just a syntactical notion that will be used in the rewriting process described below. We recall that the behavior of a transaction in an active environment will be defined later in terms of the “active” effect of its translation on the underlying database.

5.2 Induced transactions

Figure 3 shows the recursive algorithm that computes the reaction of a single update. In the algorithm, the symbol \bullet 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 be modified (first step of the while loop) in order to take into account a (partial) order on

Algorithm REACTION

Input: An update u over an active database scheme (S, P) , a set of constants \mathcal{C} , and a set of attributes Z .

Output: A sequence μ_u of updates induced directly or indirectly by u .

begin

$\mu := \langle \rangle$;

Triggered(u) := $\{r \in P : r \text{ is triggered by } u\}$;

while Triggered(u) is not empty **do**

 pick a rule $r_j : u_e \Rightarrow t_a$ from Triggered(u) and let σ be
 the binding of u and u_e ;

$t := \text{SPLIT}(\sigma(t_a), \mathcal{C}, Z)$;

for each u_i in t **do** $\mu := \mu \cdot \langle u_i \rangle \cdot \text{REACTION}(u_i, \mathcal{C}, Z)$

endwhile;

$\mu_u := \mu$

end.

Fig. 3. Algorithm REACTION

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 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 $\text{Targ}(\sigma(u_1)) \cap \text{Targ}(\sigma(u_2)) \neq \emptyset$. 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 any update over (S, P) .*

Proof. Algorithm REACTION performs a recursive call for each update $\sigma(u)$, where u is 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 them, 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 5.1, 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. \square

We note that the graph G_P is strictly related with other notions of triggering graph defined in the literature for the analysis of rule properties and therefore the above lemma confirms similar earlier results (e.g., [3]).

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 [26,23]. We are now ready to present the notion of induced transactions.

Definition 5.2 (Induced transaction) *Let t be a user defined transaction over an active scheme (S, P) , \mathcal{C} be the union of the constants occurring in t and the constants occurring in P , Z be the union of the attributes mentioned in t and the attributes mentioned in P , and $t' = u_1; \dots; u_n$ be the output of Algorithm SPLIT over t , \mathcal{C} and Z . Then, consider the following transactions:*

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

We say that t_I and t_D are induced by t in (S, P) under the immediate and deferred modality respectively.

It is easy to see from the above definition that the deferred semantics is deferred in the first step and then immediate leading to a deferred-immediate semantics. The reason for this hybrid semantics that we call deferred is twofold. First, this kind of behavior allows intermediate updates to violate integrity constraint and regains integrity at the end of a transaction (first step is deferred), hoping that some subsequent update within the transaction will repair the violation. If this is not the case, the violation is repaired through an immediate semantics (second step immediate). Thus a deferred-immediate semantics is appropriate and often used [17]. Second, it allows us to consider only one REACTION algorithm simplifying the presentation and results. However a deferred-deferred semantics can also be modeled in our framework by introducing a new REACTION Algorithm and by modifying accordingly the notion of deferred modality in Definition 5.2. Intuitively, the new algorithm would require to iteratively append induced updates to the end of the user transaction in the order in which they are triggered, until all triggered rules have been considered. Clearly, at each step, not only the rules triggered by the original updates but also those triggered by the induced updates computed in the preceding steps have to be taken into account.

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 *Any transaction induced by a user defined transaction t in an active scheme (S, P) is in normal form and can be computed in polynomial time.*

Proof. By Definition 5.2, 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 4.1, 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 $|P| + |P - 1| + \dots + 1$, where $|P|$ denotes the number of rules in P , and so it is bounded by $|P|^2$. Moreover, each call of Algorithm REACTION involves one execution of Algorithm SPLIT, which requires polynomial time by Theorem 4.1, 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. \square

Example 5 *Let (S, P) be an active scheme where: $S = \{R_1(A), R_2(AB)\}$ with domains over the integers and P contains the following active rules.*

$$r_1 : -R_1[A = x] \Rightarrow -R_2[A = x]$$

$$r_2 : +R_2[A = x, B = y] \Rightarrow -R_2[A = x, B \neq y]$$

$$r_3 : +R_2[A = x, B > 2] \Rightarrow +R_1[A = x]$$

Consider now the transaction:

$$t = -R_1[A = 1]; +R_2[A = 1, B = 3]$$

Note that this transaction is in normal form.

According to Definition 5.2, to compute a transaction induced by t , we need to apply Algorithm REACTION to $u_1 = -R_1[A = 1]$ and to $u_2 = +R_2[A = 1, B = 3]$ with $Z = \{A, B\}$ and $\mathcal{C} = \{1, 2, 3\}$. The application of Algorithm

REACTION to u_1 proceeds as follows:

- initially we have $\mu = \langle \rangle$ and $\text{Triggered}(u_1) = \{r_1\}$;
- in the while loop we pick rule r_1 : this rule induces the update $-R_2[A = 1]$;
- by applying Algorithm SPLIT to $-R_2[A = 1]$, \mathcal{C} and Z , we obtain:
 - at the first step, by using axiom SA7 for adding the attribute $B \in Z$, we have:

$$t_1 = -R_2[A = 1, B < 2]; -R_2[A = 1, B = 2]; -R_2[A = 1, B > 2];$$

- at the second step, by using axiom SA6 for including the constant $3 \in \mathcal{C}$, we have:

$$t_2 = -R_2[A = 1, B < 2]; -R_2[A = 1, B = 2]; -R_2[A = 1, B \in (2, 3)]; \\ -R_2[A = 1, B = 3]; -R_2[A = 1, B > 3]$$

- finally, the third unsatisfiable update is deleted and Algorithm SPLIT stops;
- since no recursive triggering occurs in the last step of the while loop we obtain:

$$\mu = -R_2[A = 1, B < 2]; -R_2[A = 1, B = 2]; \\ -R_2[A = 1, B = 3]; -R_2[A = 1, B > 3]$$

- algorithm REACTION stops and returns: $\mu_{u_1} = \mu$.

The application of Algorithm REACTION to the update $u_2 = +R_2[A = 1, B = 3]$ proceed similarly. Note that u_2 triggers two rules: r_2 and r_3 . Then, the algorithm REACTION produces two different outputs depending on the order in which these rules are selected. Specifically, by considering rule r_2 before r_3 , and proceeding as described above we obtain:

$$\mu_{u_2} = -R_2[A = 1, B < 2]; -R_2[A = 1, B = 2]; \\ -R_2[A = 1, B > 3]; +R_1[A = 1]$$

The other possible result μ'_{u_2} has the same updates of μ_{u_2} but in a different order: the update $+R_1[A = 1]$ (which is induced through rule r_3) is in the first position and the other updates follow it in the same order of μ_{u_2} .

Then, under the immediate modality, a transaction induced by t is:

$$\begin{aligned}
t_I = & -R_1[A = 1]_{(1)}; -R_2[A = 1, B < 2]_{(1a)}; -R_2[A = 1, B = 2]_{(1b)}; \\
& -R_2[A = 1, B = 3]_{(1c)}; -R_2[A = 1, B > 3]_{(1d)}; \\
& +R_2[A = 1, B = 3]_{(2)}; -R_2[A = 1, B < 2]_{(2a)}; -R_2[A = 1, B = 2]_{(2b)}; \\
& -R_2[A = 1, B > 3]_{(2c)}; +R_1[A = 1]_{(2d)}
\end{aligned}$$

In this induced transaction we have that the updates (1a), (1b), (1c), and (1d) are induced by the update (1), whereas the updates (2a), (2b), (2c), and (2d) are induced by the update (2).

The other possible transaction induced by t is:

$$\begin{aligned}
t'_I = & -R_1[A = 1]_{(1)}; -R_2[A = 1, B < 2]_{(1a)}; -R_2[A = 1, B = 2]_{(1b)}; \\
& -R_2[A = 1, B = 3]_{(1c)}; -R_2[A = 1, B > 3]_{(1d)}; \\
& +R_2[A = 1, B = 3]_{(2)}; +R_1[A = 1]_{(2d)}; -R_2[A = 1, B < 2]_{(2a)}; \\
& -R_2[A = 1, B = 2]_{(2b)}; -R_2[A = 1, B > 3]_{(2c)}
\end{aligned}$$

We point out that given a user defined transaction and an active scheme, 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 instances 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 [26].

5.3 Semantics of induced transaction

As we have said in Section 3, 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 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 [26].

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

Definition 5.3 (Active effect) *The active effect Eff_α of an induced transaction $t = u_1; \dots; u_n$ is a mapping $Eff_\alpha(t)$ from $Inst(S)$ to $Inst(S)$, recursively defined as follows, for $1 \leq i < j \leq n$.*

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

We are finally ready to define the semantics of a transaction in an active framework.

Definition 5.4 (Effect of a user transaction in an active scheme) *A possible effect of a user transaction t in a database s over an active scheme (S, P) coincides with $Eff_\alpha(t')(s)$, where t' is a transaction induced by t in (S, P) .*

We are now ready to develop equivalence results in the next section.

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 how this property can be verified.

6.1 Equivalence of induced transactions

Transaction equivalence has been extensively investigated in the relational model [1,20]. The major results of this study concern deciding whether two transactions are equivalent and transforming a transaction into an equivalent,

<p>Algorithm SUMMARY</p> <p>Input: An induced transaction $t = u_1, \dots, u_n$.</p> <p>Output: The summary $\Sigma_t = (\Sigma_t^+, \Sigma_t^-)$ of t.</p> <p>begin</p> <p style="padding-left: 20px;">$\Sigma_0 := (\emptyset, \emptyset)$;</p> <p style="padding-left: 20px;">for each $i, 1 \leq i \leq n$ do</p> <p style="padding-left: 40px;">if (u_i is induced by u_j and u_j is not embedded in Σ_{i-1})</p> <p style="padding-left: 40px;">then $\Sigma_i := \Sigma_{i-1}$;</p> <p style="padding-left: 40px;">else</p> <p style="padding-left: 60px;">case u_i of</p> <p style="padding-left: 80px;">$+R[C] : \Sigma_i := (\Sigma_{i-1}^+ \cup \{\langle R, C \rangle\}, \Sigma_{i-1}^- - \{\langle R, C \rangle\})$;</p> <p style="padding-left: 80px;">$-R[C] : \Sigma_i := (\Sigma_{i-1}^+ - \{\langle R, C \rangle\}, \Sigma_{i-1}^- \cup \{\langle R, C \rangle\})$;</p> <p style="padding-left: 60px;">endcase;</p> <p style="padding-left: 20px;">$\Sigma_t := \Sigma_n$;</p> <p>end.</p>
--

Fig. 4. Algorithm SUMMARY

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 $Eff_\alpha(t_1) = Eff_\alpha(t_2)$.*

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*. This is similar to deltas provided in *Heraclitus* [16]. However, while in *Heraclitus* they are used to implement the execution model, here they are seen as a way to denote the behavior of a transaction and are used to test equivalence.

An *annotated condition* has the following syntax: $\langle R, C \rangle$, where R is a relation and C is a complex condition. 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 is *embedded* in Σ_t if either $u = +R[C]$ and $\langle R, C \rangle \in \Sigma_t^+$ or $u = -R[C]$ and $\langle R, C \rangle \in \Sigma_t^-$.

Example 6 *Consider the induced transaction t_I of Example 5. By applying*

Algorithm SUMMARY to this transaction we obtain the following summary:

$$\begin{aligned}\Sigma_{t_I} = (\Sigma_{t_I}^- = \{ \langle R_2, [A = 1, B < 2] \rangle, \langle R_2, [A = 1, B = 2] \rangle, \\ \langle R_2, [A = 1, B > 3] \rangle \}, \\ \Sigma_{t_I}^+ = \{ \langle R_2, [A = 1, B = 3] \rangle, \langle R_1, [A = 1] \rangle \})\end{aligned}$$

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

Lemma 6.1 $Eff_\alpha(t) = Eff(\hat{t})$.

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 $t = \hat{t}$ and so, by Definition 5.3, $Eff_\alpha(t) = Eff(\hat{t})$. With regard to the induction step, let $t = u_1; \dots; u_k; u$ ($k \geq 1$) be an induced transaction and assume that the claim holds for $t' = u_1; \dots; u_k$, that is, $Eff_\alpha(t') = Eff(\hat{t}')$. Then, by Definition 5.3, for any instance s , two different cases can arise in the execution of the last update of t over s : (1) u is induced by an update u' occurring in t' , and (2) u is not induced. In case (1) we have two subcases: (a) u' is visible on $Eff_\alpha(t')(s)$ and so $Eff_\alpha(t)(s) = Eff(u) \circ Eff_\alpha(t')(s)$, and (b) u' is not visible on $Eff_\alpha(t')(s)$ and so $Eff_\alpha(t)(s) = Eff_\alpha(t')(s)$. In case (2) we simply have that $Eff_\alpha(t)(s) = Eff(u) \circ Eff_\alpha(t')(s)$. Consider case (1a): by the inductive hypothesis, we have that, for any instance s , u' is also visible on $Eff(\hat{t}')(s)$; this implies that u' occurs in \hat{t}' and, by construction, that u' is embedded in $\Sigma_{t'}$. It follows that, after the k -th step of the execution of the Algorithm SUMMARY over t , u' is embedded in $\Sigma_k = \Sigma_{t'}$ and so, in the $(k + 1)$ -th step, the annotated condition corresponding to u is added to Σ_{k+1} and then the algorithm halts. Hence, \hat{t} equals \hat{t}' augmented with u , and so we have that, for any instance s , $Eff_\alpha(t)(s) = Eff(u) \circ Eff_\alpha(t')(s) = Eff(u) \circ Eff(\hat{t}')(s) = Eff(\hat{t})(s)$. Similar arguments apply to the other cases. It follows that, for any instance s , $Eff_\alpha(t)(s) = Eff(\hat{t})(s)$. \square

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 $\hat{t}_1 = \hat{t}_2$, and so, by Lemma 6.1, $Eff_\alpha(t_1) = Eff(\hat{t}_1) = Eff(\hat{t}_2) = Eff_\alpha(t_2)$.

(Only if) Let $Eff_\alpha(t_1) = Eff_\alpha(t_2)$, s be an instance and, for some relation scheme $R(X)$, let v be a tuple in $s(R(X))$ such that $v \notin Eff_\alpha(t_1)(s)(R(X))$. Clearly, we have also that $v \notin Eff_\alpha(t_2)(s)(R(X))$. Now consider the transactions \hat{t}_1 and \hat{t}_2 : by Lemma 6.1, $Eff_\alpha(t_1) = Eff(\hat{t}_1)$ and $Eff_\alpha(t_2) = Eff(\hat{t}_2)$, and so there is an update $u_1 = -R(C_1)$ in \hat{t}_1 and an update $u_2 = -R(C_2)$ in \hat{t}_2 such that $v \in Targ(C_1)$ and $v \in Targ(C_2)$. Nevertheless, t_1 and t_2 are induced transaction over the same set of constants and the same set of attributes and so \hat{t}_1 and \hat{t}_2 . This implies that their concatenation $\hat{t}_1; \hat{t}_2$ is a transaction in normal form and, since $Targ(C_1)$ and $Targ(C_2)$ have a non empty intersection, by Proposition 4.1, it follows that $Targ(C_1) = Targ(C_2)$ 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 arguments apply to insertions. It follows that, by construction, $\Sigma_{t_1} = \Sigma_{t_2}$. \square

Example 7 Consider the induced transaction t_I and t'_I of Example 5. By applying Algorithm SUMMARY to them we obtain, in both cases, the summary reported in example 6. This shows that these transactions are equivalent in the given active scheme.

An interesting aspect to point out is that the notion of active effect of an induced transaction generalizes the notion of effect of a user-defined transaction. This implies that the above characterization of equivalence also holds 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 produce 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 always produce the same results on any database instance.

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

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

Theorem 6.2 The equivalence of two user transactions in an active scheme is decidable and if the rules are ordered it can be tested in polynomial time.

Proof. Given a user transaction t , by Lemma 5.2 we can construct the transactions it induces in polynomial time. Since we have assumed a fixed active scheme (S, P) , the number of transactions that can be induced by a single update in an active scheme is bounded by a constant that depends only on the size of P . Specifically, it is bounded by $k = (|P|)! \times (|P - 1|)! \times \dots \times 2$, where $|P|$ denotes the cardinality of P . Now, in the most general case, the number of transactions that can be induced by t is exponential, but it is easy to see that this number decrease dramatically if we impose a partial order on the rules and if the order is total, the number 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. 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, and (2) the test for the equality of two summaries, which requires time proportional to the length of the transactions. \square

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 [3]. We then propose two notions of confluence. The first is weaker than the second since it 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 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 if the rules are ordered it 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 transactions that can be induced by t is exponential in general, but if we impose a total order on the rules, this number is bounded by value that depends linearly by the length of t (see proof of Theorem 6.2). Since, by Theorem 6.1, testing for equivalence of two induced transactions can be decided (in polynomial time), it follows that the confluence of t with respect to P can also be decided, and if the rules are ordered, the test can be performed in polynomial time. \square

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 $\mathcal{C}_A = \text{dom}(A) \cap \mathcal{C}$ 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 $\text{dom}(A) - \mathcal{C}_A$. Now, let K_{ψ_A} be a set of constants that contains any one element in the range (c_1, c_2) , and let $K_A = \bigcup_{\psi \in \Psi_A} K_{\psi_A}$. Note that since Ψ_A is finite, K_A is also finite. Then, the set \mathcal{U}_r contains all the possible updates that can be formed from u_e by substituting the variables occurring in the atomic condition C_A of u_e with constants in K_A .

Intuitively, the set \mathcal{U}_r contains all the “representatives” of triggering updates for the rule r , and thus specifies 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 locally 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.

Example 8 *Consider again the active program P of Example 5 reported here for convenience.*

$$\begin{aligned}
r_1 &: -R_1[A = x] \Rightarrow -R_2[A = x] \\
r_2 &: +R_2[A = x, B = y] \Rightarrow -R_2[A = x, B \neq y] \\
r_3 &: +R_2[A = x, B > 2] \Rightarrow +R_1[A = x]
\end{aligned}$$

Consider rule r_2 . We have $\mathcal{C}_A = \emptyset$ since no constant occur in P in the atomic conditions involving the attribute A . We then have $\Psi_A = \{(-\infty, +\infty)\}$ and so we can choose $K_{\psi_A} = K_A = \{0\}$. For the attribute B we have: $\mathcal{C}_B = \{2\}$ (a constant occurring in r_3), $\Psi_B = \{(-\infty, 2)(2, +\infty)\}$, and we can choose, for instance, $K_{\psi_B} = \{0, 5\}$, and so $K_B = \{0, 2, 5\}$. Thus, we have:

$$\mathcal{U}_{r_2} = \{+R_2[A = 0, B = 0], +R_2[A = 0, B = 2], +R_2[A = 0, B = 5]\}$$

By using the technique described in the previous section, it is possible to show that each update in \mathcal{U}_{r_2} induces equivalent transactions. It follows that P is confluent with respect to each of them, and so we can conclude that it is locally confluent on r_2 . Actually, P is locally confluent also on r_1 and r_3 : as shown below, this implies that P is strongly confluent.

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 4.1, any transaction can be transformed in an equivalent 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 5.1 (triggering) and by Algorithm SPLIT, it is easy to show, by induction on the number of steps 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 shown that, given a transaction t , if there is a partition of t in sequences of adjacent updates $t = \mu_1, \dots, \mu_k$ such that P is confluent with respect to each μ_i , $i = 1, \dots, k$, then P is confluent with respect to t . Therefore, by Definition 5.2 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. \square

It is possible to show that, while local confluence implies strong confluence that in turn implies weak confluence, 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 in P on which P is locally confluent. Note that this set can be derived once and 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 . \square

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.

Following [1], there are two types of optimization criteria for transactions. 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 [1], we assume that a tuple deletion operation (deletion of one tuple) is more complex than a tuple insertion operation (insertion of one tuple) (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 $t = u_1; \dots; u_n$ based on \mathcal{P} is syntactically optimal if for every transaction t' based on \mathcal{P} equivalent to t , $t' = u_1; \dots; u_m$, where $m \geq n$, and there exists a permutation π of $\{1, \dots, m\}$ 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, syntactical and operational optimality cannot be attained simultaneously in general.

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

In Figure 5 we present a simple algorithm for the optimization of induced transactions. 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) $t_{opt} \sim_{\alpha} t$, 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.

```

Algorithm OPTIMIZE
Input: An induced transaction  $t = u_1, \dots, u_n$ .
Output: A new transaction  $t_{\text{opt}}$ .
begin
   $i = 1$ ;
   $t_i := t$ ;
  repeat
    case of
      ( $u_i$  is related with some update  $u_j$  that precedes  $u_i$  in  $t_i$ 
        and is not induced) or
      ( $u_i$  is related with some update  $u_j$  that precedes  $u_i$  in  $t_i$ 
        and is induced by an update  $u_k$  that precedes  $u_i$  in  $t_i$ ):
         $t_{i+1} :=$  the transaction obtained by deleting  $u_j$  from  $t_i$ ;
      ( $u_i$  is induced by an update  $u_k$  that does not precede  $u_i$  in  $t_i$ ):
         $t_{i+1} :=$  the transaction obtained by deleting  $u_i$  from  $t_i$ ;
      otherwise  $t_{i+1} := t_i$ ;
    endcase
     $i = i + 1$ ;
  until (all the updates in  $t_i$  have been examined)
   $t_{\text{opt}} := t_i$ ;
end.

```

Fig. 5. Algorithm OPTIMIZE

(2) This part can be proven by showing that t and t_{opt} have the same summaries. In turn, since t_{opt} is obtained by simply deleting updates from t , this can be proved by shown that for each update $u = +R[C]$ ($u = -R[C]$) in t that does not occur in t_{opt} , it is the case that $\langle R, C \rangle \notin \Sigma_t^+$ ($\langle R, C \rangle \notin \Sigma_t^-$). This claim can be shown by induction on the length of t . The basis of the induction is trivial and the induction step follows by construction of Σ_t (Algorithm SUMMARY in Figure 4) and t_{opt} (Algorithm OPTIMIZE in Figure 5)

(3) First note that, by construction, in t_{opt} there is no pair of related updates. 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 $\text{Eff}_\alpha(t_{\text{opt}})(s)$, for some instance s , there must be an update u' with condition C' in t' such that $\text{Targ}(C) = \text{Targ}(C')$. 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 Proposition 4.1, this implies that there are two updates in t_{opt} that are related — 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 $\text{Nop}(t')(s) \preceq \text{Nop}(t_{\text{opt}})(s)$ for some instance $s \in \text{Inst}(S)$. 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 Proposition 4.1, this is possible only if there are two updates in t_{opt} that are related — again a contradiction. \square

Example 9 *Consider the induced transaction:*

$$\begin{aligned} t = & -R_1[A = 1]_{(1)}; +R_2[A = 1, B = 3]_{(2)}; +R_1[A = 0]_{(3)}; \\ & -R_2[A = 1, B = 2]_{(1a)}; -R_2[A = 1, B = 3]_{(1b)}; -R_2[A = 1, B < 2]_{(1c)}; \\ & -R_2[A = 1, B < 2]_{(2a)}; +R_1[A = 1]_{(2aa)}; \\ & -R_2[A = 1, B = 2]_{(3a)}; +R_1[A = 1]_{(3b)} \end{aligned}$$

in which the updates (1a), (1b), and (1c) are induced by the update (1), the update (2a) is induced by the update (2), the update (2aa) is induced by the update (2a), and the updates (3a), and (3b) are induced by the update (3). Note that the transaction is in normal form.

The application of Algorithm OPTIMIZE to t proceeds as follows:

- The first update that satisfies one of the conditions of the case statement in Algorithm OPTIMIZE is (1b) that is related with update (2). This update is induced by update (1) that precedes (1b) and therefore update (2) is deleted from t .
- Then, we find update (2a) that is induced by (2). This update has been deleted in the previous step and therefore update (2a) is also deleted. For the same reason, update (2aa) is deleted from t as well. At this point we have:

$$\begin{aligned} t_i = & -R_1[A = 1]_{(1)}; +R_1[A = 0]_{(3)}; -R_2[A = 1, B = 2]_{(1a)}; \\ & -R_2[A = 1, B = 3]_{(1b)}; -R_2[A = 1, B < 2]_{(1c)}; \\ & -R_2[A = 1, B = 2]_{(3a)}; +R_1[A = 1]_{(3b)} \end{aligned}$$

- We then encounter update (3a) that coincides with update (1a) and is induced by (3) that still occurs in t : update (1a) is then deleted.
- Finally, we have update (3b) that is related with update (1). Since update (3b) is also induced by (3) that occurs in t , update (1) is deleted and we obtain:

$$\begin{aligned} t_2 = & +R_1[A = 0]_{(3)}; -R_2[A = 1, B = 3]_{(1b)}; -R_2[A = 1, B < 2]_{(1c)}; \\ & -R_2[A = 1, B = 2]_{(3a)}; +R_1[A = 1]_{(3b)} \end{aligned}$$

The final transaction is equivalent to t and optimal.

8 Conclusions

We have presented a formal technique that allows us to reduce, in certain restricted but meaningful 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 several 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 results derived for transactions in a passive environment can be taken across to an active one. First, we have been able to formally investigate transaction equivalence in the framework of an active database. Second, results on transaction equivalence have been used to check for interesting and practically useful notions of confluence. Finally, optimization issues have been addressed. It should be said that we consider an abstract active rule language with limited expressiveness (simple events and basic queries as conditions) which however allow us to capture meaningful cases. In this way, we are able to provide effective results about important properties of active databases, which are difficult or impossible to test in general.

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 [26,27]. 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 [24].

References

- [1] S. Abiteboul and V. Vianu. Equivalence and optimization of relational transactions. *Journal of the ACM*, 35(1):70–120, January 1988.
- [2] S. Abiteboul, R. Hull and V. Vianu. *Foundation of Databases*. Addison-Wesley, 1995.
- [3] 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.
- [4] E. Baralis, J. Widom. An Algebraic Approach to Rule Analysis in Expert Database Systems. In *Proc. of International Conference on Very Large Data*

Bases, Santiago, pages 606–617, 1994.

- [5] E. Baralis, J. Widom. Using Delta Relations to Optimize Condition Evaluation in Active Databases. In *Proc. of the Second International Workshop on Rules in Database Systems, LNCS 985*, Springer-Verlag, pages 292–308, 1995.
- [6] S. Ceri, P. Fraternali, S. Paraboschi, and L. Tanca. Automatic generation of production rules for integrity maintenance. In *ACM Transaction on Database Systems*, 19(3):367–422, September 1994.
- [7] 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.
- [8] 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.
- [9] 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.
- [10] S. Ceri and J. Widom. Active Database Systems: Triggers and Rules for Advanced Database Processing. Morgan-Kaufmann, 1996.
- [11] C. Collet and J. Manchado. Optimization of active rules with parallelism. In *Proc. of Active and Real Time Database Systems*, Springer-Verlag, pages 82–103, 1995.
- [12] A. Dinn, N. W. Paton, M. H. Williams. Active Rule Analysis and Optimisation in the Rock & Roll Deductive Object-Oriented Database. *Information Systems*, 24(4):327–353, 1999.
- [13] A. A. A. Fernandes, M. H. Williams and N. W. Paton. A Logic-Based Integration of Active and Deductive Databases. *New Generation Computing*, 15(2):205–244, 1997.
- [14] 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.
- [15] N. Gehani and H. V. Jagadish. ODE as an active database: constraints and triggers. In *Proc. of the International Conf. on Very Large Data Bases, Barcelona*, pages 327–336, 1991.
- [16] S. Ghandeharizadeh et al. On Implementing a Language for Specifying Active Database Execution Models. In *Proc. of the International Conf. on Very Large Data Bases, Dublin*, pages 441–454, 1993.
- [17] 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.

- [18] G. Guerrini and D. Montesi. Design and implementation of Chimera's active rule language. *Data & Knowledge Engineering*, North-Holland, 1997.
- [19] E. N. Hanson and J. Widom. Rule processing in active database systems. In *International Journal of Expert Systems*, 6(1):83–119, 1993.
- [20] 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.
- [21] A. P. Karadimce, S. D. Urban. Refined Triggering Graphs: A Logic-Based Approach to Termination Analysis in an Active Object-Oriented Database. In *Proc. of the IEEE International Conf. on Data Engineering*, pages 384–391, 1996.
- [22] 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.
- [23] D. Montesi, M. Bagnato and C. Dallera. Termination Analysis in Active Databases. In *Proc. of International Database Engineering and Applications Symposium*, Montreal, IEEE Press, Montreal, 1999.
- [24] 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.
- [25] 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.
- [26] 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.
- [27] 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.
- [28] N. W. Paton. *Active Rules in Database Systems*. Springer-Verlag, 1999.
- [29] P. Picouet and V. Vianu. Expressiveness and Complexity of Active Databases. In *Proc. of the International Conference on Data Base Theory (ICDT'97)*, Springer-Verlag, pages 155–172, 1997.
- [30] M. Stonebraker. The integration of rule systems and database systems. *IEEE Trans. on Knowledge and Data Eng.*, 4(5):415–423, October 1992.
- [31] 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.

- [32] A. Vaduva, S. Gatzju and K. R. Dittrich. Investigating Termination in Active Database Systems with Expressive Rule Languages. In *Proc. of the International Workshop on Rules in Database Systems*, page 149–164, 1997.
- [33] L. van der Voort, A. Siebes. Termination and Confluence of Rule execution. In *Proc. of the ACM International Conf. on Information and Knowledge Management*, pages 245–255, 1993.
- [34] 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.
- [35] Y. Zhou, M. Hsu. A Theory for Rule Triggering Systems. In *Proc. of International Conf. on Advances in Database Technology*, pages 407–421, 1990.