Better Static Rule Analysis for Active Database Systems*

Elena Baralis†
Jennifer Widom

Department of Computer Science
Stanford University
Stanford, CA 94305-2140

{baralis,widom}@cs.stanford.edu

Abstract

Rules in active database systems can be very difficult to program, due to the unstructured and unpredictable nature of rule processing. We provide static analysis techniques for predicting whether a given rule set is guaranteed to terminate, and whether rule execution is confluent (guaranteed to have a unique final state). Our methods are based on previous techniques for analyzing rules in active database systems. We improve considerably on the previous techniques by providing analysis criteria that are much less conservative: our methods often determine that a rule set will terminate or is confluent when previous methods could not make this determination. Our improved analysis is based on a "propagation" algorithm, which uses an extended relational algebra to accurately determine when the action of one rule can affect the condition of another, and to determine when rule actions commute. We consider both Condition-Action rules and Event-Condition-Action rules, making our approach widely applicable to relational active database rule languages.

1 Introduction

An *active database system* is a conventional database system extended with a facility for managing *active rules* (or *triggers*). Incorporating active rules into a conventional database system has raised considerable interest both in the scientific community and in the commercial world: A number of prototypes that incorporate active rules into relational and object-oriented database systems have been developed to explore different features of active rule languages, and triggers are now available in many commercial relational products [WC96]. Rules may be activated either by the occurrence of events (*Event-Condition-Action* or *ECA* rules), or by the occurrence of particular database states (*Condition-Action* or *CA* rules). When activated, rules perform actions that may range from database manipulation operations to external procedure calls. Active rules provide a powerful mechanism for database system functions that were previously performed by user applications, e.g., general integrity constraint maintenance [CW90,CFPT94] and materialized view maintenance

*This work was partially performed while the authors were at the IBM Almaden Research Center, San Jose, CA. At Stanford this work was supported by the Anderson Faculty Scholar Fund and by equipment grants from Digital Equipment Corporation and IBM Corporation.

†Permanent address: Dipartimento di Automatica e Informatica, Politecnico di Torino, Corso Duca degli Abruzzi, 24-10129 Torino, Italy.

[CW91]. Active rules also can be used for advanced features of complex database applications, e.g., workflow management [DHL90].

Active database systems are very powerful, but developing even small active rule applications can be a difficult task, due to the unstructured and unpredictable nature of rule processing. During rule processing, rules can trigger and "untrigger" each other, and the intermediate and final states of the database can depend upon which rules are triggered and executed in which order. It has been observed in the past [AHW95, KU94, vdVS93] that two important and desirable properties of active rule behavior are *termination* and *confluence*. A rule set is guaranteed to terminate if, for any database state and initial modification, rule processing cannot continue forever (i.e., rules cannot activate each other indefinitely). A rule set is confluent if, for any database state and initial modification, the final database state after rule processing is independent of the order in which activated rules are executed.

Previous work on active rule analysis, e.g., [AHW95, KU94, vdVS93, WH95], has developed compile-time techniques that allow a rule programmer to predict in advance aspects of rule behavior such as termination and confluence. These techniques are used to statically analyze a set of rules before installing the rules in the database. Thus, static rule analysis can form the basis of a design methodology and programming environment for active database systems. A different approach has been used to provide design support for triggers in commercial systems: either syntactic limitations are imposed in order to guarantee run-time termination or confluence for any rule set (see, e.g., the forthcoming SQL3 standard [ISO94]), or run-time counters are used to prevent infinite rule execution (see, e.g., [Ora92]). The first approach severely limits the expressiveness of the active rule language, while the second may cause abnormal termination of rule processing even when rule execution would have ended successfully in a few more steps. Recently, a technique has been developed to improve run-time termination analysis [BCP95c].

In this paper we present new techniques for performing static analysis of both Condition-Action and Event-Condition-Action rules. Our analysis techniques are based on a generally applicable algorithm that is used to determine when the action of one rule can affect the condition of another rule, and to determine when rule actions commute. The algorithm uses an extension of relational algebra to model rule conditions and actions. Essentially, the algorithm "propagates" one rule's action through another rule's condition (or action) to determine how the action may affect the condition (or other action); hence, we call our algorithm the *Propagation Algorithm*. The Propagation Algorithm is useful for analyzing termination since it can determine when one rule may activate another rule. The Propagation Algorithm also is useful for analyzing confluence since it can determine when the execution order of two rules is significant. The Propagation Algorithm determines these properties much more accurately than previous methods, e.g., [AHW95, HH91, ZH90]. In addition, since we take a general approach based on relational algebra, and since we consider both Event-Condition-Action and Condition-Action rules, our method is applicable to most active database systems that use the relational model.

1.1 Previous Related Work

Most previous work on rule analysis in active database systems has focused either on Condition-Action rules or on Event-Condition-Action rules, and much of the work has been tailored to specific rule languages. Our work covers both ECA and CA rules, and it relies on relational algebra rather than a particular language.

In [HH91,ZH90] methods are given for analyzing Condition-Action rules. The goal of their work is to impose restrictions on rule sets so that confluence (a "unique fixed point" in their model) is guaranteed. By contrast, we provide techniques for analyzing the behavior of arbitrary rule sets. In addition, the methods in [HH91, ZH90] have been shown to be weaker than the methods in [AHW95], which in turn are weaker than the methods presented here. The methods in [AHW95] are developed in the context of the Starburst Rule System, which uses an Event-Condition-Action rule model. Their technique for analyzing rule interaction relies on a shallow comparison of the actions performed by one rule and the events, condition, and action of another rule. We improve significantly on this shallow analysis by using a formal algebraic model that allows us to accurately analyze the interaction between rules.

In an initial report we applied our approach to termination only [BCW93]. Subsequently, in [BW94] we refined the techniques in [BCW93] limited to Condition-Action rules, and we considered both termination and confluence. In this paper, we present a complete framework for the analysis of both Condition-Action and Event-Condition-Action rules. Furthermore, we extend the analysis techniques from [BCW93] to explicitly take into account the knowledge that a rule's condition is satisfied when the rule's action executes, and we extend our algorithm for determining when rule actions commute. We also provide a discussion of the complexity of the Propagation Algorithm, and we explore soundness and completeness properties.

In other related work, [vdVS93] analyzes rule behavior in the context of object-oriented active database systems. Their work focuses on differences between *instance-oriented* and *set-oriented* rules (we consider only set-oriented rules in this paper) and on decidability properties for rule analysis. Their rule model is fairly restrictive, in that rule actions (methods in their model) can only modify data selected by the corresponding rule condition, and deletions and insertions appear to be disallowed. The properties of confluence and of termination within some fixed number of steps are shown to be decidable using an approach based on "typical databases"; a typical database contains all possible data instances that could affect the outcome of rule processing. The rule set is "run" over the typical database and the outcome is checked for the desired properties. This approach is clearly infeasible in practical applications, so lower complexity algorithms are proposed, but the details and applicability of these algorithms are not clarified.

A rather different approach to active rule analysis is taken in [KU94], where ECA rules are reduced to *term rewriting systems*, and known analysis techniques for termination and confluence of term rewriting systems are applied. The analysis in [KU94] is based on an object-oriented data model and instance-oriented rule execution model. Their approach is powerful, since it exploits the body of work on Conditional Term Rewriting Systems (CTRS), but its implementation is complex even for small rule applications. Because the details of the rule language are not presented, it is unclear whether a general relational rule model such as ours can be expressed as a term rewriting system. Termination of a CTRS is proven by finding a well-founded ordering on terms in the CTRS, which can be a difficult task. In fact, in [KU94] it was not possible to prove termination for a rule set that enforces referential integrity constraints; termination of the same rule set (rewritten in relational algebra) can be proven with our technique. Confluence analysis is based on finding *contextual critical pairs* (terms that can be rewritten in two different ways, equivalent to rules triggered by the same events on the same classes) and analyzing their feasibility. Infeasibility is determined by detecting that two rule conditions cannot be true at the same time, which in most cases is harder to prove than rule commutativity.

Work in [BCP96] is focused on termination analysis. Active rules are grouped into modules; termination of rule execution within each module is assumed, and inter-module termination is analyzed. The termination analysis techniques in this paper are complementary to those presented in [BCP96], since the techniques presented here would be applied to the analysis of rules within a module. In the degenerate case of a single module, the techniques in [BCP96] are strictly weaker than those presented in this paper.

In [WH95] a technique is presented for termination analysis of ECA rules in the context of OSCAR, an object-oriented active database system. Although the data and rule model in OSCAR are quite different from ours, the analyzed rules correspond roughly to our *quasi-CA* rules with delta relations (see Section 5.2). The analysis technique in [WH95] consists of two stages. The results obtained in the first stage are slightly weaker than ours (e.g., the technique in [WH95] fails to detect non-activation of delete actions on select-project-join conditions). The second stage does better than our technique by detecting non-activation due to incremental (resp. decremental) updates with an upper (resp. lower) bound. A simple modification to our algorithms could detect this type of condition. Note, however, that going beyond analyzing pairs of rules using the extended technique is rather complex. In fact, there are cases in which the technique described in [WH95] fails to detect unbounded execution of a rule r when two other rules loop infinitely and periodically reactivate r.

Recent work in [BCP95b] proposes a technique that exploits the complementary information provided by *Triggering Graphs* and *Activation Graphs* to analyze termination of ECA rule sets. Techniques for constructing the graphs are assumed. Although building a Triggering Graph is straightforward, building an accurate Activation Graph is not. The techniques presented in this paper for analyzing rule activation can be used to build the Activation Graph needed in [BCP95b].

In traditional expert systems, i.e., production rule systems such as OPS5 [BFKM85], predicting properties such as termination and confluence appears to be of less importance than in the database environment; consequently, there has been less work on rule analysis in traditional expert systems than in active database systems. A recent paper [TC94] addresses termination analysis in the context of real-time rule-based systems, with the goal of guaranteeing bounded response time. In this work, a method based on an *Enable-Rule* graph is proposed to analyze OPS5 rule sets. The Enable-Rule graph is analogous to our Activation Graph for Condition-Action rules [BCW93, BW94]. In [TC94] a further step is performed: when a cycle is detected, the "enabling condition" for the cycle is used to automatically generate extra rules that break execution cycles. Cycles are broken either by rolling back, or by deleting from the working memory all instances causing the loop. While the basic analysis technique in [TC94] is similar to ours, it applies to a more restricted rule language.

Our Propagation Algorithm is closely related to the problem of *independence of queries and updates*, addressed in, e.g., [Elk90,LS93]. [LS93], which subsumes [Elk90], gives an algorithm for determining if the outcome of a query, expressed as a Datalog program, can be affected by a given insertion or deletion. The algorithm described in [LS93] applies to more general queries than we consider here (e.g., recursive queries), but it returns a boolean "affects/does-not-affect" answer, while our Propagation Algorithm returns a relational expression describing the actual effect. For analyzing active database rules, when a query and update are not independent, we need to know whether the update adds to, removes from, or modifies the result of the query. With appropriate extensions, this information could be returned by the algorithm in [LS93] as well, in which case their algorithm could be used in our rule analysis techniques in place of our Propagation Algorithm.

Finally, our Propagation Algorithm is somewhat related to *incremental evaluation*, as in, e.g., [BW95,QW91,RCBB89]. Both problems address the effect of a data modification on a relational expression. However, incremental evaluation techniques are designed for run-time, when the actual modifications are known, while our techniques apply at compile-time, when the modifications are represented as relational expressions.

1.2 The Example Rule Set

In this section, we describe informally the rule set on which we will base examples throughout the paper. The rules refer to the following relations:

`ACCOUNT(`*num*`,name,balance,rate)`
`CUSTOMER(`*name*`,address,city)`
`LOW-ACC(`*num,start*`,end)`

Relation `ACCOUNT` contains information on a bank's accounts, while relation `CUSTOMER` contains information on the bank's customers. Relation `LOW-ACC` contains a history of all time periods in which an account had a low balance, including the date on which the balance became low and the date on which it became high again (the latter value can be null, indicating that the account is still low). We assume that the italicized attributes are a key for the corresponding relation, although our method does not rely on keys.

The following rules automatically enforce some of the bank's policies for managing customers and accounts:

r_1: Rule `bad-account` states that if an account has a balance less than 500 and an interest rate greater than 0%, then that account's interest rate is lowered to 0%.

r_2: Rule `raise-rate` states that if an account has an interest rate greater than 1% but less than 2%, then that account's interest rate is raised to 2%.

r_3: Rule `SF-bonus` states that when the number of customers living in San Francisco exceeds 1000, then the interest rate of all San Francisco customers' accounts with a balance greater than 5000 and an interest rate less than 3% is increased by 1%.

r_4: Rule `start-bad` states that if an account has a balance less than 500 and is not yet recorded with a null end date in the `LOW-ACC` relation, then the account is inserted into the `LOW-ACC` relation with the current date as start date and a null end date.

r_5: Rule `end-bad` states that if an account with a null end date in the `LOW-ACC` relation has a balance of at least 500 in the `ACCOUNT` relation, then its end date is set to the current date.

r_6: Rule `decrease-bad` states that if the total number of low days for an account (as recorded in the `LOW-ACC` relation) is greater than 50 and its current balance is between 500 and 1000, then its interest rate is set to 1% in the `ACCOUNT` relation.

1.3 Outline of the Paper

In Section 2 we describe the algebraic language we use as the basis of our work. Section 3 contains the Propagation Algorithm, examples of its application, a discussion of its complexity, and a proof of its completeness. In Sections 4 and 5 we describe the Condition-Action and Event-Condition-Action rule languages respectively, and we apply the Propagation Algorithm to the analysis of termination and confluence for these languages; several examples are included. In Section 6 we present an improvement of commutativity analysis for rule actions. Section 7 draws conclusions and outlines future work.

2 Algebraic Query and Modification Representation

In this section we describe the extensions to relational algebra that are required to represent general database query and modification operations. This representation will be used in the following sections for rule conditions, which are queries on the database, and rule actions, which are data modifications; both are represented by relational algebra expressions. We first define the operators in our extended relational algebra. Then we give the representation of modification operations, and finally we give some examples of both queries and modifications.

2.1 Algebraic Operators

Based on [CG85,Klu82], we define an extension to relational algebra that allows us to represent any queries that are expressible in SQL (or Quel), with the exception of the handling of duplicates and ordering conditions. We also introduce an extension that allows us to represent the SQL data modification operations **insert**, **delete**, and **update**.

Operator	Description
$\ltimes_p$	semijoin with predicate p
$\ltimes_{\nexists p}$	not-exists semijoin with predicate p
$\alpha_{A_1;A_2}$	attribute rename
$\mathcal{E}[X = expr]$	attribute extension and expression evaluation
$\mathcal{A}[X = a(A); B]$	attribute extension and aggregate function evaluation

Table 1: Additional algebraic operators

Our extended relational algebra includes the basic relational algebra operators *select* (σ), *project* (π), *cross-product* ($\times$), *natural join* ($\bowtie$), *union* ($\cup$), and *difference* ($-$), which we do not elaborate on here; see [Ull89]. The first two lines of Table 1 present useful operators derived from the basic operators, while the next three lines present additional operators that we use. In the table, X and A denote attributes, B, A_1, and A_2 denote attribute lists, a is an aggregate function, and $expr$ is an expression (explained below). In line 1, $E_1 \ltimes_p E_2 = \pi_{schema(E_1)}(\sigma_p(E_1 \times E_2))$; in line 3, α renames the attributes in list A_1 as A_2. In the remainder of the paper, we adopt the shorthand notation $E_1 \ltimes E_2$ and $E_1 \ltimes_{\nexists} E_2$ to denote $E_1 \ltimes_p E_2$ and $E_1 \ltimes_{\nexists p} E_2$ when predicate p equates all attributes with the same name in both $schema(E_1)$ and $schema(E_2)$ (similar to the natural join). We now discuss the other operators in more detail, then we present the modification operations.

2.1.1 Not-Exists Semijoin

The *not-exists semijoin* operator, $\ltimes_{\nexists p}$, is introduced to concisely express *negative subqueries* as they are expressed in SQL (e.g., **not exists**); negative subqueries appear frequently in rule definitions [CW90]. The not-exists semijoin operator is defined as:

$$E_1 \ltimes_{\nexists p} E_2 = E_1 - (E_1 \ltimes_p E_2)$$

Note that we could instead define the relational difference operator in terms of not-exists semijoin: $E_1 - E_2 = E_1 \ltimes_{\nexists} E_2$ (with renaming of attributes in E_1 and E_2 as necessary). For convenience, we consider only the not-exists semijoin and not the difference operator in the remainder of the paper.

2.1.2 Aggregate Functions and Expression Evaluation

The *attribute extension* operators allow us to extend a relational expression E with a new attribute; this approach is used for aggregate functions and for modification operations. We have:

- The $\mathcal{E}$ operator, which computes expressions applied to each tuple of E
- The $\mathcal{A}$ operator, which computes aggregate functions (e.g., `max, min, avg, sum, count`) over partitions of E

Operation	Algebraic expression	New database state
insert	E_{ins}	$R \cup E_{ins}$
delete	E_{del}	$R \ltimes_{\nexists} E_{del}$
update	E_{upd}	$(R \ltimes_{\nexists} E_{upd}) \ \cup \ \alpha_{A'_u;A_u}(\pi_{A_r,A'_u} E_{upd})$

Table 2: Algebraic description of insert, delete, and update operations

$\mathcal{E}$ is a unary operator applied to a relational expression E producing a result with schema $schema(E) \cup \{X\}$. Recall from Table 1 that the $\mathcal{E}$ operator is expressed as:

$$\mathcal{E}[X = expr]E$$

$expr$ is an expression evaluated over each tuple t of E (a conventional expression involving attributes of t and constants) yielding one value for each tuple; this value is entered into the new attribute X for each tuple of E. For details of similar operators see [CCRL+90]; examples are given in later sections.

$\mathcal{A}$ is also a unary operator applied to a relational expression E producing a result with schema $schema(E) \cup \{X\}$. Recall from Table 1 that the $\mathcal{A}$ operator is expressed as:

$$\mathcal{A}[X = a(A); B]E$$

B defines a set of attributes on which the result of E is partitioned; each group in the partition contains all tuples with the same B value. a is an aggregate function that is applied to the (multiset of) values contained in the projection of each partition on attribute A, yielding one value for each partition; this value is entered into the new attribute X for each tuple of the partition. The attributes B are optional: when B is omitted, no grouping is performed, and the aggregate function a is applied to the entire result of E, yielding one value; that value is entered into the new attribute X for each tuple of E. For details see [CG85].

2.1.3 Modification Operations

We represent data modification operations in relational algebra by characterizing the operations in terms of the database state they produce. Table 2 presents inserts, deletes, and updates by indicating the algebraic expressions that are used to denote the operations, and the way in which these expressions are applied to a relation R to produce a new value for R. In the table, A_u denotes the attributes of R that are updated, A'_u denotes primed versions of these attributes (explained below), and $A_r = schema(R) - A_u$.

Insert operation. An insert operation is denoted by a relational expression E_{ins}. E_{ins} produces the tuples to be inserted (either a constant tuple or the result of an algebraic expression). The schema of E_{ins} must coincide with the schema of R.

Delete operation. A delete operation is denoted by a relational expression E_{del}. E_{del} produces the tuples to be deleted. The schema of E_{del} must coincide with the schema of R.

Update operation. An update operation is denoted by a relational expression E_{upd}. E_{upd} has schema $schema(R) \cup A'_u$, where attributes A'_u contain the new values for the updated attributes A_u. As convention, the new values for the updated attributes are always assigned the corresponding "primed" attribute names. That is, if attribute $A \in A_u$ is updated, then the new value for A is assigned to attribute A'. The update operation only operates on tuples already present in the relation to be updated. Thus, $\pi_{schema(R)} E_{upd} \subseteq R$. E_{upd} is expressed as:

$$E_{upd} = \mathcal{E}[A'_{u1} = expr_1]\ \mathcal{E}[A'_{u2} = expr_2]\ \ldots\ \mathcal{E}[A'_{un} = expr_n]\ E_c$$

where E_c is an expression producing the tuples to be updated (i.e., the "selection condition" of the update operation). The schema of E_c must coincide with the schema of R. $\mathcal{E}[A'_{ui} = expr_i]$ evaluates expression $expr_i$ on each tuple of E_c and assigns the result to the new attribute A'_{ui}.

As specified in Table 2, the new state of R after the update operation is the union of two terms:

1. The first term $R \ltimes_{\nexists} E_{upd}$ includes in the result all tuples in R that are not modified by the update operation.

2. The second term $\alpha_{A'_u;A_u}(\pi_{A_r,A'_u} E_{upd})$ includes in the result the original values for the non-updated attributes of the modified tuples and the new values for the modified attributes, with the primed attribute names replaced by the original attribute names.

Given a relational expression E with schema $schema(R) \cup A'_u$, we often need the corresponding expression that is compatible in schema with R and contains either the pre-updated (old) or the updated (new) values for the modified attributes. For convenience we will use the abbreviations:

$$\rho_{old}(E) = \pi_{schema(R)}(E)$$
$$\rho_{new}(E) = \alpha_{A'_u;A_u}(\pi_{schema(E)-A_u} E)$$

2.2 Examples

In this section we give the algebraic representation of a query and a modification operation. The examples use the relations defined in Section 1.2.

Example 2.1: Select the information on all San Francisco (SF) customers' accounts with a balance greater than 5000 and an interest rate less than 3%. In our algebraic language this query is expressed as:

$$(\sigma_{\texttt{balance}>5000 \wedge \texttt{rate}<3} \texttt{ACCOUNT}) \ltimes (\sigma_{\texttt{city}='\texttt{SF}'} \texttt{CUSTOMER})$$

Example 2.2: Raise the interest rate to 2% for all accounts that have an interest rate greater than 1% but less than 2%. In our algebraic language this modification is expressed as:

$$\mathcal{E}[\texttt{rate}' = 2]\sigma_{\texttt{rate}>1 \wedge \texttt{rate}<2}\texttt{ACCOUNT}$$

3 The Propagation Algorithm

In this section we describe a general algorithm, which we call the *Propagation Algorithm*, that uses syntactic analysis to determine how a database query can be affected by the execution of a data modification operation. We initially describe the structure of the algorithm's output, and we discuss the accuracy of the algorithm. Then we present the algorithm itself along with several examples of its application. Finally we discuss the complexity of the algorithm, along with its soundness and completeness.

3.1 Output of the Algorithm

The input to the Propagation Algorithm is a query and a modification, expressed in our algebraic language. The output of the algorithm is zero or more of each of the operations *insert*, *delete*, and *update*, characterizing how the result of the query may change due to the execution of the modification: If the algorithm produces an *insert* operation, then the query may contain more data after the modification; if the algorithm produces a *delete* operation, then the query may contain less data after the modification; if the algorithm produces an *update* operation, then the query may contain updated data after the modification; if no operations are produced, then the result of the query cannot change due to the modification. The operations produced by our algorithm are represented as relational expressions in the same way that we algebraically represent data modification operations in Section 2.1.3, except here the modifications apply to arbitrary relational expressions instead of only to single relations.

The output of the Propagation Algorithm may contain more than one each of the insert, delete, and update operations. Let the Propagation Algorithm, applied to a query Q and a modification M, produce n_I insert operations $E^1_{ins}, \ldots, E^{n_I}_{ins}$, n_U update operations $E^1_{upd}, \ldots, E^{n_U}_{upd}$, and n_D delete operations $E^1_{del}, \ldots, E^{n_D}_{del}$. We also consider the changes to the result of Q due to the execution of M as represented by *negative modifications* E^- and *positive modifications* E^+, defined based on the Propagation Algorithm's output as:

$$E^-(Q,M) = \bigcup_{i=1}^{n_D} E^{(i)}_{del} \cup \bigcup_{i=1}^{n_U} \rho_{old}(E^{(i)}_{upd})$$

$$E^+(Q,M) = \bigcup_{i=1}^{n_I} E^{(i)}_{ins} \cup \bigcup_{i=1}^{n_U} \rho_{new}(E^{(i)}_{upd})$$

$E^-(Q,M)$ and $E^+(Q,M)$ are respectively the changes subtracted from and added to the result of Q. In practice, the number of modifications of each type, i.e., n_I, n_D, and n_U, are almost always zero or one (see the examples in Section 3.5).

3.2 Accuracy of the Output

Given a query Q and a modification M as input, the Propagation Algorithm produces as output a set of modifications that characterize how the result of Q changes due to the execution of M. The Propagation Algorithm is "conservative," as follows. Let $Q(d)$ denote the result of evaluating a query Q on database state d. The operations produced by the Propagation Algorithm are guaranteed to yield, for all database states d, all the modifications on $Q(d)$ that are caused by performing the original modification M on d. However, the operations may also yield some modifications that are not actually performed on $Q(d)$ due to the execution of M on d. For example, the algorithm may output an insert operation that, when evaluated in state d includes tuple t, but tuple t is already in $Q(d)$, thus t is not actually inserted.

Hence, the Propagation Algorithm accurately determines if the execution of a modification M *may* produce a change in Q—if the result is empty, we know Q is always unchanged by M. In practice, we have found that the slight "conservativeness" of the algorithm does not compromise the effectiveness of our analysis techniques.

3.3 The Algorithm

The Propagation Algorithm takes as input a query Q and a modification M, both expressed in extended relational algebra as defined in Section 2. As an initial filter, if the query Q does not reference the relation modified by M, then clearly M cannot affect the result of Q. Otherwise, M is "propagated" through a tree representation of query Q. The leaves of the tree are relations, and one of these leaves corresponds to the relation R that is modified by M. (We assume there is only one reference to R in query Q. If there are multiple references, we would need to treat each reference independently and take the union of the results.) Modification M is propagated from the affected relation up the query tree, and it may be transformed into one or more different modification operations during the propagation process. To describe the propagation, we give formal rules specifying how arbitrary modifications are propagated through arbitrary nodes of the tree. After each propagation through a node in the tree, the modifications obtained are checked for "consistency" (explained next). Inconsistent modifications are discarded, while consistent modifications are propagated further. The propagation process continues until the root of the query tree is reached or all modifications have been discarded as inconsistent. At each point during the propagation process, the modifications associated with a node N in the tree indicate the modifications that may occur to N's subtree as a result of performing the original modification M. Hence, the consistent modifications that reach the root of the tree indicate how the original modification M may affect query Q.

A modification produced by the propagation process is *consistent* when the algebraic expression representing the modification does not contain contradictions, i.e., it is satisfiable. Satisfiability of relational expressions is undecidable in the general case, so we can give sufficient but not necessary conditions for satisfiability of the propagated expressions. However, for most expressions that arise in practice, either we can see trivially whether the expression is satisfiable (as in examples in

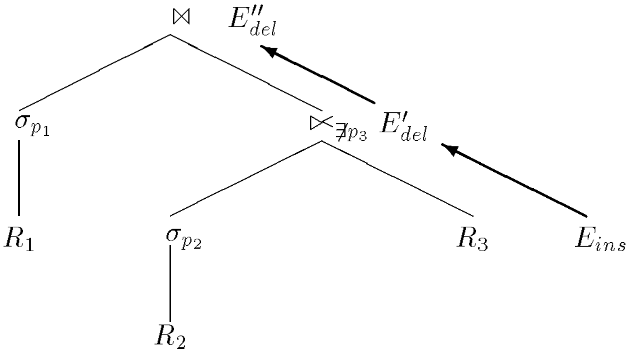

Figure 1: Propagation of E_{ins} modification

Section 3.5), or we can verify satisfiability using the tableau method in [Ull89].[1]

Figure 1 illustrates the propagation of an insert operation (represented by expression E_{ins}) on relation R_3 through the nodes of the query tree representing the query $Q = (\sigma_{p_1} R_1) \bowtie (\sigma_{p_2} R_2 \triangleright\!\!<_{\nexists p_3} R_3)$. The bold line represents the propagation path of the E_{ins} modification: The E_{ins} modification is first substituted for the affected relation R_3. Then, starting from the $\triangleright\!\!<_{\nexists p_3}$ node, for each node with an operand affected by the E_{ins} modification, the corresponding propagated expression is computed. At the end of the propagation process a delete operation E''_{del} is obtained at the root. As the reader may verify, an insert operation on R_3 may only cause data satisfying Q to be deleted.

3.4 The Rules for Propagation

The rules for propagation are given in tables based on the kind of incoming modification: insert and delete modifications in Tables 3 and 4 respectively, and update modifications in Tables 5 and 6. Each row in the tables contains the propagated modification(s), E^{out}, as a function of the incoming modification, E^{in}, and the relational operator in the query tree. The column labeled "Applicability condition" specifies when different propagation rules are used for different cases. In the tables, A_1, A_2, and B are attribute lists, $A_{jn} = schema(E_1) \cap schema(E_2)$, $A_{E_2} = schema(E_2)$, A_u are the updated attributes, A_p and A_e are the attributes involved in predicate p and expression $expr$ respectively, $p' = \alpha_{A_u;A'_u} p$ and $expr' = \alpha_{A_u;A'_u} expr$, $p(B)$ equates all attributes in list B, and $p'(A_{uB})$ equates all attributes in A'_{uB} with the corresponding B attributes. Since the natural join, cartesian product, and union operators are symmetric, without loss of generality we assume that the first operand is modified; analogous rules apply for modifications to the second operand. Observe that aggregate functions require, in addition to the incoming modification, the entire relational expression E to which the aggregate function is applied.

[1] Note that a conservative test for satisfiability is not really a limitation here, since our entire approach is based on syntactic analysis and hence is conservative; recall Section 3.2.

	Propagated modification: $E^{in}_{ins} \rightarrow E^{out}_{modification}$	
Node	Applicability condition	Resulting expression
$\sigma_p E$		$E^{out}_{ins} = \sigma_p E^{in}_{ins}$
$\pi_{A_1} E$		$E^{out}_{ins} = \pi_{A_1} E^{in}_{ins}$
$E_1 \bowtie E_2$		$E^{out}_{ins} = E^{in}_{ins} \bowtie E_2$
$E_1 \times E_2$		$E^{out}_{ins} = E^{in}_{ins} \times E_2$
$E_1 \cup E_2$		$E^{out}_{ins} = E^{in}_{ins} \ltimes_{\nexists} E_2$
$E_1 \ltimes_p E_2$	insert into E_1	$E^{out}_{ins} = E^{in}_{ins} \ltimes_p E_2$
	insert into E_2	$E^{out}_{ins} = E_1 \ltimes_p E^{in}_{ins}$
$E_1 \ltimes_{\nexists p} E_2$	insert into E_1	$E^{out}_{ins} = E^{in}_{ins} \ltimes_{\nexists p} E_2$
	insert into E_2	$E^{out}_{del} = E_1 \ltimes_p E^{in}_{ins}$
$\alpha_{A_1;A_2} E$		$E^{out}_{ins} = \alpha_{A_1;A_2} E^{in}_{ins}$
$\mathcal{E}[X = expr]E$		$E^{out}_{ins} = \mathcal{E}[X = expr]E^{in}_{ins}$
$\mathcal{A}[X = a(A); B]E$	$B = \emptyset$	$E^{out}_{ins} = E^{in}_{ins} \bowtie \mathcal{A}[X = a(A)](E \cup E^{in}_{ins})$ $E^{out}_{upd} = (\mathcal{A}[X' = a(A)](E \cup E^{in}_{ins})) \bowtie (\mathcal{A}[X = a(A)]E)$
	$B \neq \emptyset$	$E^{out}_{ins} = E^{in}_{ins} \bowtie \mathcal{A}[X = a(A); B](E \cup E^{in}_{ins})$ $E^{out}_{upd} = ((\mathcal{A}[X' = a(A); B](E \cup E^{in}_{ins})) \bowtie$ $(\mathcal{A}[X = a(A); B]E)) \ltimes_{p(B)} E^{in}_{ins}$

Table 3: Insert operation propagation

	Propagated modification: $E^{in}_{del} \rightarrow E^{out}_{modification}$	
Node	Applicability condition	Resulting expression
$\sigma_p E$		$E^{out}_{del} = \sigma_p E^{in}_{del}$
$\pi_{A_1} E$		$E^{out}_{del} = \pi_{A_1} E^{in}_{del}$
$E_1 \bowtie E_2$		$E^{out}_{del} = E^{in}_{del} \bowtie E_2$
$E_1 \times E_2$		$E^{out}_{del} = E^{in}_{del} \times E_2$
$E_1 \cup E_2$		$E^{out}_{del} = E^{in}_{del} \ltimes_{\nexists} E_2$
$E_1 \ltimes_p E_2$	delete from E_1	$E^{out}_{del} = E^{in}_{del} \ltimes_p E_2$
	delete from E_2	$E^{out}_{del} = E_1 \ltimes_p E^{in}_{del}$
$E_1 \ltimes_{\nexists p} E_2$	delete from E_1	$E^{out}_{del} = E^{in}_{del} \ltimes_{\nexists p} E_2$
	delete from E_2	$E^{out}_{ins} = E_1 \ltimes_p E^{in}_{del}$
$\alpha_{A_1;A_2} E$		$E^{out}_{del} = \alpha_{A_1;A_2} E^{in}_{del}$
$\mathcal{E}[X = expr]E$		$E^{out}_{del} = \mathcal{E}[X = expr]E^{in}_{del}$
$\mathcal{A}[X = a(A); B]E$	$B = \emptyset$	$E^{out}_{del} = E^{in}_{del} \bowtie (\mathcal{A}[X = a(A)]E)$ $E^{out}_{upd} = (\mathcal{A}[X' = a(A)](E \ltimes_{\nexists} E^{in}_{del})) \bowtie (\mathcal{A}[X = a(A)]E)$
	$B \neq \emptyset$	$E^{out}_{del} = E^{in}_{del} \bowtie \mathcal{A}[X = a(A); B]E$ $E^{out}_{upd} = ((\mathcal{A}[X' = a(A); B](E \ltimes_{\nexists} E^{in}_{del})) \bowtie$ $(\mathcal{A}[X = a(A); B]E)) \ltimes_{p(B)} E^{in}_{del}$

Table 4: Delete operation propagation

	Propagated modification: $E^{in}_{upd} \rightarrow E^{out}_{modification}$	
Node	Applicability condition	Resulting expression
$\sigma_p E$	$A_u \cap A_p = \emptyset$	$E^{out}_{upd} = \sigma_p E^{in}_{upd}$
	$A_u \cap A_p \neq \emptyset$	$E^{out}_{ins} = \rho_{new}((\sigma_{p'} E^{in}_{upd}) \ltimes_{\nexists} (\sigma_p E^{in}_{upd}))$ $E^{out}_{del} = \rho_{old}((\sigma_p E^{in}_{upd}) \ltimes_{\nexists} (\sigma_{p'} E^{in}_{upd}))$ $E^{out}_{upd} = (\sigma_{p'} E^{in}_{upd}) \bowtie (\sigma_p E^{in}_{upd})$
$\pi_{A_1} E$	$A_u \cap A_1 = \emptyset$	$\emptyset$
	$A_u \cap A_1 = A_{u1}$	$E^{out}_{upd} = \pi_{A_1, A'_{u1}} E^{in}_{upd}$
$E_1 \bowtie E_2$	$A_u \cap A_{jn} = \emptyset$	$E^{out}_{upd} = E^{in}_{upd} \bowtie E_2$
	$A_u \cap A_{jn} = A_{ujn}$	$E^{out}_{ins} = \rho_{new}((E^{in}_{upd} \bowtie (\alpha_{A_{ujn}; A'_{ujn}} E_2)) \ltimes_{\nexists} (E^{in}_{upd} \bowtie E_2))$ $E^{out}_{del} = \rho_{old}((E^{in}_{upd} \bowtie E_2) \ltimes_{\nexists} (E^{in}_{upd} \bowtie (\alpha_{A_{ujn}; A'_{ujn}} E_2)))$ $E^{out}_{upd} = ((E^{in}_{upd} \bowtie (\alpha_{A_{E_2}; A'_{E_2}} E_2)) \bowtie (E^{in}_{upd} \bowtie E_2))$
$E_1 \times E_2$		$E^{out}_{upd} = E^{in}_{upd} \times E_2$
$E_1 \cup E_2$		$E^{ins}_{out} = \rho_{new}((E^{in}_{upd} \ltimes E_2) \ltimes_{\nexists} (\alpha_{A_u; A'_u} E_2))$ $E^{del}_{out} = \rho_{old}((E^{in}_{upd} \ltimes_{\nexists} E_2) \ltimes (\alpha_{A_u; A'_u} E_2))$ $E^{upd}_{out} = (E^{in}_{upd} \ltimes_{\nexists} E_2) \ltimes_{\nexists} (\alpha_{A_u; A'_u} E_2)$
$E_1 \ltimes_p E_2$	update E_1, $A_u \cap A_p = \emptyset$	$E^{out}_{upd} = E^{in}_{upd} \ltimes_p E_2$
	update E_1, $A_u \cap A_p \neq \emptyset$	$E^{out}_{ins} = \rho_{new}((E^{in}_{upd} \ltimes_{p'} E_2) \ltimes_{\nexists} (E^{in}_{upd} \ltimes_p E_2))$ $E^{out}_{del} = \rho_{old}((E^{in}_{upd} \ltimes_p E_2) \ltimes_{\nexists} (E^{in}_{upd} \ltimes_{p'} E_2))$ $E^{out}_{upd} = (E^{in}_{upd} \ltimes_{p'} E_2) \bowtie (E^{in}_{upd} \ltimes_p E_2)$
	update E_2, $A_u \cap A_p = \emptyset$	$\emptyset$
	update E_2, $A_u \cap A_p \neq \emptyset$	$E^{out}_{ins} = (E_1 \ltimes_{p'} E^{in}_{upd}) \ltimes_{\nexists} (E_1 \ltimes_p E^{in}_{upd})$ $E^{out}_{del} = (E_1 \ltimes_p E^{in}_{upd}) \ltimes_{\nexists} (E_1 \ltimes_{p'} E^{in}_{upd})$
$E_1 \ltimes_{\nexists p} E_2$	update E_1, $A_u \cap A_p = \emptyset$	$E^{out}_{upd} = E^{in}_{upd} \ltimes_{\nexists p} E_2$
	update E_1, $A_u \cap A_p \neq \emptyset$	$E^{out}_{ins} = \rho_{new}((E^{in}_{upd} \ltimes_{\nexists p'} E_2) \ltimes_{\nexists} (E^{in}_{upd} \ltimes_{\nexists p} E_2))$ $E^{out}_{del} = \rho_{old}((E^{in}_{upd} \ltimes_{\nexists p} E_2) \ltimes_{\nexists} (E^{in}_{upd} \ltimes_{\nexists p'} E_2))$ $E^{out}_{upd} = (E^{in}_{upd} \ltimes_{\nexists p'} E_2) \bowtie (E^{in}_{upd} \ltimes_{\nexists p} E_2)$
	update E_2, $A_u \cap A_p = \emptyset$	$\emptyset$
	update E_2, $A_u \cap A_p \neq \emptyset$	$E^{out}_{ins} = (E_1 \ltimes_{\nexists p'} E^{in}_{upd}) \ltimes_{\nexists} (E_1 \ltimes_{\nexists p} E^{in}_{upd})$ $E^{out}_{del} = (E_1 \ltimes_{\nexists p} E^{in}_{upd}) \ltimes_{\nexists} (E_1 \ltimes_{\nexists p'} E^{in}_{upd})$
$\alpha_{A_1; A_2} E$	$A_1 \cap A_u = \emptyset$	$E^{out}_{upd} = \alpha_{A_1; A_2} E^{in}_{upd}$
	$A_1 \cap A_u = A_{u1}$,	$E^{out}_{upd} = \alpha_{A_1; A_2} \alpha_{A'_{u1}; A'_{u2}} E^{in}_{upd}$
$\mathcal{E}[X = expr]E$	$A_u \cap A_e = \emptyset$	$E^{out}_{upd} = \mathcal{E}[X = expr] E^{in}_{upd}$
	$A_u \cap A_e \neq \emptyset$	$E^{out}_{upd} = \mathcal{E}[X = expr] \mathcal{E}[X' = expr'] E^{in}_{upd}$

Table 5: Update operation propagation

	Propagated modification: $E^{in}_{upd} \rightarrow E^{out}_{modification}$	
Node	Applicability condition	Resulting expression
$\mathcal{A}[X = a(A); B]E$	$A_u \cap A = \emptyset$, $A_u \cap B = \emptyset$	$E^{out}_{upd} = E^{in}_{upd} \bowtie (\mathcal{A}[X = a(A); B]E)$
	$B = \emptyset$, $A_u \supseteq A$	$E^{out}_{upd1} = (E^{in}_{upd} \bowtie (\mathcal{A}[X = a(A)]E)) \bowtie$ $\alpha_{A_u;A'_u}(\mathcal{A}[X' = a(A)](\rho_{new}(E^{in}_{upd}) \cup (E \ltimes_{\nexists} \rho_{old}(E^{in}_{upd})))))$ $E^{out}_{upd2} = ((\mathcal{A}[X = a(A)]E) \bowtie (\mathcal{A}[X' = a(A)](\rho_{new}(E^{in}_{upd}) \cup$ $(E \ltimes_{\nexists} \rho_{old}(E^{in}_{upd})))))$
	$B \neq \emptyset$, $A_u \supseteq A$, $A_u \cap B = \emptyset$	$E^{out}_{upd1} = (E^{in}_{upd} \bowtie (\mathcal{A}[X = a(A); B]E)) \bowtie$ $\alpha_{A_u;A'_u}(\mathcal{A}[X' = a(A); B](\rho_{new}(E^{in}_{upd}) \cup (E \ltimes_{\nexists} \rho_{old}(E^{in}_{upd})))))$ $E^{out}_{upd2} = ((\mathcal{A}[X = a(A); B]E) \bowtie (\mathcal{A}[X' = a(A); B](\rho_{new}(E^{in}_{upd}) \cup$ $(E \ltimes_{\nexists} E^{in}_{upd})))) \ltimes_{p(B)} E^{in}_{upd})$
	$B \neq \emptyset$, $A_u \cap B = A_{uB}$	$E^{out}_{upd1} = (E^{in}_{upd} \bowtie (\mathcal{A}[X = a(A); B]E)) \bowtie$ $\alpha_{A_u;A'_u}(\mathcal{A}[X' = a(A); B](\rho_{new}(E^{in}_{upd}) \cup$ $(E \ltimes_{\nexists} \rho_{old}(E^{in}_{upd})))))$ $E^{out}_{upd2} = ((\mathcal{A}[X = a(A); B]E) \bowtie (\mathcal{A}[X' = a(A); B](\rho_{new}(E^{in}_{upd}) \cup$ $(E \ltimes_{\nexists} E^{in}_{upd}))) \ltimes_{p(B) \wedge p'(A_{uB})} E^{in}_{upd})$

Table 6: Update operation propagation (cont.)

The formulas given in Table 5 don't take into account the internal structure of selection predicates and update expressions. In the case of simple predicates (comparisons between an attribute and a constant[2]) and simple arithmetic update expressions (addition or subtraction of constants from an attribute), in many cases it is possible to eliminate some of the propagated modifications. For example, consider the propagation of the update $A = A + 1$ through the operation $\sigma_{A>5}$. Intuitively, this update will never cause tuples to be deleted from the expression rooted in $\sigma_{A>5}$. Thus, the propagated delete operation can be eliminated. Table 7 shows the modifications that can be eliminated in the different cases. In the table, "other" indicates an arbitrary arithmetic expression, which in the case of an equality or non-equality predicate still allows an update operation to be eliminated.

3.5 Examples

We give two examples of applying the Propagation Algorithm using queries and modifications expressed on the relations defined in Section 1.2. In each example, we fully describe the propagation process and the satisfiability test.

Example 3.1: Query Q selects the balance and rate of all accounts that have balance < 500 and rate $> 0\%$. Update operation M increases by 1% the rate of all San Francisco customers having accounts with balance > 5000 and rate $< 3\%$. The input to the algorithm is:

[2]Actually, any expression involving non-updated attributes and constants can be considered as a constant in this context.

	Arithmetic update expression		
predicate	addition	subtraction	other
$A_u = k,\ A_u \neq k$	E_{upd}	E_{upd}	E_{upd}
$A_u > k,\ A_u \geq k$	E_{del}	E_{ins}	–
$A_u < k,\ A_u \leq k$	E_{ins}	E_{del}	–

Table 7: Eliminated modifications

$$Q = \pi_{\texttt{balance,rate}}(\sigma_{\texttt{balance<500} \wedge \texttt{rate>0}}\texttt{ACCOUNT})$$
$$M = E_{upd} = \mathcal{E}[\texttt{rate}' = \texttt{rate} + 1](\sigma_{\texttt{balance>5000} \wedge \texttt{rate<3}}(\texttt{ACCOUNT} \ltimes (\sigma_{\texttt{city='SF'}}\texttt{CUSTOMER})))$$

Using Table 5, the propagation of E_{upd} through the selection operation in Q yields insert and update operations (the delete operation is eliminated, see Table 7). We have:

$$E'_{ins} = \rho_{new}((\sigma_{\texttt{balance<500} \wedge \texttt{rate'>0}}E_{upd}) \ltimes_{\nexists} (\sigma_{\texttt{balance<500} \wedge \texttt{rate>0}}E_{upd}))$$
$$E'_{upd} = (\sigma_{\texttt{balance<500} \wedge \texttt{rate'>0}}E_{upd}) \bowtie (\sigma_{\texttt{balance<500} \wedge \texttt{rate>0}}E_{upd})$$

In both cases, predicates $\texttt{balance} < \texttt{500}$ and $\texttt{balance} > \texttt{5000}$ (the latter from E_{upd}) are contradictory, so both expressions E'_{ins} and E'_{upd} are unsatisfiable. Intuitively, modification M operates on data not read by query Q. We conclude that modification M cannot affect query Q.

Example 3.2: Query Q selects accounts with balance < 500 and interest rate $> 0\%$. Update operation M sets to 2% the interest rate of all accounts with rate between 1% and 2%. The input to the algorithm is:

$$Q = \pi_{\texttt{balance,rate}}(\sigma_{\texttt{balance<500} \wedge \texttt{rate>0}}\texttt{ACCOUNT})$$
$$M = E_{upd} = \mathcal{E}[\texttt{rate}' = \texttt{2}](\sigma_{\texttt{rate>1} \wedge \texttt{rate<2}}\texttt{ACCOUNT})$$

The propagation of E_{upd} through the selection operation in Q yields insert, delete, and update operations:

$$E'_{ins} = \rho_{new}((\sigma_{\texttt{balance<500} \wedge \texttt{rate'>0}}E_{upd}) \ltimes_{\nexists} (\sigma_{\texttt{balance<500} \wedge \texttt{rate>0}}E_{upd}))$$
$$E'_{del} = \rho_{old}((\sigma_{\texttt{balance<500} \wedge \texttt{rate>0}}E_{upd}) \ltimes_{\nexists} (\sigma_{\texttt{balance<500} \wedge \texttt{rate'>0}}E_{upd}))$$
$$E'_{upd} = (\sigma_{\texttt{balance<500} \wedge \texttt{rate'>0}}E_{upd}) \bowtie (\sigma_{\texttt{balance<500} \wedge \texttt{rate>0}}E_{upd})$$

These expressions do not contain contradictory predicates, so they may be satisfiable and the propagation continues.[3] The propagation of E'_{ins}, E'_{del}, and E'_{upd} through the projection operation in Q yields:

[3]A sophisticated satisfiability check would determine that the expressions $\sigma_{\texttt{balance<500} \wedge \texttt{rate>0}}E_{upd}$ and $\sigma_{\texttt{balance<500} \wedge \texttt{rate'>0}}E_{upd}$ select the same data, thus both E'_{ins} and E'_{del} are empty.

$$E''_{ins} = \pi_{\texttt{balance,rate}} E'_{ins}$$
$$E''_{del} = \pi_{\texttt{balance,rate}} E'_{del}$$
$$E''_{upd} = \pi_{\texttt{balance,rate,rate}'} E'_{upd}$$

All three expressions may be satisfiable, thus modification M can affect the result of query Q. Furthermore, E''_{ins}, E''_{del}, and E''_{upd} describe the modifications that may be performed on Q as a result of the execution of M.

3.6 Complexity of the Algorithm

In this section we outline the time complexity of the Propagation Algorithm; a complete complexity analysis is beyond the scope of this paper. The execution time of the Propagation Algorithm depends on two additive terms:

- The time T_p required to perform the complete propagation process, disregarding the time required to test expression satisfiability. T_p depends on the number of modifications progressively generated during the propagation, which in turn depends on the type of operators in the query.

- The time T_s required to check the satisfiability of the propagated modifications. Testing the satisfiability of a single modification depends on the structure of its expression. Since the satisfiability test is repeated after each propagation step, the total time T_s depends also on the number of modifications generated during the propagation.[4]

First consider the time T_p required to perform the complete propagation process. T_p depends on the depth of the query tree. In particular, it depends on the depth D_l of the relation on which the initial modification is performed. When the query does not contain aggregates, T_p grows linearly with D_l, while in the case of queries containing aggregates, it grows exponentially with D_l. This can be understood by analyzing Tables 3–6. Consider first queries without aggregates: (a) The propagation of insert and delete operations always produces at most one insert or delete operation, thus the number of propagated modifications at each propagation step is one. (b) In the worst case (e.g., a selection node σ_p where p includes attributes modified by the incoming update operation, see Table 5 line 2), an incoming update operation propagates as one insert, one delete, and one update operation, thus the number of propagated modifications is increased by at most two at each propagation step. In both cases the time required to perform the complete propagation process grows linearly with the depth D_l of the relation initially modified. Consider now propagation through an aggregate node: (a) The propagation of insert or delete operations always produces at most a pair of either insert and update, or delete and update operations. (b) The propagation of update operations always produces at most a pair of update operations. Thus, the number of

[4]From an efficiency viewpoint, it is not obvious whether it is better to perform the satisfiability test at each propagation step, thus avoiding further propagation of unsatisfiable modifications, or to perform the test once at the end of the propagation process. Either approach is correct, and we may investigate the trade-off as future work.

propagated modifications doubles after each propagation through an aggregate node. Hence, when aggregates are present, time T_p can grow exponentially with the depth D_l of the relation initially modified, although in practice this does not pose a problem since the number of aggregates is few.

Now consider time T_s. Let N be the total number of modification expressions generated considering all steps of the complete propagation process. By the previous analysis we know that N depends on the depth of the query tree and the type of its operators. Then, $T_s = \sum_{i=1}^{N} T_e^{(i)}$, where $T_e^{(i)}$ is the time required to verify the satisfiability of the i^{th} modification expression. Each $T_e^{(i)}$ depends on several factors, including the expression's size and the number of predicates in its selection and join operators. At each propagation step, the size of the propagated expression grows linearly in the number of operators in the case of insert and delete operations and non-aggregate nodes, while it grows (at most) exponentially otherwise. For example, in the case of update operations, in the worst case it doubles its size (see, e.g., Table 5 line 2). However, observe that each propagated modification expression always includes the expression of the incoming modification (repeated twice when the size doubles). Hence, the satisfiability test can be performed "incrementally" during the propagation process, by considering only the effect of the new operators and predicates added by each propagation step.

Finally, observe that the Propagation Algorithm is used for off-line (compile-time) analysis of queries and modifications, and that the relational expressions to which it is applied are normally fairly small. Hence, even though the complexity of the Propagation Algorithm can be exponential, we expect performance to be acceptable in practice.

3.7 Soundness and Completeness

Recall from Section 3.1 that, given a query Q and a modification M as input, the Propagation Algorithm produces as output a set of modifications that characterize how the result of Q changes due to the execution of M. Intuitively, we say that the Propagation Algorithm is *complete* if, applied to an arbitrary query Q and modification M, on any database state its output operations produce all modifications on Q caused by performing the original modification M. For example, if for some database state d the execution of M causes the insertion of tuple t into $Q(d)$, then the Propagation Algorithm outputs an insert operation that produces tuple t on state d. (Recall that $Q(d)$ denotes the result of evaluating query Q on database state d.) We say that the Propagation Algorithm is *sound* if there is no state d such that its output operations produce a modification that is not performed on $Q(d)$.

We show in this section that the Propagation Algorithm is complete but not sound. We first introduce notation that allows us to formalize the intuitive definitions of soundness and completeness given above.

Definition 3.1: Let Q be a query and M a modification. Let d be an arbitrary database state. The *positive changes* Q^+ and *negative changes* Q^- to $Q(d)$ based on the execution of M are given by:

$$Q^+(M,d) = Q(d') - Q(d)$$
$$Q^-(M,d) = Q(d) - Q(d')$$

where d' is the database state obtained from d by applying modification M. $\square$

Now we can formally define soundness and completeness of the Propagation Algorithm.

Definition 3.2: The Propagation Algorithm is complete if the following holds. Let the Propagation Algorithm be applied to an arbitrary query Q and modification M and produce as output a set E^{out} of database modifications. Let $E^+(Q,M)$ and $E^-(Q,M)$ be the positive and negative modifications corresponding to E^{out}, as defined in Section 3.1. Consider an arbitrary database state d, and let $Q^+(M,d)$ and $Q^-(M,d)$ be as in Definition 3.1. Then $E^-(Q,M)(d) \supseteq Q^-(M,d)$ and $E^+(Q,M)(d) \supseteq Q^+(M,d)$, where $E^-(Q,M)(d)$ denotes the evaluation of expression $E^-(Q,M)$ on database state d (and similarly for $E^+(Q,M)$). $\square$

Definition 3.3: The Propagation Algorithm is sound if the following holds. Let the Propagation Algorithm be applied to an arbitrary query Q and modification M and produce as output a set E^{out} of modifications. Let $E^+(Q,M)$ and $E^-(Q,M)$ be the positive and negative modifications corresponding to E^{out}, as defined in Section 3.1. Consider an arbitrary database state d, and let $Q^+(M,d)$ and $Q^-(M,d)$ be as in Definition 3.1. Then $E^-(Q,M)(d) \subseteq Q^-(M,d)$ and $E^+(Q,M)(d) \subseteq Q^+(M,d)$, where $E^-(Q,M)(d)$ denotes the evaluation of expression $E^-(Q,M)$ on database state d (and similarly for $E^+(Q,M)$). $\square$

The following theorem proves the completeness of the Propagation Algorithm. The proof proceeds step-by-step for each propagation rule given in Tables 3–6, and the proof technique is analogous for all rules. Hence, we outline the proof procedure for only one propagation rule; see [Bar94] for a complete proof.

Theorem 3.1: The Propagation Algorithm, based on the propagation rules in Tables 3–6, is complete.

Proof sketch: Let the Propagation Algorithm be applied to an arbitrary query Q and modification M and produce as output a set E^{out} of modifications. Let $E^+(Q,M)$ and $E^-(Q,M)$ be the positive and negative modifications corresponding to E^{out}, as defined in Section 3.1. Consider an arbitrary database state d, and let $Q^+(M,d)$ and $Q^-(M,d)$ be as in Definition 3.1. By Definition 3.2, to prove completeness we must show that $E^-(Q,M)(d) \supseteq Q^-(M,d)$ and $E^+(Q,M)(d) \supseteq Q^+(M,d)$. The proof proceeds by induction on the depth of query tree Q. *Base case:* Q is a single node representing the modified relation R. Consider $M = E_{upd}$, so $E^+(Q,M) = \rho_{new}(E_{upd})$, $E^-(Q,M) = \rho_{old}(E_{upd})$. The execution of M produces a new database state d' in which $Q(d') = (R(d) \ltimes_{\nexists} \rho_{old}(E_{upd})(d)) \cup \rho_{new}(E_{upd})(d)$. Hence, by Definition 3.1, $Q^-(M,d) = \rho_{old}(E_{upd})(d)$ and $Q^+(M,d) = \rho_{new}(E_{upd})(d)$. Thus, $E^+(Q,M)(d) = Q^+(M,d)$ and $E^-(Q,M)(d) = Q^-(M,d)$. $M = E_{ins}$ and $M = E_{del}$ are proved similarly. *Induction step:* Let the root of Q be a unary (resp. binary) relational operator *op* over a subtree S with incoming modifications E^{in} (resp. over two subtrees S and S', of which

S has incoming modifications E^{in}). By the induction hypothesis, we assume the positive and negative modifications $E^+_{in}(S, M)$ and $E^-_{in}(S, M)$, obtained by the modifications in E^{in}, are such that $E^+_{in}(S, M)(d) \supseteq S^+(M, d)$ and $E^-_{in}(S, M)(d) \supseteq S^-(M, d)$, where S^+ and S^- are the positive and negative changes performed on the expression rooted in S as a result of executing the original modification M on d. We must show that if we apply the appropriate propagation rule for *op* to obtain E^{out} from E^{in}, then $E^+(Q, M)$ and $E^-(Q, M)$ obtained from E^{out} are such that, for all database states d, $E^+(Q, M)(d) \supseteq Q^+(M, d)$ and $E^-(Q, M)(d) \supseteq Q^-(M, d)$. As an example, let *op* be a selection σ_p performed over an arbitrary subtree S, and consider an update operation E^{in}_{upd} associated with S and performed on an attribute in p. Applying our propagation rules from the second line of Table 5, we obtain a triple $\langle E^{out}_{ins}, E^{out}_{del}, E^{out}_{upd}\rangle$, corresponding to tuples added to, deleted from, and updated in the result of $Q = \sigma_p S$. Then:

$$E^+(Q, M)(d) = \rho_{new}((\sigma_{p'} E^{in}_{upd}(d)) \ltimes_{\nexists} (\sigma_p E^{in}_{upd}(d))) \cup \rho_{new}((\sigma_{p'} E^{in}_{upd}(d)) \bowtie (\sigma_p E^{in}_{upd}(d)))$$
$$E^-(Q, M)(d) = \rho_{old}((\sigma_p E^{in}_{upd}(d)) \ltimes_{\nexists} (\sigma'_p E^{in}_{upd}(d))) \cup \rho_{old}((\sigma_{p'} E^{in}_{upd}(d)) \bowtie (\sigma_p E^{in}_{upd}(d)))$$

The execution of M produces a new database state d' in which $Q(d') = \sigma_p(S(d) \ltimes_{\nexists} \rho_{old}(E^{in}_{upd})(d)) \cup \rho_{new}(E_u pd^{in})(d)$. Hence, by Definition 3.1:

$$Q^+(M, d) = (\sigma_p((S(d) \ltimes_{\nexists} \rho_{old}(E^{in}_{upd}(d)) \cup \rho_{new}(E^{in}_{upd}(d)))) \ltimes_{\nexists} (\sigma_p S(d))$$
$$Q^-(M, d) = (\sigma_p S(d)) \ltimes_{\nexists} (\sigma_p((S(d) \ltimes_{\nexists} \rho_{old}(E^{in}_{upd}(d)) \cup \rho_{new}(E^{in}_{upd}(d))))$$

After several algebraic manipulations, we obtain $E^+(Q, M)(d) \supseteq Q^+(M, d)$ and $E^-(Q, M)(d) \supseteq Q^-(M, d)$. The other propagation rules are verified similarly. $\Box$

The following theorem proves that the Propagation Algorithm is not sound. The proof gives a counterexample: a query Q, modification M, and database state d such that $E^-(Q, M)(d) \supset Q^-(M, d)$.

Theorem 3.2: The Propagation Algorithm is not sound.

Proof: Consider a database d containing a single relation R with attributes A and B, and let $R(d) = \{(1, 1), (-1, 1)\}$. By applying the Propagation Algorithm to query $Q = \pi_B R$ and delete operation $E_{del} = \sigma_{A>0} R$, we obtain the propagated modification $E'_{del} = \pi_B E_{del}$. Before the execution of E_{del}, $Q(d) = \{(1)\}$. After E_{del} is executed, in the new database state d', $R(d') = \{(-1, 1)\}$ and $Q(d') = \{(1)\}$. Then, $Q^+(E_{del}, d) = Q^-(E_{del}, d) = \emptyset$. But $E^-(Q, E_{del})(d) = \{(1)\}$. So $E^-(Q, E_{del})(d) \supset Q^-(E_{del}, d)$. $\Box$

4 Condition-Action Rules

In this section we use the Propagation Algorithm as the basis of techniques for analyzing the behavior of Condition-Action (CA) active database rules. We first define Condition-Action rules based on the algebraic language presented in Section 2, and we specify the example rules from Section 1.2 using our language. We then show how to apply the Propagation Algorithm to analyze

```
(1)   repeat until no rule has a true condition:
(2)      select a rule r with a true condition;
(3)      execute r's action
```

Figure 2: Rule processing algorithm for CA rules.

termination and confluence of CA rules; a number of examples are included. Finally, we show how the analysis can be improved by explicitly taking into account the knowledge that a rule's condition is satisfied when its action is executed.

4.1 Syntax and Semantics

A Condition-Action rule in our language is defined as $C \rightarrow A$ where:

- C states the rule's condition as an expression in our extended relational algebra of Section 2.
- A states the rule's action as a data modification operation expressed using E_{ins}, E_{del}, or E_{upd} as given in Table 2 of Section 2.1.3.[5]

When rule $C \rightarrow A$ is evaluated, the condition C is true if and only if $C - C^{old} \neq \emptyset$, where C^{old} denotes the result of C the last time the rule was evaluated during rule processing (see below). If the rule has not previously been evaluated, then $C^{old} = \emptyset$. That is, informally, the condition is true whenever the query produces "new" tuples. This is identical to the interpretation of conditions in the CA rules of, e.g., *Ariel* [Han92], *RPL* [DE89], and set-oriented adaptations of OPS5 [GP91].

The action A is a normal data modification operation executed on the current database state. In some active database systems, e.g., [GP91,Han92], a rule's action implicitly operates only on the data selected by the condition, rather than on the entire database. We could use a similar rule model here, but it would complicate the syntax and semantics and has no bearing on our analysis methods; see Section 7 for further discussion.

Rule processing is invoked after some set of user or application modifications to the database. The basic algorithm for rule processing is given in Figure 2. Rule processing is an iterative loop in which, at each iteration, a rule with a true condition is selected (step (2) in Figure 2) and its action is executed on the current database state (step (3)). Rule processing continues until no rule has a true condition (step (1)).

In this paper, we do not consider the effect of a *conflict resolution policy* for selecting among multiple rules with true conditions [WC96]. However, as an extension to our framework it is possible to incorporate conflict resolution using rule priorities; see Section 7. Note also that the

[5]For simplicity, we consider rules with a single action here, although many active database rule languages allow rules with a sequence of actions. Our methods easily extend to multiple actions, usually simply by applying the method once for each action [Bar94].

"granularity" of rule processing invocation with respect to user modifications [WC96] is not relevant here in the context of rule analysis.

4.2 Examples

In the following we give the algebraic representation of the six rules described in Section 1.2.

r_1: Rule `bad-account` is expressed in our language as $C \rightarrow E_{upd}$ where

$$\begin{aligned} C &= \pi_{\texttt{balance,rate}}(\sigma_{\texttt{balance<500} \wedge \texttt{rate>0}}\texttt{ACCOUNT}) \\ E_{upd} &= \mathcal{E}[\texttt{rate}' = \texttt{0}]E_c \\ E_c &= \sigma_{\texttt{balance<500} \wedge \texttt{rate>0}}\texttt{ACCOUNT} \end{aligned}$$

r_2: Rule `raise-rate` is expressed in our language as $C \rightarrow E_{upd}$ where

$$\begin{aligned} C &= \pi_{\texttt{rate}}(\sigma_{\texttt{rate>1} \wedge \texttt{rate<2}}\texttt{ACCOUNT}) \\ E_{upd} &= \mathcal{E}[\texttt{rate}' = \texttt{2}]E_c \\ E_c &= \sigma_{\texttt{rate>1} \wedge \texttt{rate<2}}\texttt{ACCOUNT} \end{aligned}$$

r_3: Rule `SF-bonus` is expressed in our language as $C \rightarrow E_{upd}$ where

$$\begin{aligned} C &= \pi_{\texttt{city,C}}(\sigma_{\texttt{C>1000}}(\mathcal{A}[\texttt{C} = \texttt{count(name)}](\sigma_{\texttt{city='SF'}}\texttt{CUSTOMER}))) \\ E_{upd} &= \mathcal{E}[\texttt{rate}' = \texttt{rate} + \texttt{1}]E_c \\ E_c &= (\sigma_{\texttt{balance>5000} \wedge \texttt{rate<3}}\texttt{ACCOUNT}) \ltimes (\sigma_{\texttt{city='SF'}}\texttt{CUSTOMER}) \end{aligned}$$

r_4: Rule `start-bad` is expressed in our language as $C \rightarrow E_{ins}$ where

$$\begin{aligned} C &= \pi_{\texttt{num,balance}}((\sigma_{\texttt{balance<500}}\texttt{ACCOUNT}) \ltimes_{\nexists} (\sigma_{\texttt{end=null}}\texttt{LOW-ACC})) \\ E_{ins} &= \mathcal{E}[\texttt{start} = \texttt{today()}]\mathcal{E}[\texttt{end} = \texttt{null}] \\ &\quad\; \pi_{\texttt{num}}((\sigma_{\texttt{balance<500}}\texttt{ACCOUNT}) \ltimes_{\nexists} (\sigma_{\texttt{end=null}}\texttt{LOW-ACC})) \end{aligned}$$

and `today()` is a system-defined function returning the current date.

r_5: Rule `end-bad` is expressed in our language as $C \rightarrow E_{upd}$ where

$$\begin{aligned} C &= \pi_{\texttt{num,end}}((\sigma_{\texttt{end=null}}\texttt{LOW-ACC}) \ltimes (\sigma_{\texttt{balance}\geq\texttt{500}}\texttt{ACCOUNT})) \\ E_{upd} &= \mathcal{E}[\texttt{end}' = \texttt{today()}]E_c \\ E_c &= (\sigma_{\texttt{end=null}}\texttt{LOW-ACC}) \ltimes (\sigma_{\texttt{balance}\geq\texttt{500}}\texttt{ACCOUNT}) \end{aligned}$$

r_6: Rule `decrease-bad` is expressed in our language as $C \rightarrow E_{upd}$ where

$$\begin{aligned} C &= \pi_{\texttt{num}}((\sigma_{\texttt{rate>1} \wedge \texttt{balance}\geq\texttt{500} \wedge \texttt{balance}\leq\texttt{1000}}\texttt{ACCOUNT}) \ltimes \\ &\quad\; (\sigma_{\texttt{D>50}}\mathcal{A}[\texttt{D} = \texttt{sum(end} - \texttt{start)}; \texttt{num}]\texttt{LOW-ACC})) \\ E_{upd} &= \mathcal{E}[\texttt{rate}' = \texttt{1}]E_c \\ E_c &= (\sigma_{\texttt{rate>1} \wedge \texttt{balance}\geq\texttt{500} \wedge \texttt{balance}\leq\texttt{1000}}\texttt{ACCOUNT}) \ltimes \\ &\quad\; (\sigma_{\texttt{D>50}}\mathcal{A}[\texttt{D} = \texttt{sum(end} - \texttt{start)}; \texttt{num}]\texttt{LOW-ACC}) \end{aligned}$$

4.3 Termination Analysis

Recall the rule processing loop in Figure 2. Termination for a rule set is guaranteed if rule processing always reaches a state in which no rule has a true condition. Notice that, according to the semantics in Section 4.1, after the first execution of each rule r, r's condition is true again if and only if new data satisfies its condition. Hence, informally, rule processing does not terminate if and only if rules provide new data to each other indefinitely.

We say that a rule r_i may activate a rule r_j if executing r_i's action may cause new data to satisfy r_j's condition. More precisely:

Definition 4.1: Consider two rules $r_i : C_i \rightarrow A_i$ and $r_j : C_j \rightarrow A_j$. Let C_j^{old} denote the result of C_j the last time r_j was evaluated during rule processing, and let $C_j^{old} = \emptyset$ if r_j has not been evaluated previously. r_i *may activate* r_j if the execution of action A_i can change the database from a state in which $C_j - C_j^{old} = \emptyset$ to a state in which $C_j - C_j^{old} \neq \emptyset$. □

We analyze termination by building an *Activation Graph*. In the graph, nodes represent rules, and directed edges indicate that one rule may activate the other. If there are no cycles in the graph, then rule processing is guaranteed to terminate [AHW95,BCW93]. Hence, the core of termination analysis is determining when an edge should be included in the graph, i.e., when one rule may activate another rule. The more accurately we can make this decision, the more accurately we can analyze termination.

We use our Propagation Algorithm to decide when an edge $r_i \rightarrow r_j$ belongs in the Activation Graph. Note that rules may activate themselves, so $r_i = r_j$ is included in the analysis. To determine if r_i may activate r_j, we apply the Propagation Algorithm to r_j's condition C and r_i's action A. If the Propagation Algorithm yields *insert* or *update* operations, then the execution of r_i may result in new data satisfying r_j's condition. Thus, r_i may activate r_j, and the edge $r_i \rightarrow r_j$ belongs in the graph. If only *delete* operations or no operations are produced by the Propagation Algorithm, then the execution of r_i cannot result in new data for r_j's condition, and the edge is not included in the graph.

Our use of the Activation Graph is similar to, e.g., [AHW95,CW90], but our approach is far less conservative since we exploit the algebraic structure of conditions and actions to accurately determine when edges belong in the graph.

4.3.1 Examples

We apply our analysis technique to build the Activation Graph for the rule set in Section 4.2. We describe in detail the analysis of two rule pairs, then we discuss the result of applying our analysis technique to the complete rule set.

Example 4.1: Consider rules `bad-account` (r_1) and `raise-rate` (r_2). Both rule conditions reference attribute `rate` and both rule actions update `rate`. Hence, intuitively (and according to the method in [AHW95]), the two rules might activate each other indefinitely. We have shown in

Example 3.2 that r_2's action may provide data to r_1's condition (since *insert* and *update* operations are produced by the Propagation Algorithm), thus the edge $r_2 \to r_1$ belongs in the Activation Graph. Now we use the Propagation Algorithm to determine if r_1 may activate r_2. The input to the algorithm is:

$$\begin{aligned} C &= \pi_{\texttt{rate}}(\sigma_{\texttt{rate}>1 \wedge \texttt{rate}<2}\texttt{ACCOUNT}) \\ A &= E_{upd} = \mathcal{E}[\texttt{rate}' = \texttt{0}](\sigma_{\texttt{balance}<500 \wedge \texttt{rate}>0}\texttt{ACCOUNT}) \end{aligned}$$

The propagation of E_{upd} through the selection operation in C yields:

$$\begin{aligned} E'_{ins} &= \rho_{new}((\sigma_{\texttt{rate}'>1 \wedge \texttt{rate}'<2}E_{upd}) \ltimes_{\nexists} (\sigma_{\texttt{rate}>1 \wedge \texttt{rate}<2}E_{upd})) \\ E'_{upd} &= (\sigma_{\texttt{rate}'>1 \wedge \texttt{rate}'<2}E_{upd}) \bowtie (\sigma_{\texttt{rate}>1 \wedge \texttt{rate}<2}E_{upd}) \\ E'_{del} &= \rho_{old}((\sigma_{\texttt{rate}>1 \wedge \texttt{rate}<2}E_{upd}) \ltimes_{\nexists} (\sigma_{\texttt{rate}'>1 \wedge \texttt{rate}'<2}E_{upd})) \end{aligned}$$

Since predicates $\texttt{rate}' > \texttt{1}$ and (assignment) $\texttt{rate}' = \texttt{0}$ are contradictory,[6] expressions E'_{ins} and E'_{upd} are not satisfiable and hence are discarded. The propagation of E'_{del} through the projection operation in C yields:

$$E''_{del} = \pi_{\texttt{rate}}E'_{del}$$

which is satisfiable. Thus, r_1's action may result in a deletion of tuples from r_2's condition. However, since neither an *insert* nor an *update* action is produced, r_1 cannot activate r_2, and the edge $r_1 \to r_2$ is not included in the Activation Graph. Hence, our analysis correctly concludes that r_1 and r_2 cannot activate each other indefinitely.

Example 4.2: Consider rules `start-bad` (r_4) and `end-bad` (r_5). Here again, according to the method in [AHW95], the two rules might activate each other indefinitely. We use the Propagation Algorithm to determine if r_4 may activate r_5. The input to the algorithm is:

$$\begin{aligned} C = \quad & \pi_{\texttt{num},\texttt{end}}((\sigma_{\texttt{end}=\texttt{null}}\texttt{LOW-ACC}) \ltimes (\sigma_{\texttt{balance}\geq 500}\texttt{ACCOUNT})) \\ A = \quad & E_{ins} = \mathcal{E}[\texttt{start} = \texttt{today()}]\mathcal{E}[\texttt{end} = \texttt{null}](\pi_{\texttt{num}}((\sigma_{\texttt{balance}<500}\texttt{ACCOUNT}) \ltimes_{\nexists} \\ & (\sigma_{\texttt{end}=\texttt{null}}\texttt{LOW-ACC}))) \end{aligned}$$

The propagation of E_{ins} through the semijoin operation in C yields:

$$\begin{aligned} E'_{ins} = \quad & (\mathcal{E}[\texttt{start} = \texttt{today()}]\mathcal{E}[\texttt{end} = \texttt{null}](\pi_{\texttt{num}}(\textit{low-bal} \ltimes_{\nexists} \\ & (\sigma_{\texttt{end}=\texttt{null}}\texttt{LOW-ACC})))) \ltimes \textit{high-bal} \end{aligned}$$

where *low-bal* is an abbreviation for $\sigma_{\texttt{balance}<500}\texttt{ACCOUNT}$ and *high-bal* is an abbreviation for $\sigma_{\texttt{balance}\geq 500}\texttt{ACCOUNT}$. This expression is not satisfiable, since it requires a tuple with a given `num` value to satisfy both predicates $\texttt{balance} < \texttt{500}$ and $\texttt{balance} \geq \texttt{500}$. Hence, r_4 cannot activate

[6] It is trivial to recognize syntactically that these predicates should be considered together in the satisfiability check.

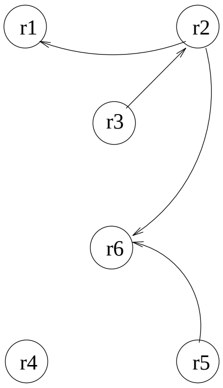

Figure 3: Activation Graph

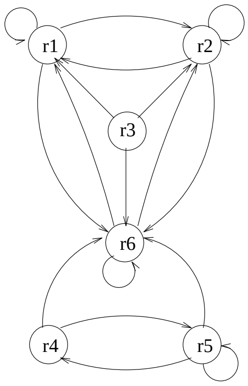

Figure 4: Triggering Graph

r_5, and the edge $r_4 \rightarrow r_5$ is not included in the Activation Graph. We conclude that r_4 and r_5 cannot activate each other indefinitely.

Figure 3 shows the Activation Graph resulting from applying our analysis technique to the complete rule set presented in Section 4.2. By contrast, Figure 4 shows the corresponding Triggering Graph obtained by applying the technique in [AHW95,CW90].[7] Observe that the Activation Graph generated with our technique does not contain any cycles—we correctly determine that the rules cannot activate each other indefinitely. However, in Figure 4 there are numerous cycles. Thus, by applying our technique it is possible to conclude that rule execution terminates for this rule set, while the technique in [AHW95] is unable to determine the same property.

4.4 Confluence Analysis

Recall again the rule processing loop in Figure 2. In each iteration there may be multiple rules eligible for execution, since more than one rule may have a true condition. A rule set is *confluent* if the final state of the database does not depend on which eligible rule is chosen for execution at

[7] To apply the analysis technique in [AHW95,CW90] to CA rules, we must first generate all the events that may activate the condition. This step can be performed using, e.g., the technique described in [CW90].

any iteration.

To formally describe confluence and confluence analysis, we introduce the notion of a *rule execution state* and a *rule execution sequence*. Let R be the set of rules under consideration.

Definition 4.2: A *rule execution state* S is a pair (d, R_A), where d is a database state and $R_A \subseteq R$ is a set of activated rules. $\square$

Definition 4.3: A *rule execution sequence* is a sequence σ consisting of a series of rule execution states linked by (executed) rules. A rule execution sequence is *complete* if the last state is $(d, \emptyset)$, i.e., the last state has no activated rules. A rule execution sequence is *valid* if it represents a correct execution sequence: only activated rules are executed, and pairs of adjacent states properly represent the effect of executing the corresponding rule; for details see [AHW95,Bar94]. $\square$

We now define confluence in terms of execution sequences.

Definition 4.4: A rule set is *confluent* if, for every initial rule execution state S (produced by an initial database state followed by a set of user modifications), every valid and complete rule execution sequence beginning with S has the same final state. $\square$

Clearly we cannot use this definition directly to analyze confluence, since it requires the exhaustive verification of all possible execution sequences for all possible initial states. We give sufficient conditions for confluence based on the *commutativity* of rule pairs. Two rules r_i and r_j commute if, starting with any rule execution state S, executing r_i followed by r_j produces the same rule execution state as executing r_j followed by r_i. We formalize the concepts of rule "deactivation" and commutativity of rule actions, then we give conditions for rule commutativity.

We say that a rule r_i may deactivate a rule r_j when r_i's action may delete all data that satisfied r_j's condition. More precisely:

Definition 4.5: Consider two rules $r_i : C_i \rightarrow A_i$ and $r_j : C_j \rightarrow A_j$. Let C_j^{old} denote the result of C_j the last time r_j was evaluated during rule processing, and let $C_j^{old} = \emptyset$ if r_j has not been evaluated previously. r_i *may deactivate* r_j if the execution of action A_i can change the database from a state in which $C_j - C_j^{old} \neq \emptyset$ to a state in which $C_j - C_j^{old} = \emptyset$. $\square$

We also define when two rule actions commute:

Definition 4.6: Let A_i and A_j be two data modification operations (i.e., rule actions). A_i and A_j commute iff, for all database states, the execution of A_i followed by A_j and the execution of A_j on followed by A_i produce the same final database state. $\square$

We state sufficient conditions for rule commutativity in the following Lemma, whose proof is straightforward and only sketched here. For a complete proof see [Bar94].

Lemma 4.1: Distinct rules r_i and r_j commute if: (1) r_i cannot activate r_j; (2) r_i cannot deactivate r_j; (3) conditions (1) and (2) with i and j reversed; (4) r_i's action and r_j's action commute.

Proof sketch: Let $S_1 = (d_1, R_{A1})$ be an arbitrary rule execution state in which rules $r_i, r_j \in R_{A1}$. We must prove that if conditions (1)–(4) hold, then rule execution sequences $\sigma = S_1 \xrightarrow{r_i} S' \xrightarrow{r_j} S_2$ and $\overline{\sigma} = S_1 \xrightarrow{r_j} S'' \xrightarrow{r_i} \overline{S}_2$ yield the same final execution state, i.e., $S_2 = \overline{S}_2$. Let $S_2 = (d_2, R_{A2})$ and $\overline{S}_2 = (\overline{d}_2, \overline{R}_{A2})$. $S_2 = \overline{S}_2$ if $R_{A2} = \overline{R}_{A2}$ and $d_2 = \overline{d}_2$. $R_{A2} = \overline{R}_{A2}$ is guaranteed if: (a) execution of rule r_i does not reactivate r_j after r_j has already executed (guaranteed by condition (1)), (b) execution of rule r_i does not prevent r_j from executing (guaranteed by condition (2)), (c) same reversing r_i and r_j (guaranteed by condition (3)), and (d) the set of rules activated and deactivated by the execution of r_i and r_j does not depend on their execution order (condition (4) guarantees this by guaranteeing that the rule actions commute). Finally, $d_2 = \overline{d}_2$ is guaranteed by condition (4), which states that the order in which the two rules' actions are executed does not affect the final database state. □

Note that even though conditions (1)–(4) are not necessarily satisfied when $r_i = r_j$, it is the case that a rule always commutes with itself.

We now prove two Lemmas, followed by the main Theorem on confluence. The first Lemma states, under the assumption of commutative rules, that two execution sequences with the same initial state and executed rules have the same final state. The second Lemma states, again under the assumption of commutativity, that two sequences with the same initial state must have the same executed rules.

Lemma 4.2: Let all pairs of rules in R commute. Let σ_1 and σ_2 be two valid and complete rule execution sequences with the same initial state, such that the same rules are executed in σ_1 and σ_2 although not necessarily in the same order. Then σ_1 and σ_2 have the same final state.

Proof: Since σ_1 and σ_2 have the same executed rules, we can "permute" σ_1 so that its rules are considered in the same order as σ_2. We exchange adjacent rules in σ_1 one pair at a time; with each exchange, there is no change to the outer two execution states due to commutativity. Hence, since σ_1 and σ_2 have the same initial state, they must have the same final state. □

Lemma 4.3: Let all pairs of rules in R commute. Let σ_1 and σ_2 be two valid and complete rule execution sequences with the same initial state. Then the same rules are executed in σ_1 and σ_2.

Proof: We again use commutativity to permute sequences without affecting outer execution states. In each sequence, we exchange rules one pair at a time until the rules appear in "sorted" order according to some criterion (the criterion is irrelevant as long as the same criterion is used for σ_1 and σ_2). Suppose, for the sake of a contradiction, that σ_1 and σ_2 have different executed rules, and consider the first point of divergence, i.e., where a rule r appears in σ_1 but a different rule r' appears in σ_2. Let S be the execution state preceding these rules; S is the same in σ_1 and σ_2, so r and r' are both activated in S. Without loss of generality, assume that r precedes r' in the sorted order. Then r cannot appear in σ_2 beyond S. Consequently, execution of some rule other than r in σ_2 must deactivate r. But this contradicts condition (1) of commutativity. □

Based on these Lemmas, the following Theorem presents a sufficient condition to guarantee confluence of a rule set.

Theorem 4.1: A rule set R is confluent if all pairs of rules in R commute.

Proof: Suppose all rule pairs commute, and consider two valid and complete execution sequences σ_1 and σ_2 with the same initial state. By Lemma 4.3, σ_1 and σ_2 have the same set of executed rules. Then by Lemma 4.2, σ_1 and σ_2 have the same final state. $\square$

The requirement for confluence in Theorem 4.1 may seem rather strong, but there is no way to weaken this requirement without a more sophisticated conflict resolution policy or priorities among rules. We believe this argues for the importance of rule priorities. Notice also that, in the case where no rule can activate itself, the confluence requirement as stated in Theorem 4.1 trivially implies termination, since the pairwise commutativity of all rules includes the requirement that no rule activates another rule. However, if one or more rules can activate themselves, then confluence does not imply termination.

Commutativity of rule pairs forms the basis of most methods for analyzing confluence of database rules, e.g., [AHW95, vdVS93]. The remainder of this section describes our technique for determining commutativity of rule pairs. Needless to say, we use our Propagation Algorithm to analyze commutativity, exploiting the algebraic description of rule conditions and actions to yield a much more accurate analysis technique than, e.g., [AHW95].

4.4.1 Analyzing Commutativity

To guarantee commutativity of two rules r_i and r_j, we must verify conditions (1)–(4) in Lemma 4.1 above. For (1), we determine that r_i cannot activate r_j exactly as we have done for termination; recall Section 4.3. To verify condition (2), which requires that r_i cannot deactivate r_j, we must show that r_i's action A_i cannot "take away" data from r_j's condition C_j. It is easy to see that action A_i can take away data from condition C_j only if the Propagation Algorithm applied to A_i and C_j produces a *delete* operation. Hence, one application of the Propagation Algorithm is sufficient for verifying (1) and (2). For (3), we reverse the roles of r_i and r_j in the analysis of (1) and (2).

For (4), we must determine if r_i's action A_i can change the effect of r_j's action A_j and vice-versa. We first transform action A_j into a query C_{A_j} such that if the result of query C_{A_j} cannot be affected by the execution of A_i, then A_i cannot change the effect of action A_j. We then apply the Propagation Algorithm to analyze A_i and C_{A_j}: if the algorithm produces $\emptyset$, then A_i cannot change the effect of A_j; if the algorithm produces one or more *insert*, *delete*, or *update* operations, then A_i may change the effect of A_j. Then, we reverse the roles of A_i and A_j. If again the algorithm produces $\emptyset$, then A_i and A_j commute.

Consider how query C_{A_j} is derived from action A_j. If A_j is an insert operation, then $A_j = E_{ins}$ is a query representing the inserted data, hence we let $C_{A_j} = E_{ins}$. Similarly, if A_j is a delete operation, then $A_j = E_{del}$ is a query representing the deleted data, and we let $C_{A_j} = E_{del}$. Suppose A_j is an update operation on attribute A, defined by $E_{upd} = \mathcal{E}[A' = expr]E_c$.[8] We start with the "selection condition" E_c. C_{A_j} is the projection of E_c onto all attributes referenced within E_c,

[8]The extension to multiple updated attributes is straightforward.

together with all attributes referenced in the $\mathcal{E}$ operation (both A and the attributes referenced in $expr$). If any of these attributes can be affected by the execution of A_i, then A_i may change the effect of A_j's update; if not, then A_i cannot change the effect of A_j. By using the projection here, rather than the entire expression E_c, we ignore modifications to attributes that do not affect the evaluation of E_c or the assignment of the new values to the updated attribute.

4.4.2 Examples

We apply our technique to analyze rule commutativity for the rule set defined in Section 4.2. We describe in detail the analysis of two rule pairs, then we discuss the result of applying our analysis technique to the complete rule set.

Example 4.3: Consider rules `bad-account` (r_1) and `SF-bonus` (r_3). We first analyze the effect of r_1's action on r_3. Since r_3's condition does not reference the relation updated by r_1, r_1's action trivially cannot affect r_3's condition. We use the Propagation Algorithm to analyze the effect of r_1's action on the query corresponding to r_3's action: $\pi_{\texttt{balance,rate,name,city}} E_c$. The input to the algorithm is:

$$C = \pi_{\texttt{balance,rate,name,city}}(\sigma_{\texttt{balance>5000}\wedge\texttt{rate<3}}(\texttt{ACCOUNT} \ltimes (\sigma_{\texttt{city='SF'}}\texttt{CUSTOMER})))$$
$$A = E_{upd} = \mathcal{E}[\texttt{rate}' = \texttt{0}](\sigma_{\texttt{balance<500}\wedge\texttt{rate>0}}\texttt{ACCOUNT})$$

The propagation of E_{upd} through the semijoin operation in C yields:

$$E'_{upd} = E_{upd} \ltimes (\sigma_{\texttt{city='SF'}}\texttt{CUSTOMER})$$

The propagation of E'_{upd} through the selection operation in C yields:

$$E''_{ins} = \rho_{new}((\sigma_{\texttt{balance>5000}\wedge\texttt{rate'<3}} E'_{upd}) \ltimes_{\nexists} (\sigma_{\texttt{balance>5000}\wedge\texttt{rate<3}} E'_{upd}))$$
$$E''_{del} = \rho_{old}((\sigma_{\texttt{balance>5000}\wedge\texttt{rate<3}} E'_{upd}) \ltimes_{\nexists} (\sigma_{\texttt{balance>5000}\wedge\texttt{rate'<3}} E'_{upd}))$$
$$E''_{upd} = (\sigma_{\texttt{balance>5000}\wedge\texttt{rate'<3}} E_{upd}) \bowtie (\sigma_{\texttt{balance>5000}\wedge\texttt{rate<3}} E_{upd})$$

In all three expressions, predicates `balance` $>$ `5000` and `balance` $<$ `500` (the latter from E_{upd}) are contradictory, so the expressions are unsatisfiable. Hence, the Propagation Algorithm produces no actions and we conclude that executing r_1's action cannot change the effect of r_2's action.

A similar analysis reveals that r_3's action cannot affect r_1's action, and we have already shown in Example 3.1 that r_3's action cannot affect r_1's condition. Hence, we conclude that rules r_1 and r_3 commute.

Example 4.4: Consider rules `start-bad` (r_4) and `end-bad` (r_5). We have already shown in Example 4.2 that r_4's action cannot affect r_5's condition. An analogous analysis shows that r_4's action cannot affect the query corresponding to r_5's action: $\pi_{num,end} E_c$. Consider the effect of rule r_5 on rule r_4. We first apply the Propagation Algorithm to r_5's action and r_4's condition. The input to the algorithm is:

	r_1	r_2	r_3	r_4	r_5	r_6
r_1	Y					
r_2	N	Y				
r_3	N	N	Y			
r_4	Y	Y	Y	Y		
r_5	Y	Y	Y	N	Y	
r_6	N	N	N	N	N	Y

(a)

	r_1	r_2	r_3	r_4	r_5	r_6
r_1	Y					
r_2	N	Y				
r_3	Y	N	Y			
r_4	Y	Y	Y	Y		
r_5	Y	Y	Y	Y	Y	
r_6	Y	N	Y	Y	N	Y

(b)

Figure 5: Commutativity Analysis

$$C = \pi_{\texttt{num},\texttt{balance}}((\sigma_{\texttt{balance}<500}\texttt{ACCOUNT}) \ltimes_{\nexists} (\sigma_{\texttt{end}=\texttt{null}}\texttt{LOW-ACC}))$$
$$A = E_{upd} = \mathcal{E}[\texttt{end}' = \texttt{today}()]((\sigma_{\texttt{end}=\texttt{null}}\texttt{LOW-ACC}) \ltimes (\sigma_{\texttt{balance}\geq 500}\texttt{ACCOUNT}))$$

The propagation of E_{upd} through the selection operation yields:

$$E'_{ins} = \rho_{new}((\sigma_{\texttt{end}'=\texttt{null}}E_{upd}) \ltimes_{\nexists} (\sigma_{\texttt{end}=\texttt{null}}E_{upd}))$$
$$E'_{upd} = (\sigma_{\texttt{end}'=\texttt{null}}E_{upd}) \bowtie (\sigma_{\texttt{end}=\texttt{null}}E_{upd})$$
$$E'_{del} = \rho_{old}((\sigma_{\texttt{end}=\texttt{null}}E_{upd}) \ltimes_{\nexists} (\sigma_{\texttt{end}'=\texttt{null}}E_{upd}))$$

E'_{ins} and E'_{upd} are unsatisfiable, due to the contradiction of predicates $\texttt{end}' = \texttt{null}$ and (assignment) $\texttt{end}' = \texttt{today}()$. The propagation of E'_{del} through the $\ltimes_{\nexists}$ operation in C yields:

$$E''_{ins} = \textit{low-bal} \ltimes ((\sigma_{\texttt{end}=\texttt{null}}\texttt{LOW-ACC}) \ltimes \textit{high-bal})$$

where *low-bal* is an abbreviation for $\sigma_{\texttt{balance}<500}\texttt{ACCOUNT}$ and *high-bal* is an abbreviation for $\sigma_{\texttt{balance}\geq 500}\texttt{ACCOUNT}$. This expression is not satisfiable, since it requires a tuple with a given `num` value to satisfy both predicates $\texttt{balance} < 500$ and $\texttt{balance} \geq 500$. Thus, r_5's action cannot affect r_4's condition. An analogous analysis shows that r_5's action cannot affect the query corresponding to r_4's action, E_{ins}. Hence, we conclude that rules r_4 and r_5 commute.

The result of the pairwise commutativity analysis of the entire rule set from Section 4.2 is presented in Figure 5. Figure 5(a) shows commutative (Y) and non-commutative (N) rule pairs obtained by applying the technique in [AHW95], while Figure 5(b) shows the result obtained by applying our Propagation Algorithm as described above. Note that by using our technique it is possible to determine the commutativity of several rule pairs whose commutativity could not be determined with the method in [AHW95]. Nevertheless, it is the case that some rule pairs do not commute, and the rule set is not confluent.

4.5 Improving the Analysis

In this section we describe improvements to the termination and confluence analysis techniques presented in the previous sections. Recall that a basic step in analyzing both termination and confluence is determining whether one rule can activate or deactivate another rule. This test is performed by applying the Propagation Algorithm to a rule r_1's action and a rule r_2's condition. The test can be made more accurate by taking into account the fact that, during rule processing, when r_1's action is executed r_1's condition must be true. This improvement is described in Section 4.5.1. In addition, recall that analyzing confluence of a rule set requires checking commutativity for all pairs of rules (Section 4.4). Note, however, that if two rules r_1 and r_2 can never be eligible for execution at the same time (because their conditions are mutually exclusive), or if one action is guaranteed to have no effect when both rules are eligible, then we need not guarantee commutativity for these rules. This improvement is described in Section 4.5.2.

4.5.1 Activation and Deactivation

Recall from Section 3 that the Propagation Algorithm determines the effect on a query Q of executing a modification M. In performing termination and confluence analysis, we have used the Propagation Algorithm to determine the effect on a rule's condition of executing a rule's action. From the rule processing algorithm in Figure 2 we know that when a rule's action is executed its condition must be true, and we can exploit this knowledge in our analysis. More specifically, consider two rules $r_1 : C_1 \rightarrow A_1$ and $r_2 : C_2 \rightarrow A_2$. We extend our application of the Propagation Algorithm to determine the effect of performing action A_1 on condition C_2 when A_1 is executed on a database state in which condition C_1 is true (and similarly for A_2 and C_1). The knowledge that C_1 is true when A_1 executes may be used to determine that some of the actions obtained by the propagation of A_1 on C_2 are not satisfiable. Hence, it is sometimes possible to determine that, in the context of rule processing, r_1 cannot activate (resp. deactivate) r_2, even though action A_1 may cause condition C_2 to become true (resp. false) in the general case.

To make this improvement, we first observe a fact about rule condition evaluation (recall Section 4.1). Informally, a rule's condition C is true whenever it produces tuples that were not produced at its last evaluation during rule processing, i.e., it produces "new" tuples. Note that the "new" tuples produced by C are a subset of the tuples C would produce if it were evaluated on the current database state.[9] Hence, if condition C is true on "new" data, it is certainly true on the current database state as well.

The improved application of the Propagation Algorithm is presented in Figure 6. This algorithm takes as input a query C_2, a modification A_1, and a query C_1 that is known to be non-empty when A_1 is executed. Figure 6 uses the Propagation Algorithm described in Section 3 as a subroutine, denoted as `PA`. In step (1), the Propagation Algorithm is applied to query C_2 and modification A_1 to produce a set of modifications A'. We then reduce the set A' by discarding modifications that are unsatisfiable when C_1 is non-empty. Steps (2)–(3) check applicability conditions for discarding modifications. First, if modification A_1 contains query C_1, using the classical notion of *query containment* [Ull89], then the result obtained by applying the Propagation Algorithm cannot be

[9] This property follows from the fact that in our algebraic language negation is not allowed at the outermost level of a relational expression (see Example 4.5 below).

```
(1)     A'= PA(C2,A1)
(2)     if not contains(CA1,C1) then
(3)        if PA(C1,A1) does not include delete modifications
(4)           for each Ai' in A' do
(5)               if contains(not(C1),CAi) then A' = A' - Ai'
```

Figure 6: Improved activation and deactivation analysis

reduced. (This case occurs when, e.g., a rule's action modifies all the data selected by the rule's condition, a relatively common scenario.) This case is checked in step (2), where C_{A_1} is the query obtained from modification A_1 as described in Section 4.4.1. Known algorithms [LS93] are applied for the containment test, which is denoted as subroutine `contains`.[10] Second, we cannot discard modifications if the execution of A_1 may cause the result of query C_1 to become empty. This case is checked in step (3) by applying the Propagation Algorithm to C_1 and A_1 and verifying that the output does not include delete modifications. Steps (4)–(5) perform the reduction. They are based on the fact that a modification $A'_i \in A'$ has no effect if the condition C_{A_i} obtained from A'_i (Section 4.4.1) is not satisfiable. Thus, if $C_1 \neq \emptyset \Rightarrow C_{A_i} = \emptyset$, then A'_i will have no effect and it is eliminated from the output set A'. Checking if $C_1 \neq \emptyset \Rightarrow C_{A_i} = \emptyset$ holds can again use query containment: $C_1 \neq \emptyset \Rightarrow C_{A_i} = \emptyset$ when the negation of query C_1 contains query C_{A_i}. Step (5) performs the containment test and is repeated for all modifications in A'_1.

Example 4.5: Consider the following queries and modification that use our example relations from Section 1.2:

$$C_1 = \{1\} \ltimes_{\nexists} \pi_1(\{1\} \times (\sigma_{\mathtt{num}>10}\mathtt{ACCOUNT}))$$
$$A_1 = E_{del} = (\sigma_{\mathtt{num}>10}\mathtt{LOW\text{-}ACC})$$
$$C_2 = \mathtt{ACCOUNT} \ltimes_{\nexists} \mathtt{LOW\text{-}ACC}$$

where $\{1\}$ is a constant relation containing only the element 1. Query C_1 is satisfied if there are no tuples with `num` $>$ `10` in relation `ACCOUNT`. (The relational expression for this query illustrates how negation at the outermost level can be represented in our algebraic language using cartesian product and not-exists semijoin.) Modification A_1 deletes from relation `LOW-ACC` all tuples with `num` $>$ `10`. Query C_2 selects all accounts in relation `ACCOUNT` for which there are no corresponding tuples in relation `LOW-ACC`. We will show how the improved application of the Propagation Algorithm exploits the knowledge that C_1 is non-empty to determine that A_1 cannot affect C_2; refer to Figure 6.

The Propagation Algorithm applied to C_2 and A_1 (step (1)) yields:

$$A'_1 = E'_{ins} = \mathtt{ACCOUNT} \ltimes (\sigma_{\mathtt{num}>10}\mathtt{LOW\text{-}ACC})$$

[10] Although containment tests can have exponential complexity [Ull89], we expect to be considering relatively small expressions here.

The condition associated with modification E_{del} is the expression E_{del} itself. It is trivial to see that $C_1 \not\subseteq E_{del}$ (step (2)). Further, the application of the Propagation Algorithm to C_1 and A_1 yields $\emptyset$ (step (3)). The condition associated with E'_{ins} is E'_{ins} itself. The negated version of C_1 contains E'_{ins}. Hence, action E'_{ins} is unsatisfiable (steps (4)–(5)) and is discarded. Thus, modification A_1, executed from a state in which C_1 is non-empty, cannot affect query C_2.

4.5.2 Commutativity

To analyze confluence of a rule set, each pair of rules in the set is tested for commutativity by verifying the conditions described in Section 4.4. However, it is not necessary for commutativity to hold if two rules cannot be eligible for execution together, i.e., if their conditions cannot be true at the same time. Furthermore, even if both rule conditions are true, it may be that one rule's action is guaranteed to have no effect when the other rule's condition is true. Here again it is not necessary to test commutativity of the rule pair.

Consider two rules $r_1 : C_1 \to A_1$ and $r_2 : C_2 \to A_2$. It is not necessary to perform the commutativity test for r_1 and r_2 when either: (1) C_1 and C_2 cannot be true at the same time, or (2) if C_1 and C_2 are both true, then one of A_1 or A_2 has no effect. Condition (1) can be verified by checking if either $C_1 \neq \emptyset \Rightarrow C_2 = \emptyset$ or $C_2 \neq \emptyset \Rightarrow C_1 = \emptyset$. Similar to Section 4.5.1, $C_1 \neq \emptyset \Rightarrow C_2 = \emptyset$ can be checked using query containment: the implication holds if the negation of query C_1 contains query C_2; $C_2 \neq \emptyset \Rightarrow C_1 = \emptyset$ is tested analogously. Checking condition (2) is similar to the improvement described in Section 4.5.1: we verify if either $C_1 \neq \emptyset \Rightarrow C_{A_2} = \emptyset$ or $C_2 \neq \emptyset \Rightarrow C_{A_1} = \emptyset$, where C_{A_1} and C_{A_2} are the conditions corresponding to A_1 and A_2, respectively. Consider $C_1 \neq \emptyset \Rightarrow C_{A_2} = \emptyset$. As in Section 4.5.1, we must first check that the execution of A_1 cannot cause query C_1 to become empty. We do this by applying the Propagation Algorithm to C_1 and A_1 and verifying that its output does not include delete actions. Then, $C_1 \neq \emptyset \Rightarrow C_{A_2} = \emptyset$ holds if the negation of query C_1 contains query C_{A_2}. $C_2 \neq \emptyset \Rightarrow C_{A_1} = \emptyset$ is verified similarly.

Example 4.6: Consider two rules r_1 and r_2 with the following conditions using our example relations from Section 1.2:

$$C_1 = \sigma_{\texttt{rate}>3}\texttt{ACCOUNT}$$
$$C_2 = \{1\} \ltimes_{\nexists} \pi_1(\{1\} \times (\sigma_{\texttt{rate}>4}\texttt{ACCOUNT}))$$

where $\{1\}$ is a constant relation containing only the element 1. Query C_1 selects all accounts in relation `ACCOUNT` with rate > 3. Query C_2 is non-empty if there are no tuples with rate > 4 in relation `ACCOUNT`. Clearly, the negation of C_1 contains C_2. Thus, in analyzing confluence of a rule set containing r_1 and r_2, commutativity is not required for this rule pair.

5 Event-Condition-Action Rules

An Event-Condition-Action rule (or ECA rule) has a set of *events*, an optional *condition*, and an *action*. An ECA rule is *triggered*, i.e., is eligible for evaluation, when any event specified in the rule's event set occurs. If the rule includes a condition, the condition must be true on the current database state for the rule's action to be executed. Due to the notion of triggering events in ECA rules, and due to a different semantics for condition evaluation ("current state" versus "new data"), it is not possible to apply directly the analysis techniques described in Section 4 for CA rules. However, it is possible to identify a common class of ECA rules, which we call *quasi-CA* rules, to which our analysis techniques can be applied. For ECA rules that are not quasi-CA, previously known weaker techniques can be used [AHW95].

In this section we first describe the syntax and semantics of the ECA rule language we consider, based on the algebra presented in Section 2. Then we discuss the differences between ECA rules and CA rules, and we characterize quasi-CA rules. The example rule set from Section 1.2 is presented using the ECA rule language, and for each rule we show how to determine whether it is quasi-CA. Finally, a general technique for analyzing termination and confluence of ECA rules is presented. This technique exploits our Propagation Algorithm based analysis when possible, and degenerates to a weaker analysis similar to [AHW95] otherwise.

5.1 Syntax and Semantics

An Event-Condition-Action rule in our language is defined as $\{T\} : C \rightarrow A$ where:

- $\{T\}$ denotes the set of triggering events.
- C states the rule's condition as an expression (query) in our extended relational algebra.
- A states the rule's action as a data modification operation expressed using E_{ins}, E_{del}, or E_{upd} as given in Table 2.

The possible triggering events correspond to data modifications: *ins* R (insert into R), *del* R (delete from R), and *upd* $R.A$ (update R's attribute A).[11] A rule is triggered when any of its triggering events occurs. When a triggered rule is evaluated, its condition C is true if and only if $C \neq \emptyset$ on the current database state. (The rule condition may be omitted, in which case it is always true.) This interpretation of rule conditions is used by several active database rule languages, e.g., *A-RDL*, *Chimera*, *Postgres*, and *Starburst*, as well as commercial trigger systems [WC96]. The action A is a data modification operation, identical to CA rules.

Rule processing is invoked after a user or application has performed modifications on the database. The basic algorithm is given in Figure 7. Rule processing is an iterative loop in which a triggered rule is selected (step (2) in Figure 7) and, if its condition is true (step (3)), its action is executed (step (4)). In step (2) more than one rule may be triggered; in this case, one rule is nondeterministically chosen for evaluation.[12]

[11] We do not consider complex or time-related events [WC96], which would significantly complicate the analysis.

[12] Analogous to CA rules, we do not consider a conflict resolution policy for selecting among multiple triggered rules; see Section 7.

```
repeat until no rule is triggered:
   select a triggered rule r;
   if r's condition is true
      execute r's action;
```

Figure 7: Rule processing algorithm for ECA rules.

When a rule is triggered, triggering occurs with respect to a database *transition*—the changes that have occurred since some previous database state. We consider a semantics in which each rule uses the transition since that rule was last selected, or since the state prior to the user modifications if the rule has not yet been selected during rule processing. Note the close correspondence between the notion of a transition here and the use of $C - C^{old}$ in CA rule processing (recall Section 4.1). Many ECA rule languages allow the condition and/or the action to refer to *delta relations* (or *transition tables*), which encapsulate the changes that occurred during a rule's transition. Typically, the delta relations available are the self-explanatory *inserted*(R), *deleted*(R), *old-updated*(R), and *new-updated*(R), with a *net effect* interpretation [Han92,SKdM92,WF90].

5.2 Quasi-CA Rules

Recall from Section 4.1 that a CA rule is activated when its condition becomes true as a result of modifications occurring during the rule's transition. We say that an ECA rule is activated when: (a) it is triggered by an event produced during the rule's transition, and (b) its condition is true on the current database state. In order to apply to ECA rules the analysis techniques for CA rules described in Sections 4.3 and 4.4, ECA rules must behave identically to CA rules with respect to rule activation. We identify a class of ECA rules, which we call *quasi-CA* rules, that, despite having ECA execution semantics, are activated identically to CA rules. We feel that many (if not most) ECA rules are quasi-CA in practice, justified by active database case studies in [CW90,BCP95a].

Definition 5.1: An ECA rule r is quasi-CA if the following conditions hold for all possible transitions and the associated database states. Let C^{old} and C be the results of evaluating the rule's condition in the states before and after the transition (where $C^{old} = \emptyset$ if the rule is evaluated for the first time during rule processing).

1. If $C - C^{old} \neq \emptyset$, then r is triggered by the transition. (That is, the rule is always triggered when a corresponding CA rule would be activated.)

2. If r is triggered by the transition and $C \neq \emptyset$, then $C - C^{old} \neq \emptyset$. (That is, if the triggered rule's condition C is true on the current state, then a corresponding CA rule's condition would be true on the transition.)

□

We now identify classes of ECA rules that are guaranteed to be quasi-CA. We separately consider rules that do or do not reference delta relations in their condition. First consider an ECA rule $r : \{T\} : C \rightarrow A$ that does not reference delta relations. For (1) to hold, the set of triggering events in r must include all modifications that may cause condition C to become true, i.e., that may cause $C - C^{old}$ to be non-empty. We call these events the *relevant events* $e(C)$ of condition C. Given a condition C, it is possible to derive the set $e(C)$ from C; a technique to do so is described, e.g., in [CW90]. To guarantee (2), C must be non-empty only when it produces new tuples with respect to its previous evaluation during rule processing. Recall from Section 5.1 that C is evaluated on the current database state, independent of the previous (pre-transition) state. Thus, it is not guaranteed that when r is triggered, $C - C^{old} \neq \emptyset$.[13] For (2) to hold, we must ensure that all data selected by C is modified or deleted by the transition. This would guarantee that if $C \neq \emptyset$, then C is satisfied by new data, i.e., $C - C^{old} \neq \emptyset$. The only rule we can be sure is executed during the transition is r. Hence, (2) holds if r's action A modifies or deletes all data selected by C.

To verify that A modifies or deletes all data selected by C, we apply the Propagation Algorithm to C and A: If the negative modifications E^- (recall Section 3.1) produced by the Propagation Algorithm contain C (again, using the standard notion of query containment [Ull89]), then we are guaranteed that all data in the result of C's query is deleted or modified by A. Recall from Section 3.7 that the Propagation Algorithm applied to a query Q and a modification M can produce a superset of the modifications on the result of Q due to the execution of M. However, the following theorem proves that, if $E^- \supseteq Q$, then all data in the result of query Q is actually deleted or modified by M. This says that the "conservativeness" of the algorithm is not a problem when verifying (2) above.

Theorem 5.1: Let M be a modification on a relation R, Q a query, and E^- the negative modifications obtained by applying the Propagation Algorithm to Q and M. For all database states d, if $M(d) \subseteq R(d)$[14] and $E^-(d) \supseteq Q(d)$, then $E^-(d) = Q(d)$.

Proof: The proof is given by induction on the structure of Q. *Base case:* $Q = R$. Let $M = E_{del}$. Then, $E^- = E_{del}$. If $E^-(d) \supseteq Q(d)$, then $E_{del}(d) \supseteq Q(d)$. By hypothesis, $E_{del}(d) \subseteq R(d)$. Thus $E_{del}(d) = R(d)$. The case of $M = E_{upd}$ is proved analogously. *Induction step:* Let the root of Q be a unary (resp. binary) relational operator *op* over a subtree S with incoming negative modification E^-_{in} (resp. over two subtrees S and S', of which S has incoming negative modifications E^-_{in}). By the induction hypothesis, we assume that $E^-_{in}(d) = S(d)$. We must show that if $E^-(d) \supseteq Q(d)$, then $E^-(d) = Q(d)$. As an example, consider $Q = \sigma_p S$. Then, $Q(d) = \sigma_p(S(d))$. Applying the propagation rule from the first line in Table 4, we obtain $E^- = \sigma_p E^-_{in}$. Hence, $E^-(d) = \sigma_p E^-_{in}(d)$. By the induction hypothesis, $E^-(d) = \sigma_p(S(d)) = Q(d)$. The other relational operators are verified

[13] For instance, if r is the only rule, and r's action does not operate on the data selected by C, then $C = C^{old}$ and $C - C^{old} = \emptyset$.

[14] More precisely, if $M = E_{del}$, then $E_{del}(d) \subseteq R(d)$, while if $M = E_{upd}$, then $\rho_{old}(E_{upd})(d) \subseteq R(d)$.

similarly. $\square$

The following theorem states formally when an ECA rule that does not reference delta relations is guaranteed to be quasi-CA.

Theorem 5.2: Consider an ECA rule $r : \{T\} : C \rightarrow A$ that does not reference delta relations. Let $e(C)$ be C's relevant events, and let the application of the Propagation Algorithm to C and A produce negative modifications E^-. If: (a) $e(C) \subseteq T$, and (b) $C \subseteq E^-$, then rule r is quasi-CA.

Proof: We must prove that when (a) and (b) hold, then both (1) and (2) in Definition 5.1 hold as well. Consider an arbitrary pair of database states and the transition between them. When $C - C^{old} \neq \emptyset$, by definition an event in $e(C)$ is generated. By (a) above, $e(C) \subseteq T$, hence r is always triggered. This guarantees (1). Consider (2). If r has not yet been executed, $C^{old} = \emptyset$. Hence, if r is triggered and $C \neq \emptyset$, then $C - C^{old} \neq \emptyset$. If r has been executed, let r's transition be represented as $d \xrightarrow{t_1} d' \xrightarrow{t_2} d''$, where d and d' are the database states immediately before and after r's previous execution, d'' is the state on which r is evaluated, t_1 is the transition due to the execution of r's action A, and t_2 contains all database changes leading from d' to d''. Then, by (b), all data selected by C was either deleted or updated by A in transition t_1. Let C' be the result of evaluating r's condition on d'. Either: (i) all data in C^{old} has been deleted and $C' = \emptyset$, or (ii) the changed data does not satisfy r's condition, so $C' = \emptyset$, or (iii) the changed data still satisfies r's condition, and $C' \neq \emptyset$, but also $C' - C^{old} \neq \emptyset$, since the data satisfying C^{old} has been modified. In case (i) and (ii), if r is triggered and $C \neq \emptyset$, then C is satisfied by tuples produced by t_2, and $C - C^{old} \neq \emptyset$.[15] In case (iii), by condition (1) in Definition 5.1 (proved above), r must be triggered by A and point (2) in Definition 5.1 is guaranteed. $\square$

We now discuss when ECA rules containing references to delta relations are quasi-CA. We say that a condition C is *incremental* if it references delta relations and the following conditions hold: (a) If C contains union operators, then all operands of unions are delta relations. (b) If C contains not-exists semijoin operators, then the first operand is a delta relation. These restrictions guarantee that, after C's evaluation, since its delta relations effectively become empty, $C = \emptyset$. Let the set of *relevant delta events* $e_\Delta(C)$ for condition C be all events corresponding to delta relations that appear "positively" in C, i.e., that do not occur within the second operand of a not-exists semijoin. If C is incremental, then $C \neq \emptyset$ only if some delta relation positive in C is non-empty. Thus, to guarantee point (1) in Definition 5.1, the set of triggering events in r must include $e_\Delta(C)$. If condition C is incremental, then after C's evaluation $C = \emptyset$; furthermore, only changes occurring after C is evaluated may cause $C \neq \emptyset$ again. Thus (2) holds.

The following theorem states formally when an ECA rule that references delta relations is guaranteed to be quasi-CA.

Theorem 5.3: Consider an ECA rule $r : \{T\} : C \rightarrow A$ that references delta relations in its

[15] Actually, it is possible for the net effect of t_1 and t_2 to make $C - C^{old} = \emptyset$, but in this case the rule would not have been triggered.

condition C. Let $e_\Delta(C)$ be C's relevant delta events. If: (a) C is incremental, and (b)$e_\Delta(C) \subseteq T$, then rule r is quasi-CA.

Proof: We must prove that when (a) and (b) hold, then both (1) and (2) in Definition 5.1 hold as well. After C is evaluated, all delta relations referenced by it effectively become empty. Since C is incremental, $C - C^{old} \neq \emptyset$ only if some tuple is produced in a delta relation positive in C. By (b) above, $e_\Delta(C) \subseteq T$, hence r is triggered. This guarantees (1) in Definition 5.1. Since C is incremental, point (2) is also guaranteed: since $C^{old} = \emptyset$ at the beginning of the transition, when $C \neq \emptyset$ we have $C - C^{old} \neq \emptyset$. □

5.3 Examples

We now use our ECA rule language to express the six rules described in Section 1.2 and we determine which rules are quasi-CA.

r_1: Rule `bad-account` is expressed in our ECA rule language as $\{T\} : C \rightarrow E_{upd}$ where

$$\begin{aligned}
T &= \textit{ins}\ \texttt{ACCOUNT},\ \textit{upd}\ \texttt{ACCOUNT.balance},\ \textit{upd}\ \texttt{ACCOUNT.rate} \\
C &= \pi_{\texttt{balance,rate}}(\sigma_{\texttt{balance<500} \wedge \texttt{rate>0}}\texttt{ACCOUNT}) \\
E_{upd} &= \mathcal{E}[\texttt{rate}' = 0]E_c \\
E_c &= \sigma_{\texttt{balance<500} \wedge \texttt{rate>0}}\texttt{ACCOUNT}
\end{aligned}$$

Rule r_1 is quasi-CA: (1) C does not reference delta relations, (2) $T = e(C)$, and (3) E_{upd} propagated on C produces an action E'_{del} that contains C. Note that this rule could have been written equivalently using a delta relation in place of relation `ACCOUNT`. However, we omitted the delta relation to provide an example of quasi-CA rules satisfying the conditions in Theorem 5.2.

r_2: Rule `raise-rate` is expressed in our ECA rule language as $\{T\} : C \rightarrow E_{upd}$ where

$$\begin{aligned}
T &= \textit{ins}\ \texttt{ACCOUNT},\ \textit{upd}\ \texttt{ACCOUNT.rate} \\
C &= \pi_{\texttt{rate}}(\sigma_{\texttt{rate>1} \wedge \texttt{rate<2}}\Delta(\texttt{ACCOUNT})) \\
E_{upd} &= \mathcal{E}[\texttt{rate}' = 2]E_c \\
E_c &= \sigma_{\texttt{rate>1} \wedge \texttt{rate<2}}\texttt{ACCOUNT}
\end{aligned}$$

and $\Delta(\texttt{ACCOUNT}) = \textit{inserted}\,(\texttt{ACCOUNT}) \cup \textit{new-updated}\,(\texttt{ACCOUNT})$. This rule is quasi-CA: C is incremental and $T = e_\Delta(C)$.

r_3: Rule `SF-bonus` is expressed in our ECA rule language as $\{T\} : C \rightarrow E_{upd}$ where

$$\begin{aligned}
T &= \textit{ins}\ \texttt{CUSTOMER},\ \textit{upd}\ \texttt{CUSTOMER.city} \\
C &= \pi_{\texttt{city,C}}(\sigma_{\texttt{C>1000}}(\mathcal{A}[\texttt{C} = \texttt{count(name)}](\sigma_{\texttt{city='SF'}}\texttt{CUSTOMER}))) \\
E_{upd} &= \mathcal{E}[\texttt{rate}' = \texttt{rate} + 1]E_c \\
E_c &= (\sigma_{\texttt{balance>5000} \wedge \texttt{rate<3}}\texttt{ACCOUNT}) \ltimes_{name} (\sigma_{\texttt{city='SF'}}\texttt{CUSTOMER})
\end{aligned}$$

This rule is not quasi-CA. Although $T = e(C)$, action E_{upd} only modifies data in relation ACCOUNT, which is not referenced in C. (Thus, the result of the Propagation Algorithm on C and E_{upd} is trivially empty.)

r_4: Rule start-bad is expressed in our ECA rule language as $\{T\} : C \rightarrow E_{ins}$ where

$$\begin{aligned}
T &= \mathit{ins}\ \texttt{ACCOUNT},\ \mathit{upd}\ \texttt{ACCOUNT.balance} \\
C &= \pi_{\texttt{num,balance}}((\sigma_{\texttt{balance<500}}\Delta(\texttt{ACCOUNT})) \ltimes_{\nexists num} (\sigma_{\texttt{end=null}}\texttt{LOW-ACC})) \\
E_{ins} &= \mathcal{E}[\texttt{start} = \texttt{today()}]\mathcal{E}[\texttt{end} = \texttt{null}]\pi_{\texttt{num,name}}((\sigma_{\texttt{balance<500}}\texttt{ACCOUNT}) \ltimes_{\nexists num} \\
&\quad (\sigma_{\texttt{end=null}}\texttt{LOW-ACC}))
\end{aligned}$$

and $\Delta(\texttt{ACCOUNT}) = \mathit{inserted}\ (\texttt{ACCOUNT}) \cup \mathit{new\text{-}updated}\ (\texttt{ACCOUNT})$, and today() is a system-defined function returning the current date. This rule is quasi-CA: C is incremental and $T = e_\Delta(C)$.

r_5: Rule end-bad is expressed in our ECA rule language as $\{T\} : C \rightarrow E_{upd}$ where

$$\begin{aligned}
T &= \mathit{upd}\ \texttt{ACCOUNT.balance} \\
C &= \pi_{\texttt{num}}((\sigma_{\texttt{end=null}}\texttt{LOW-ACC}) \ltimes_{num} (\sigma_{\texttt{balance}\geq\texttt{500}}\Delta(\texttt{ACCOUNT}))) \\
E_{upd} &= \mathcal{E}[\texttt{end}' = \texttt{today()}]((\sigma_{\texttt{end=null}}\texttt{LOW-ACC}) \ltimes_{num} (\sigma_{\texttt{balance}\geq\texttt{500}}\texttt{ACCOUNT}))
\end{aligned}$$

and $\Delta(\texttt{ACCOUNT}) = \mathit{new\text{-}updated}\ (\texttt{ACCOUNT})$. This rule is quasi-CA: C is incremental and $T = e_\Delta(C)$.

r_6: Rule decrease-bad is expressed in our ECA rule language as $\{T\} : C \rightarrow E_{upd}$ where

$$\begin{aligned}
T &= \mathit{upd}\ \texttt{ACCOUNT.balance},\ \mathit{upd}\ \texttt{ACCOUNT.rate} \\
C &= \pi_{\texttt{num}}((\sigma_{\texttt{rate>1} \wedge \texttt{balance}\geq\texttt{500} \wedge \texttt{balance}\leq\texttt{1000}}\Delta(\texttt{ACCOUNT})) \ltimes_{num} \\
&\quad (\sigma_{\texttt{D>50}}\mathcal{A}[\texttt{D} = \texttt{sum(end} - \texttt{start)}; \texttt{num}]\texttt{LOW-ACC})) \\
E_{upd} &= \mathcal{E}[\texttt{rate}' = 1]((\sigma_{\texttt{rate>1} \wedge \texttt{balance}\geq\texttt{500} \wedge \texttt{balance}\leq\texttt{1000}}\texttt{ACCOUNT}) \ltimes_{num} \\
&\quad (\sigma_{\texttt{D>50}}\mathcal{A}[\texttt{D} = \texttt{sum(end} - \texttt{start)}; \texttt{num}]\texttt{LOW-ACC}))
\end{aligned}$$

and $\Delta(\texttt{ACCOUNT}) = \mathit{new\text{-}updated}\ (\texttt{ACCOUNT})$. This rule is quasi-CA: C is incremental and $T = e_\Delta(C)$.

5.4 Termination Analysis

Identical to CA rules, a quasi-CA rule is activated by changes to the database occurring during the rule's transition. Hence, the Propagation Algorithm can be applied to verify activation of quasi-CA rules exactly as it was applied for CA rules (recall Section 4.3). For ECA rules that are not quasi-CA, the rule's condition may be true independent of the modification that caused the rule to become triggered. Hence, when verifying rule activation for these rules, it is necessary to conservatively assume that when a rule is triggered its condition is true. Thus, the activation of ECA rules in the general case can be verified only by considering triggering events and types of modifications.

This analysis technique, described in [AHW95,CW90], and which we call *event-action* analysis, is very conservative. We propose a "mixed" termination analysis technique that exploits the stronger method provided by the Propagation Algorithm for quasi-CA rules and degenerates to the weaker event-action technique for ECA rules that are not quasi-CA.

Assume we have determined which rules in the rule set are quasi-CA. The Activation Graph is built similarly to Section 4.3, by considering each rule pair in the rule set and analyzing if the first rule activates the second rule. Consider an arbitrary potential edge $r_i \rightarrow r_j$:

1. If r_j is quasi-CA, then the Propagation Algorithm can be used to detect activation, as justified in Section 5.2. If the Propagation Algorithm applied to r_i's action and r_j's condition returns insert or update actions, the edge is included in the activation graph.

2. If r_j is not quasi-CA, the edge is included in the activation graph if r_i performs a modification type included in r_j's set of triggering events.

Note that in case (1) we may need to apply the Propagation Algorithm to expressions containing delta relations. Incorporating the actual semantics of delta relations into the Propagation Algorithm appears to be very complex and not particularly effective. Thus, before applying the Propagation Algorithm, we substitute all references to delta relations by references to the corresponding relations. Because our analysis is performed at compile time and does not consider actual database states, this substitution, although conservative, allows us to perform a correct analysis of quasi-CA rules.

5.4.1 Examples

Consider the rule set in Section 5.3. The Triggering Graph generated using the technique in [AHW95, CW90] was shown in Figure 4. In Section 5.3 we have shown that rules r_1, r_2, r_4, r_5 and r_6 are quasi-CA, while rule r_3 is not. However, since rule r_3 has no incoming edges, the entire Triggering Graph can be analyzed using our stronger analysis technique. We thus obtain the Activation Graph that was shown in Figure 3 in which all cycles have been eliminated.

5.5 Confluence Analysis

Recall from Section 4.4 that rule commutativity is the core of our analysis technique for confluence of CA rules: If all rules in the rule set commute, then the rule set is confluent. This same condition holds for ECA rules [AHW95]. Similar to termination analysis, we propose a "mixed" analysis technique to analyze commutativity of ECA rules, based on the Propagation Algorithm for quasi-CA rules and on a weaker event-action analysis for ECA rules that are not quasi-CA. Assume we have determined which rules in the rule set are quasi-CA. Commutativity is verified for a rule pair ri, r_j by checking the four conditions in Lemma 4.1. To verify condition (1), we determine if r_i may activate r_j exactly as we have done for termination analysis. For condition (2), which requires that r_i cannot deactivate r_j, we consider the type of rule r_j: If r_j is quasi-CA, we can apply the Propagation Algorithm to r_j's condition and r_i's action; if the algorithm does not produce a delete operation, then condition (2) is satisfied. If r_j is not quasi-CA, then similar to Section 5.4 it is necessary to conservatively assume that when r_j is triggered its condition is true. Thus, a simple syntactic analysis is used to check if r_i deletes from any relation for which insert or update modifications are included in r_j's triggering events. For condition (3), we reverse the roles of r_i and r_j in the analysis of (1)–(2). For condition (4), we must determine if r_i's action may change the effect of r_j's action and vice-versa. Since action execution is identical in CA and ECA rules, the analysis here is identical as well.

5.5.1 Examples

Consider the pairwise commutativity analysis of the rule set described in Section 5.3. Although rule r_3 is not quasi-CA, the results of our mixed commutativity analysis are the same as for the corresponding CA rules; recall Figure 4.4.1(b). The result of applying the technique in [AHW95] was given in Figure 4.4.1(a). Thus, here again it is possible to detect the commutativity of several rule pairs (e.g., (r_1, r_3), (r_4, r_5)) for which commutativity cannot be detected using the technique in [AHW95].

6 Improving Commutativity Analysis

One important component of determining whether two rules are commutative is to test whether the rule's actions commute. Recall from Sections 4.4 and 5.5 that we can use the Propagation Algorithm to check commutativity of rule actions, which are database modifications, as follows. To determine if two modifications M_1 and M_2 commute, we must check if they may operate on the same data. We first rewrite M_1 as a query selecting data to be modified (recall Section 4.4.1) and we propagate data modification M_2 on it. Then we reverse the roles of M_1 and M_2. If the Propagation Algorithm returns an empty result in both cases, modifications M_1 and M_2 are guaranteed to commute. Our Propagation Algorithm approach is sufficient for guaranteeing commutativity of modifications. However, there are several special cases in which, although the modifications M_1 and M_2 may modify the same data, and thus applying the Propagation Algorithm produces a "may not commute" answer, M_1 and M_2 actually do commute. We consider cases when: (1) M_1 and M_2 are both delete operations, (2) M_1 is an update operation and M_2 is a delete operation, and (3) M_1 and M_2 are both update operations.

Intuitively, an important requisite for two modifications M_1 and M_2 performed on the same relation R to commute is that M_1 will not read tuples modified by M_2 to select the tuples on which it operates (although it may modify these tuples), and vice-versa. To guarantee commutativity of two delete operations, it is sufficient that neither modification evaluates aggregates on R, while more stringent conditions are required in the case of the other operation pairs. The conditions are formalized below.

We believe that all the remaining data modification pairs, which always involve an insert operation, do not commute if the modifications may operate on overlapping data. Informally, let I be an insert operation that reads data from some relation R that is modified by operation M (a delete, insert, or update). If the tuples read by I to perform the insertion may be modified by M, the outcome of I may be different depending on if it is performed before or after M. Hence I and M may not commute.

In the remainder of this section, we describe how commutativity analysis can be refined in the cases mentioned above and we provide illustrative examples. Proofs are given in Appendix A.

6.1 Pair of Delete Modifications

Consider two delete operations E_{del1} and E_{del2}, both defined on the same relation R. If E_{del1} and E_{del2} do not contain aggregates on attributes of R, then E_{del1} and E_{del2} commute, even though in some cases the outcome of applying the Propagation Algorithm is "may not commute." The following example provides intuition.

Example 6.1: Consider relation ACCOUNT from Section 1.2 and the delete operations:

$$E_{del1} = \sigma_{\texttt{balance}<0}\texttt{ACCOUNT}$$
$$E_{del2} = \sigma_{rate=0}\texttt{ACCOUNT}$$

The application of the Propagation Algorithm to the query obtained from E_{del1}, which is E_{del1}, and the delete operation E_{del2} yields:

$$E'_{del2} = \sigma_{\texttt{balance}<0}\sigma_{\texttt{rate}=0}\texttt{ACCOUNT}$$

which is satisfiable. An analogous result is produced by reversing the roles of E_{del1} and E_{del2}. This leads to the conclusion that the two delete operations may not commute. However, the operations do commute. Suppose E_{del1} is performed first. It deletes from ACCOUNT all tuples with $\texttt{balance} < 0$. Then E_{del2} is executed. E_{del2} selects for deletion all original tuples with $\texttt{rate} = 0$ (which it would have selected were it executed first) minus the tuples in $E_{del1} \cap E_{del2}$ (tuples with both $\texttt{balance} < 0$ and $\texttt{rate} = 0$), which have already been deleted by E_{del1}. Hence, after E_{del2} is executed, the tuples in $E_{del1} \cup E_{del2}$ (i.e., the tuples that either have $\texttt{balance} < 0$ or $\texttt{rate} = 0$) are deleted from ACCOUNT. The situation is analogous if we reverse the execution order of E_{del1} and E_{del2}. Thus, the two modifications commute.

The following theorem is proved in Appendix A.

Theorem 6.1: Let E_{del1} and E_{del2} be two delete operations on the same relation R. If neither operation contains aggregations over attributes of R then E_{del1} and E_{del2} commute. $\square$

6.2 Pair of Update and Delete Modifications

Consider a delete operation E_{del} and an update operation E_{upd} on the same relation R, both not including any aggregates on attributes of R. If the attributes updated by E_{upd} are not referenced in any predicate in E_{del}, then E_{del} and E_{upd} commute, even though in some cases the outcome of applying the Propagation Algorithm is "may not commute." The following example provides intuition.

Example 6.2: Consider relation ACCOUNT from Section 1.2 and the delete and update operations:

$$E_{del} = \sigma_{\texttt{balance}<0}\texttt{ACCOUNT}$$
$$E_{upd} = \mathcal{E}[\texttt{rate}' = \texttt{rate} + \texttt{1}]\sigma_{\texttt{num}<100}\texttt{ACCOUNT}$$

The application of the Propagation Algorithm to the query obtained from E_{upd} ($\pi_{\texttt{num,rate}}\sigma_{\texttt{num}<100}\texttt{ACCOUNT}$) and to modification E_{del} yields:

$$E'_{del} = \pi_{\texttt{num,rate}}\sigma_{\texttt{num}<100}\sigma_{\texttt{balance}<0}\texttt{ACCOUNT}$$

which is satisfiable. This leads to the conclusion that the two operations may not commute, while they actually do commute. Let us analyze the effect of applying the two modifications in either order. Let E_{del} be performed first and delete from ACCOUNT all tuples with $\texttt{balance} < 0$. E_{upd} then selects for the update operation the original tuples it would have selected were it executed first (with $\texttt{num} < 0$), minus the tuples with $\texttt{num} < 0$ and $\texttt{balance} < 0$, which have already been deleted by E_{del}. Thus, after both modifications have been performed, the tuples with $\texttt{balance} < 0$ are deleted from the database, while the tuples with $\texttt{num} < 0$ but $\texttt{balance} \geq 0$ have been updated. If E_{upd} is executed first, the tuples with $\texttt{num} < 0$ are updated. E_{del} then selects the tuples to be deleted without reading attribute `rate`, which has been updated by E_{upd}. Thus, it selects the same tuples (with $\texttt{balance} < 0$) it would have selected were it executed first, and it deletes them from relation ACCOUNT. Hence the two modifications commute.

The following theorem is proved in Appendix A.

Theorem 6.2: Let E_{del} be a delete operation and $E_{upd} = \mathcal{E}[A'_u = e]E_c$ an update operation expressed on the same relation R. E_{del} and E_{upd} commute if: (a) $A_u \notin A_p$, where A_p is the list of attributes referenced by E_{del} and A_u is the updated attribute[16], and (b) E_{del} and E_{upd} contain no aggregates on attributes of R. □

6.3 Pair of Update Modifications

Consider two update operations E_{upd1} and E_{upd2} on the same attribute A in the same relation R such that neither operation includes aggregates on attributes of R. If neither condition E_{c1} or E_{c2} reads attribute A and the two modifications to A can be applied in either order, then E_{upd1} and E_{upd2} commute, even though in some cases the outcome of applying the Propagation Algorithm is "may not commute." The following example provides intuition.

Example 6.3: Consider relation ACCOUNT from Section 1.2 and the update operations:

[16] If update E_{upd} modifies more than one attribute, this property must hold for all updated attributes.

$$E_{upd1} = \mathcal{E}[\mathtt{rate}' = \mathtt{rate} + 2]E_{c1}, \text{ where } E_{c1} = \sigma_{\mathtt{balance}>2000}\mathtt{ACCOUNT}$$
$$E_{upd2} = \mathcal{E}[\mathtt{rate}' = \mathtt{rate} - 1]E_{c2}, \text{ where } E_{c2} = \sigma_{\mathtt{num}<10}\mathtt{ACCOUNT}$$

The application of the Propagation Algorithm to the query obtained from E_{upd2} ($\pi_{\mathtt{num,rate}}E_{c2}$) and E_{upd1} yields:

$$E'_{upd} = \pi_{\mathtt{num,rate}}\sigma_{\mathtt{num}<10}E_{upd1}$$

which is satisfiable. (A similar result is obtained reversing the roles of E_{upd1} and E_{upd2}.) This leads to the conclusion that the two operations may not commute, while they actually do commute. Consider the effect of performing the two updates. Let E_{upd1} be performed first. It updates all tuples with `balance > 2000` (selected by its condition E_{c1}), increasing their rate by 2. Then E_{upd2} is performed. Its condition E_{c2} neither reads the updated attribute `rate` nor performs aggregates on attributes of `ACCOUNT`. Thus, it selects for update the same tuples with $\mathtt{num} < 10$ it would have selected were it executed first. E_{upd2} decrements the rate by 1 for all selected tuples. After the two updates have been performed, the tuples with $\mathtt{num} < 10$ and $\mathtt{balance} > 2000$ have been incremented by 1, the tuples with $\mathtt{num} \geq 10$ and $\mathtt{balance} > 2000$ have been incremented by 2, and the tuples with $\mathtt{num} < 10$ and $\mathtt{balance} \leq 2000$ have been decremented by 1. Reversing the roles of E_{upd1} and E_{upd2} yields the same final result, thus the two update operations commute.

The following theorem is proved in Appendix A.

Theorem 6.3: Let E_{upd1} and E_{upd2} be two update operations on relation R, where $E_{upd1} = \mathcal{E}[A' = f_1(A, B_1, \ldots, B_n)]E_{c1}$ and $E_{upd2} = \mathcal{E}[A' = f_2(A, B_1, \ldots, B_m)]E_{c2}$, and where E_{c1} and E_{c2} are relational expressions selecting the data to be updated in R. E_{upd1} and E_{upd2} commute if: (a) $f_1(f_2(A, B_1, \ldots, B_m), B_1, \ldots . B_n) = f_2(f_1(A, B_1, \ldots, B_n), B_1, \ldots, B_m)$, (b) the updated attribute $A \notin A_{c1}$ and $A \notin A_{c2}$, where A_{c1} and A_{c2} are lists of attributes referenced by E_{c1} and E_{c2} respectively, and (c) E_{c1} and E_{c2} do not contain aggregates on attributes of R. □

7 Conclusions

We have developed a general framework for static analysis of both Condition-Action (CA) and Event-Condition-Action (ECA) active database rules. We have provided a representation of active rules based on a generic extended relational algebra, which allows us to encode rules from most relational active database rule languages. We have presented our Propagation Algorithm for analyzing the interactions between queries and modifications, and we have discussed the complexity, soundness, and completeness of the algorithm. We have shown how the Propagation Algorithm can be applied to check termination and confluence for sets of CA and ECA rules, including a number of refinements to its most straightforward application. Our techniques improve considerably upon previous methods because our formal approach allows us to exploit the semantics of conditions and actions to analyze the interaction between rules. Note that the methods we present are also applicable to rule languages that "pass data" from the condition to the action (e.g., [GP91,Han92]), since our algorithm detects the actual modifications to rule conditions (inserts, deletes, and updates), not simply the transition between true and false. As in [AHW95], our analysis techniques identify the responsible rules when termination or confluence is not guaranteed. Hence, our techniques can be used as the kernel of an interactive development tool that helps rule designers develop sets of rules that are guaranteed to have the desired properties [BCP95a].

As future work, we plan to:

- Extend our rule model and analysis techniques to incorporate conflict resolution based on rule priorities [WC96]. Priorities restrict the possible execution sequences of rules, making the analysis more complex but sometimes more precise. Coupling our accurate analysis of rule interactions with the priority-based methods in [AHW95] should immediately produce a quite powerful analysis method for prioritized rules.

- Use the Propagation Algorithm to determine when it is guaranteed that a rule r_1 always activates another rule r_2 (i.e., when executing r_1's action always inserts into or updates the result of r_2's condition). This analysis is a straightforward modification of the algorithm for improving rule analysis presented in Section 4.5. The results of this analysis would be useful, e.g., for performing compile-time and run-time optimizations of rule processing.

The techniques presented in this paper have been used as the basis of an initial prototype rule analysis tool, implemented in the context of the *Chimera* project [WC96]. While the tool does not include all of the analysis techniques covered in this paper, it does use a number of our methods and will be extended in the future; see [BCP95a] for details.

Acknowledgements

We are grateful to members of the Stanford Database Group, especially Ashish Gupta, Dallan Quass, and Jeff Ullman, for lively and useful discussions about this work, and to Stefano Ceri for providing the technical impetus and enabling the collaboration.

References

[AHW95] A. Aiken, J.M. Hellerstein, and J. Widom. Static analysis techniques for predicting the behavior of active database rules. *ACM Transactions on Database Systems*, 20(1):3–41, March 1995.

[Bar94] E. Baralis. *An Algebraic Approach to the Analysis and Optimization of Active Database Rules.* PhD thesis, Politecnico di Torino, Torino, Italy, February 1994.

[BCP95a] E. Baralis, S. Ceri, and S. Paraboschi. ARACHNE: A tool for the analysis of active rules. In *Proceedings of the Second International Conference on Applications of Databases*, pages 68–81, Santa Clara, California, December 1995.

[BCP95b] E. Baralis, S. Ceri, and S. Paraboschi. Improved rule analysis by means of triggering and activation graphs. In *Proceedings of the Second International Workshop on Rules in Databases Systems*, pages 165–181, Athens, Greece, September 1995.

[BCP95c] E. Baralis, S. Ceri, and S. Paraboschi. Run-time detection of non-terminating active rule systems. In *Proceedings of the Fourth International Conference on Deductive and Object-Oriented Databases*, pages 38–54, Singapore, December 1995.

[BCP96] E. Baralis, S. Ceri, and S. Paraboschi. Modularization techniques for active rules design. *ACM Transactions on Database Systems*, March 1996. To appear.

[BCW93] E. Baralis, S. Ceri, and J. Widom. Better termination analysis for active databases. In *Proceedings of the First International Workshop on Rules in Database Systems*, pages 163–179, Edinburgh, Scotland, August 1993.

[BFKM85] L. Brownston, R. Farrell, E. Kant, and N. Martin. *Programming Expert Systems in OPS5: An Introduction to Rule-Based Programming.* Addison-Wesley, Reading, Massachusetts, 1985.

[BW94] E. Baralis and J. Widom. An algebraic approach to rule analysis in expert database systems. In *Proceedings of the Twentieth International Conference on Very Large Data Bases*, pages 475–486, Santiago, Chile, September 1994.

[BW95] E. Baralis and J. Widom. Using delta relations to optimize condition evaluation in active databases. In *Proceedings of the Second International Workshop on Rules in Databases Systems*, pages 292–308, Athens, Greece, September 1995.

[CCRL+90] S. Ceri, S. Crespi-Reghizzi, L. Lamperti, L. Lavazza, and R. Zicari. Algres: An advanced database for complex applications. *IEEE Software*, June 1990.

[CFPT94] S. Ceri, P. Fraternali, S. Paraboschi, and L. Tanca. Automatic generation of production rules for integrity maintenance. *ACM Transactions on Database Systems*, 19(3):367–422, September 1994.

[CG85] S. Ceri and G. Gottlob. Translating SQL into relational algebra: Optimization, semantics, and equivalence of SQL queries. *IEEE Transactions on Software Engineering*, 11(4):324–345, April 1985.

[CW90] S. Ceri and J. Widom. Deriving production rules for constraint maintenance. In *Proceedings of the Sixteenth International Conference on Very Large Data Bases*, pages 566–577, Brisbane, Australia, August 1990.

[CW91] S. Ceri and J. Widom. Deriving production rules for incremental view maintenance. In *Proceedings of the Seventeenth International Conference on Very Large Data Bases*, pages 577–589, Barcelona, Spain, September 1991.

[DE89] L.M.L. Delcambre and J.N. Etheredge. The Relational Production Language: A production language for relational databases. In L. Kerschberg, editor, *Expert Database Systems—Proceedings from the Second International Conference*, pages 333–351. Benjamin/Cummings, Redwood City, California, 1989.

[DHL90] U. Dayal, M. Hsu, and R. Ladin. Organizing long-running activities with triggers and transactions. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 204–214, Atlantic City, New Jersey, May 1990.

[Elk90] C. Elkan. Independence of logic database queries and updates. In *Proceedings of the Ninth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 154–160, March 1990.

[GP91] D.N. Gordin and A.J. Pasik. Set-oriented constructs: From Rete rule bases to database systems. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 60–67, Denver, Colorado, May 1991.

[Han92] E.N. Hanson. Rule condition testing and action execution in Ariel. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 49–58, San Diego, California, June 1992.

[HH91] J.M. Hellerstein and M. Hsu. Determinism in partially ordered production systems. IBM Research Report RJ 8009, IBM Almaden Research Center, San Jose, California, March 1991.

[ISO94] ISO-ANSI working draft: Database language SQL3 (X3H2/94/080 and SOU/003), 1994.

[Klu82] A. Klug. Equivalence of relational algebra and relational calculus query languages having aggregate functions. *Journal of the ACM*, 29(3):699–727, 1982.

[KU94] A.P. Karadimce and S.D. Urban. Conditional term rewriting as a formal basis for analysis of active database rules. In *Fourth International Workshop on Research Issues in Data Engineering (RIDE-ADS '94)*, pages 156–162, Houston, Texas, February 1994.

[LS93] A. Levy and Y. Sagiv. Queries independent of updates. In *Proceedings of the Ninetenth International Conference on Very Large Data Bases*, pages 171–181, Dublin, Ireland, August 1993.

[Ora92] ORACLE7 Reference Manual. Oracle Corporation, Redwood City, California, 1992.

[QW91] X. Qian and G. Wiederhold. Incremental recomputation of active relational expressions. *IEEE Transactions on Knowledge and Data Engineering*, 3(3):337–341, September 1991.

[RCBB89] A. Rosenthal, S. Chakravarthy, B. Blaustein, and J. Blakeley. Situation monitoring for active databases. In *Proceedings of the Fifteenth International Conference on Very Large Data Bases*, pages 455–464, Amsterdam, The Netherlands, August 1989.

[SKdM92] E. Simon, J. Kiernan, and C. de Maindreville. Implementing high level active rules on top of a relational DBMS. In *Proceedings of the Eighteenth International Conference on Very Large Data Bases*, pages 315–326, Vancouver, British Columbia, August 1992.

[TC94] H. Tsai and A.M.K. Cheng. Termination analysis of OPS5 expert systems. In *Proceedings of the AAAI National Conference on Artificial Intelligence*, pages 193–198, Seattle, Washington, September 1994.

[Ull89] J.D. Ullman. *Principles of Database and Knowledge-Base Systems, Volumes I and II.* Computer Science Press, Rockville, Maryland, 1989.

[vdVS93] L. van der Voort and A. Siebes. Termination and confluence of rule execution. In *Proceedings of the Second International Conference on Information and Knowledge Management*, Washington, DC, November 1993.

[WC96] J. Widom and S. Ceri. *Active Database Systems: Triggers and Rules for Advanced Database Processing*. Morgan Kaufmann, San Francisco, California, 1996.

[WF90] J. Widom and S.J. Finkelstein. Set-oriented production rules in relational database systems. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 259–270, Atlantic City, New Jersey, May 1990.

[WH95] T. Weik and A. Heuer. An algorithm for the analysis of termination of large trigger sets in an OODBMS. In *Proceedings of the International Workshop on Active and Real-Time Databases Systems*, Skövde, Sweden, June 1995.

[ZH90] Y. Zhou and M. Hsu. A theory for rule triggering systems. In *Advances in Database Technology—EDBT '90, Lecture Notes in Computer Science 416*, pages 407–421. Springer-Verlag, Berlin, March 1990.

A Commutativity Proofs

We give proofs of commutativity for the pairs of modifications described in Section 6. We use $Q(R_1, \ldots, R_n)$ to denote the result of evaluating query Q on relations $R_1, \ldots, R_n$. For a modification M, $M(R)$ denotes the result of evaluating the condition that selects the data on which modification M operates. For instance, $E_{del}(R)$ denotes the tuples in relation R that are selected by the delete operation characterized by relational expression E_{del}.

In the following Lemma, which is preliminary to the proof of the main theorem on commutativity for a pair of delete operations, we prove that relational difference is distributive with respect to a query Q when Q does not contain aggregates.

Lemma A.1: Let $R' \subseteq R$ be relations and $Q(R, S_1, \ldots, S_n)$ a query over relations $R, S_1, \ldots, S_n$, such that: (a) schema (Q) = schema (R), (b) $Q(R, S_1, \ldots, S_n) \subseteq R$, and (c) Q contains no aggregates on attributes of R. Then $Q(R, S_1, \ldots, S_n) - Q(R', S_1, \ldots, S_n) = Q(R - R', S_1, \ldots, S_n)$.

Proof: We use the abbreviation $Q(R)$ for $Q(R, S_1, \ldots, S_n)$, and analogously for $Q(R')$ and $Q(R'')$. The proof proceeds by induction on the cardinality of R'. *Base case:* Let card$(R') = 0$. Then $Q(R) - Q(R') = Q(R) - Q(\emptyset) = Q(R)$. Also $Q(R - R') = Q(R - \emptyset) = Q(R)$. *Induction step:* let $R'' = R' \cup \{t\}$ for some tuple $t \in R$ such that $R' \cap t = \emptyset$ and $R'' \subseteq R$. We want to prove that $Q(R) - Q(R'') = Q(R - R'')$.

$$
\begin{array}{ll}
Q(R) - Q(R'') = & \\
= Q(R) - (Q(R') \cup Q(\{t\})) & \text{no aggr. in } Q(R) \Rightarrow Q(R' \cup \{t\}) = Q(R') \cup Q(\{t\}) \\
= (Q(R) - Q(R')) - Q(\{t\}) & A - (BUC) = (A - B) - C \\
= Q(R - R') - Q(\{t\}) & \text{induction hypothesis} \\
= Q(R - (R'' - \{t\})) - Q(\{t\}) & R'' = R' \cup \{t\} \text{ and } R' \cap \{t\} = \emptyset \\
= Q((R - R'') \cup \{t\}) - Q(\{t\}) & A - (B - C) = (A - B) \cup C \\
= (Q(R - R'') \cup Q(\{t\})) - Q(\{t\}) & Q \text{ without aggr.} \Rightarrow \text{distrib. of } \cup \text{ with respect to } Q
\end{array}
$$

Now two cases are given: (a) $Q(\{t\}) = \emptyset$, which is trivial, and (b) $Q(\{t\}) = \{t\}$. In case (b), since $t \notin (R - R'')$ we have $Q(R - R'') \cap Q(\{t\}) = \emptyset$. Thus $Q(R) - Q(R'') = Q(R - R'')$. □

Theorem 6.1: Let E_{del1} and E_{del2} be two delete operations on the same relation R. If neither operation contains aggregations over attributes of R then E_{del1} and E_{del2} commute.

Proof: Let E_{del1} execute first. Its execution on R will produce a new relation $R' = R - E_{del1}(R)$. Then E_{del2} executes on R' and produces a new relation R''.

$$
\begin{array}{lll}
R'' & = R' - E_{del2}(R') & \text{defn. of } R'' \\
& = (R - E_{del1}(R)) - E_{del2}(R') & \text{defn. of } R' \\
& = R - (E_{del1}(R) \cup E_{del2}(R')) & (A - B) - C = A - (B \cup C) \\
& = R - (E_{del1}(R) \cup (E_{del2}(R - E_{del1}(R)))) & \text{defn. of } R' \\
& = R - (E_{del1}(R) \cup (E_{del2}(R) - E_{del2}(E_{del1}(R)))) & E_{del1}(R) \subseteq R \text{ and Lemma A.1} \\
& = R - (E_{del1}(R) \cup E_{del2}(R)) & E_{del2}(E_{del1}(R)) \subseteq E_{del1}(R)
\end{array}
$$

Now let E_{del2} execute first, producing $\overline{R}' = R - E_{del2}(R)$, and E_{del1} execute next, producing $\overline{R}''$.

$$
\begin{array}{lll}
\overline{R}'' & = \overline{R}' - E_{del1}(\overline{R}') & \text{defn. of } \overline{R}'' \\
& = (R - E_{del2}(R)) - E_{del1}(\overline{R}') & \text{defn. of } \overline{R}' \\
& = R - (E_{del2}(R) \cup E_{del1}(\overline{R}')) & (A - B) - C = A - (B \cup C) \\
& \ldots \text{ with steps analogous to above} \ldots & \\
& = R - (E_{del2}(R) \cup E_{del1}(R)) = R'' &
\end{array}
$$

□

The following two Lemmas prove properties of update and delete modifications that are needed in the proofs of the subsequent commutativity theorems.

Lemma A.2: Let E_{del} be a delete operation and $E_{upd} = \mathcal{E}[A'_u = e]E_c$ an update operation expressed on the same relation R such that E_{del} and E_{upd} contain no aggregates on attributes of R. Then $E_{upd}(E_{del}(R)) = \mathcal{E}[A'_u = e](E_c(R) \cap E_{del}(R))$.

Proof:

$$
\begin{array}{lll}
E_{upd}(E_{del}(R)) & = \mathcal{E}[A'_u = e]E_c(E_{del}(R)) & \text{defn. of } E_{upd} \\
 & = \mathcal{E}[A'_u = e](E_c(R) \cap E_{del}(R)) & \text{no aggr.} \Rightarrow Q_1(Q_2(R)) = Q_1(R) \cap Q_2(R)
\end{array}
$$

□

Lemma A.3: Let E_{del} be a delete operation and $E_{upd} = \mathcal{E}[A'_u = e]E_c$ an update operation expressed on the same relation R such that: (a) $A_u \notin A_p$, where A_p is the list of attributes referenced by E_{del} and A_u is the attribute updated by E_{upd}, and (b) E_{del} and E_{upd} contain no aggregates on attributes of R. Then $E_{del}(\rho_{old}(E_{upd}(R))) = E_{del}(\rho_{new}(E_{upd}(R)))$.

Proof: Follows directly from hypothesis. □

Theorem 6.2: Let E_{del} be a delete operation and $E_{upd} = \mathcal{E}[A'_u = e]E_c$ an update operation expressed on the same relation R. E_{del} and E_{upd} commute if: (a) $A_u \notin A_p$, where A_p is the list of attributes referenced by E_{del} and A_u is the updated attribute, and (b) E_{del} and E_{upd} contain no aggregates on attributes of R.

Proof: Let E_{del} execute first. Its execution on R produces a new relation $R' = R - E_{del}(R)$. Then E_{upd} executes on R' and produces a new relation $R'' = (R' - \rho_{old}(E_{upd}(R'))) \cup \rho_{new}(E_{udp}(R'))$, where

$$
\begin{array}{lll}
E_{upd}(R') & = E_{upd}(R - E_{del}(R)) & \text{defn. of } R' \\
 & = E_{upd}(R) - E_{upd}(E_{del}(R)) & \text{Lemma A.1} \\
 & = E_{upd}(R) - \mathcal{E}[A'_u = e](E_c(R) \cap E_{del}(R)) & \text{Lemma A.2} \\
 & = \mathcal{E}[A'_u = e]E_c(R) - \mathcal{E}[A'_u = e](E_c(R) \cap E_{del}(R)) & \text{defn. of } E_{upd} \\
 & = \mathcal{E}[A'_u = e](E_c(R) - (E_c(R) \cap E_{del}(R)) & \text{factoring of } \mathcal{E} \\
 & = \mathcal{E}[A'_u = e](E_c(R) - E_{del}(R)) & A - (A \cap B) = A - B
\end{array}
$$

Then

$$
\begin{array}{rcll}
R'' & = & (R' - \rho_{old}(E_{upd}(R'))) \cup \rho_{new}(E_{udp}(R')) & \text{defn. of } R'' \\
 & = & ((R - E_{del}(R)) - \rho_{old}(\mathcal{E}[A'_u = e](E_c(R) - E_{del}(R))) \cup & \text{defn. of } R'' \\
 & & \rho_{new}(\mathcal{E}[A'_u = e](E_c(R) - E_{del}(R))) &
\end{array}
$$

Now let E_{upd} execute first producing $\overline{R}' = (R - \rho_{old}(E_{upd}(R))) \cup \rho_{new}(E_{upd}(R))$ and E_{del} execute next producing $\overline{R}''$.

$$
\begin{array}{rcll}
\overline{R}'' & = & \overline{R}' - E_{del}(\overline{R}') & \text{defn. of } \overline{R}'' \\
 & = & ((R - \rho_{old}(E_{upd}(R))) \cup \rho_{new}(E_{upd}(R))) - & \text{defn. of } \overline{R}' \\
 & & E_{del}((R - \rho_{old}(E_{upd}(R))) \cup & \\
 & & \rho_{new}(E_{upd}(R))) & \\
 & = & ((R - \rho_{old}(E_{upd}(R))) \cup \rho_{new}(E_{upd}(R))) - & \text{distrib. of } \cup \\
 & & (E_{del}(R - \rho_{old}(E_{upd}(R))) \cup & \\
 & & E_{del}(\rho_{new}(E_{upd}(R))) & \\
 & = & ((R - \rho_{old}(E_{upd}(R))) \cup \rho_{new}(E_{upd}(R))) - & \text{Lemma A.1} \\
 & & ((E_{del}(R) - E_{del}(\rho_{old}(E_{upd}(R)))) \cup & \\
 & & E_{del}(\rho_{new}(E_{upd}(R)) & \\
 & = & ((R - \rho_{old}(E_{upd}(R))) \cup \rho_{new}(E_{upd}(R))) - & E_{del}(R) \supseteq E_{del}(\rho_{old}(E_{upd}(R)) \\
 & & E_{del}(R) & \text{and Lemma A.3} \\
 & = & (R \cup \rho_{new}(E_{upd}(R))) - (\rho_{old}(E_{upd}(R)) \cup E_{del}(R)) & \rho_{old}(E_{upd} \cap \rho_{new}(E_{upd}) = \emptyset \\
 & = & (R \cup \rho_{new}(E_{upd}(R))) - (E_{del}(R) \cup & \text{defn. of } E_{upd} \\
 & & \rho_{old}(\mathcal{E}[A'_u = e](E_c(R) - E_{del}(R)) & \\
 & = & ((R - E_{del}(R)) \cup \rho_{new}(\mathcal{E}[A'_u = e](E_c(R) - E_{del}(R))) - & (A \cup B) - C = (A - C) \cup (B - C) \\
 & & \rho_{old}(\mathcal{E}[A'_u = e](E_c(R) - E_{del}(R)) & \\
 & = & ((R - E_{del}(R))) - \rho_{old}(\mathcal{E}[A'_u = e](E_c(R) - E_{del}(R)) \cup & \rho_{old}(E_{upd}) \cap \rho_{new}(E_{upd}) = \emptyset \\
 & & \rho_{new}(\mathcal{E}[A'_u = e](E_c(R) - E_{del}(R)) = R'' &
\end{array}
$$

$\Box$

Theorem 6.3: Let E_{upd1} and E_{upd2} be two update operations on relation R, where $E_{upd1} = \mathcal{E}[A' = f_1(A, B_1, \ldots, B_n)]E_{c1}$ and $E_{upd2} = \mathcal{E}[A' = f_2(A, B_1, \ldots, B_m)]E_{c2}$, and where E_{c1} and E_{c2} are relational expressions selecting the data to be updated on R. E_{upd1} and E_{upd2} commute if: (a) $f_1(f_2(A, B_1, \ldots, B_m), B_1, \ldots . B_n) = f_2(f_1(A, B_1, \ldots, B_n), B_1, \ldots, B_m)$, (b) the updated attribute $A \notin A_{c1}$ and $A \notin A_{c2}$, where A_{c1} and A_{c2} are lists of attributes referenced by E_{c1} and E_{c2} respectively, and (c) E_{c1} and E_{c2} do not contain aggregates on attributes of R.

Proof: To prove commutativity, we must prove that $E_{upd1}(E_{upd2}(R)) = E_{upd2}(E_{upd1}(R))$. Let $f_1(A)$ be shorthand for $f_1(A, B_1, \ldots, B_n)$ and analogously for $f_2(A)$.

$$
\begin{array}{ll}
E_{upd1}(E_{upd2}(R)) = & \\
= \mathcal{E}[A'' = f_1(A')]E_{c1}(\mathcal{E}[A' = f_2(A)](E_{c2}(R))) & \text{defn. of } E_{upd1}, E_{upd2} \\
= \mathcal{E}[A'' = f_1(A')]\mathcal{E}[A' = f_2(A)]E_{c1}(E_{c2}(R))) & A' \notin E_{c1} \\
= \mathcal{E}[A'' = f_1(A')]\mathcal{E}[A' = f_2(A)](E_{c1}(R) \cap E_{c2}(R)) & \text{no aggr.} \Rightarrow Q_1(Q_2(R)) = Q_1(R) \cap Q_2(R) \\
= \mathcal{E}[A'' = f_1(f_2(A))](E_{c1}(R) \cap E_{c2(}R))) & \text{composition of } \mathcal{E}
\end{array}
$$

With analogous steps $E_{upd2}(E_{upd1}(R)) = \mathcal{E}[A'' = f_2(f_1(A))](E_{c1}(R) \cap E_{c2}(R))$. Then, if $f_1(f_2(A)) = f_2(f_1(A))$, $E_{upd1}(E_{upd2}(R)) = E_{upd2}(E_{upd1}(R))$, and E_{upd1} and E_{upd2} commute. $\Box$