Proc. of 1992 ACM-SIGMOD Conference, pages 59–68

Behavior of Database Production Rules: Termination, Confluence, and Observable Determinism

Alexander Aiken
Jennifer Widom
Joseph M. Hellerstein*

IBM Almaden Research Center
650 Harry Road
San Jose, CA 95120
{aiken, widom}@almaden.ibm.com, joey@postgres.berkeley.edu

Abstract. Static analysis methods are given for determining whether arbitrary sets of database production rules are (1) guaranteed to terminate; (2) guaranteed to produce a unique final database state; (3) guaranteed to produce a unique stream of observable actions. When the analysis determines that one of these properties is not guaranteed, it isolates the rules responsible for the problem and determines criteria that, if satisfied, guarantee the property. The analysis methods are presented in the context of the Starburst Rule System; they will form the basis of an interactive development environment for Starburst rule programmers.

1 Introduction

Production rules in database systems allow specification of data manipulation operations that are executed automatically whenever certain events occur or conditions are met, e.g. [GJ91,Han89,MD89,SJGP90,WF90]. Database production rules provide a general and powerful mechanism for integrity constraint enforcement, derived data maintenance, triggers and alerters, authorization checking, and versioning, as well as providing a platform for large and efficient knowledge-bases and expert systems. However, it can be very difficult in general to predict how a set of database production rules will behave. Rule processing occurs as a result of arbitrary database changes; certain rules are triggered initially, and their execution can trigger additional rules or trigger the same rules additional times. The unstructured, unpredictable, and often nondeterministic behavior of rule processing can be a nightmare for the database rule programmer.

A significant step in aiding the database rule programmer is to provide information about the following three properties of rule behavior:

- *Termination*: Is rule processing guaranteed to terminate after any set of changes to the database in any state?
- *Confluence*: Can the execution order of non-prioritized rules make any difference in the final database state? That is, if multiple rules are triggered at the same time during rule processing, can the final database state at termination of rule processing depend on which is considered first? If not, the rule set is *confluent*.
- *Observable Determinism*: If a rule action is visible to the environment (e.g., if it performs data retrieval or a rollback statement), then we say it is *observable*. Can the execution order of non-prioritized rules make any difference in the order or appearance of observable actions? If not, the rule set is *observably deterministic*.

These properties can be very difficult or impossible to decide in the general case. We have developed conservative static analysis algorithms that:

- guarantee that a set of rules will terminate or say that it may not terminate;
- guarantee that a set of rules is confluent or say that it may not be confluent;
- guarantee that a set of rules is observably deterministic or say that it may not be observably deterministic.

Furthermore, when the answer is "may not" for any of these properties, the analysis algorithms isolate the rules responsible for the problem and determine criteria that, if satisfied, guarantee the property. Hence the analysis can form the basis of an interactive environment where the rule programmer invokes the analyzer to obtain information about rule behavior. If termination, confluence, or observable determinism is desired but not guaranteed, then the user may verify that the necessary criteria are satisfied or may modify the rule set and try again.

Our analysis methods have been developed and are presented in the context of the *Starburst Rule System* [WCL91], a fully functional production rules facility integrated into the Starburst extensible relational DBMS prototype at the IBM Almaden Research Center [H^+90]. Although some aspects of the analysis are dependent on Starburst rules, we have tried to remain as general as possible and our methods certainly can be adapted to other database rule languages.

*Current address: CS Division, Department of EECS, University of California, Berkeley, CA 94720

1.1 Related Work

Most previous work in static analysis of production rules [HH91,Ras90,ZH90] differs from ours in two ways. First, it considers simplified versions of the *OPS5* production rule language [BFKM85]. OPS5 has a quite different model of rule processing than most database production rule systems, including the Starburst Rule System. Second, the goal of previous work is to impose restrictions and/or orderings on OPS5 rule sets such that unique fixed points are guaranteed. Our goal, on the other hand, is to permit arbitrary rule sets and provide useful information about their behavior in the database setting. In Section 9 we make some additional, more technical, comparisons, and explain how our analysis techniques subsume results in [HH91,Ras90,ZH90].

In [KU91], the issue of rule set termination is discussed, along with the issue of *conflicting updates*—determining when one rule may undo changes made by a previous rule. Although models and a problem-solving architecture for rule analysis are proposed, no algorithms are given. In [AS91], issues of termination and unique fixed points are considered in the context of various extensions to Datalog. In addition to the very different semantics of Datalog (logic) and production rules, [AS91] does not address the issue of determining whether a given rule set exhibits certain properties (as we do), but rather states results about whether all rule sets in a given language are guaranteed to exhibit the properties. In [CW90] we presented initial methods for analyzing termination in the context of deriving production rules for integrity constraint maintenance; these methods form the basis of our approach to termination in this paper.

1.2 Outline of Paper

As an introduction to database production rule languages and to establish a basis for our analysis techniques, in Section 2 we give the syntax and semantics of Starburst production rules. In Section 3 we introduce initial notation and definitions, and we describe some straightforward preliminary analysis of rule sets. In Section 4 we present a model of rule processing to be used as the formal basis for our analysis algorithms. Termination analysis is covered in Section 5 and confluence in Section 6. In Section 7 we give methods for analyzing *partial confluence*, which specifies that a rule set is confluent with respect to a portion of the database. Observable determinism is covered in Section 8. Finally, in Section 9 we draw conclusions and discuss future work.

2 The Starburst Rule System

We provide a brief overview of the set-oriented, SQL-based Starburst production rule language. Further details and numerous examples appear in [WCL91,WF90].

Starburst production rules are based on the notion of *transitions*. A transition is a database state change resulting from execution of a sequence of data manipulation operations. Rules consider only the *net effect* of transitions, meaning that: (1) if a tuple is updated several times, only the composite update is considered; (2) if a tuple is updated then deleted, only the deletion is considered; (3) if a tuple is inserted then updated, this is considered as inserting the updated tuple; (4) if a tuple is inserted then deleted, this is not considered at all. A formal theory of transitions and their net effects appears in [WF90].

The syntax for defining a rule is:

> **create rule** *name* **on** *table*
> **when** *transition predicate*
> [**if** *condition*]
> **then** *action*
> [**precedes** *rule-list*]
> [**follows** *rule-list*]

The *transition predicate* specifies one or more triggering operations on the rule's *table*: **inserted**, **deleted**, or **updated**$(c_1, \ldots, c_n)$, where $c_1, \ldots, c_n$ are column names. The rule is triggered by a given transition if at least one of the specified operations occurred in the net effect of the transition. The optional *condition* specifies an SQL predicate. The *action* specifies an arbitrary sequence of SQL data manipulation operations to be executed when the rule is triggered and its condition is true. The optional **precedes** and **follows** clauses are used to induce a partial ordering on the set of defined rules. If a rule r_1 specifies a rule r_2 in its **precedes** list, or if r_2 specifies r_1 in its **follows** list, then r_1 is higher than r_2 in the ordering. (We also say that r_1 has *precedence* or *priority* over r_2.) When no direct or transitive ordering is specified between two rules, their order is arbitrary.

A rule's condition and action may refer to the current state of the database through top-level or nested SQL **select** operations. In addition, rule conditions and actions may refer to *transition tables*, which are logical tables reflecting the changes to the rule's table that have occurred during the triggering transition. At the end of a given transition, transition table **inserted** in a rule refers to those tuples of the rule's table that were inserted by the transition, transition table **deleted** refers to those tuples that were deleted, and transition tables **new-updated** and **old-updated** refer to the new and old values (respectively) of the updated tuples. A rule may refer only to transition tables corresponding to its triggering operations.

Rules are activated at *rule assertion points*. There is an assertion point at the end of each transaction, and there may be additional user-specified assertion points within a transaction. We describe the semantics of rule processing at an arbitrary assertion point. The state change resulting from the user-generated database operations executed since the last assertion point (or start of the transaction) creates the first relevant transition, and some set of rules are triggered by this transition. A triggered rule r is chosen from this set for *consideration*. Rule r must be chosen so that no other triggered rule has precedence over r. If r has a condition, then it is checked. If r's condition is false, then another triggered rule is chosen for consideration. Otherwise, if r has no condition or its condition is true, then r's action is executed. After execution of r's action, all rules not yet considered are triggered only if their transition predicates hold with respect to the composite transition created by the initial transition and subsequent execution of r's action. That is, these rules see r's action as if it were executed as part of the initial transition. Rules already considered (including r) have already "processed" the initial transition; thus, they are triggered again only if their transition predicate holds with respect to the transition created by r's action. From the new set of triggered rules, a rule r' is chosen for consideration such that no other triggered rule has precedence over r'. Rule processing continues in this fashion.

At an arbitrary time in rule processing, a given rule is triggered if its transition predicate holds with respect to the (composite) transition since the last time it was considered. If it has not yet been considered, it is triggered if its transition predicate holds with respect to the transition since the last rule assertion point or start of the transaction. The values of transition tables in rule conditions and actions always reflect the rule's triggering transition. Rule processing terminates when there are no triggered rules.

The analysis techniques we present are based on this language and rule processing semantics, but with modifications they also could apply to other similar languages; see Section 9.

3 Definitions and Preliminary Analysis

Let $R = \{r_1, r_2, \ldots, r_n\}$ denote an arbitrary set of Starburst production rules to be analyzed. Analysis is performed on a fixed set of rules—when the rule set is changed, analysis must be repeated. (Incremental methods are certainly possible; see Section 9.) Let P denote the set of user-defined priority orderings on rules in R (as specified by their **precedes** and **follows** clauses), including those implied by transitivity. $P = \{r_i > r_j,\ r_k > r_l,\ \ldots\}$, where $r_i > r_j$ denotes that rule r_i has precedence over r_j. Let $T = \{t_1, t_2, \ldots, t_m\}$ denote the tables in the database schema, and let $C = \{t_i.c_j,\ t_k.c_l,\ \ldots\}$ denote the columns of tables in T. Finally, let O denote the set of database modification operations:

$$O = \{\langle \mathbf{I}, t\rangle \mid t \in T\} \ \cup\ \{\langle \mathbf{D}, t\rangle \mid t \in T\} \ \cup \\ \{\langle \mathbf{U}, t.c\rangle \mid t.c \in C\}$$

$\langle \mathbf{I}, t\rangle$ denotes insertions into table t, $\langle \mathbf{D}, t\rangle$ denotes deletions from table t, and $\langle \mathbf{U}, t.c\rangle$ denotes updates to column c of table t.

The following definitions are computed using straightforward preliminary analysis of the rules in R:

- *Triggered-By* takes a rule r and produces the set of operations in O that trigger r. *Triggered-By* is trivial to compute based on rule syntax.
- *Performs* takes a rule r and produces the set of operations in O that may be performed by r's action. *Performs* is trivial to compute based on rule syntax.
- *Triggers* takes a rule r and produces all rules r' that can become triggered as a result of r's action (possibly including r itself). $\mathit{Triggers}(r) = \{r' \in R \mid \mathit{Performs}(r) \cap \mathit{Triggered\text{-}By}(r') \neq \emptyset\}$.
- *Reads* takes a rule r and produces all columns in C that may be read by r in its condition or action. $\mathit{Reads}(r)$ contains every $t.c$ referenced in a **select** or **where** clause in $r's$ condition or action. In addition, for every $\langle \mathit{trans}\rangle.c$ referenced, where $\langle \mathit{trans}\rangle$ is one of **inserted**, **deleted**, **new-updated**, or **old-updated**, $t.c$ is in $\mathit{Reads}(r)$ for r's triggering table t. (Recall from Section 2 that **inserted**, **deleted**, **new-updated**, and **old-updated** are transition tables based on changes to t.)[1]
- *Can-Untrigger* takes a set of operations $O' \subseteq O$ and produces all rules that can be "untriggered" as a result of operations in O'. A rule is untriggered if it is triggered at some point during rule processing but not chosen for consideration, then subsequently no longer triggered because all triggering changes were undone by other rules.[2] $\mathit{Can\text{-}Untrigger}(O') = \{r \in R \mid \langle \mathbf{D}, t\rangle \in O'$ and $\langle \mathbf{I}, t\rangle$ or $\langle \mathbf{U}, t.c\rangle \in \mathit{Triggered\text{-}By}(r)$ for some $t \in T,\ t.c \in C\}$.
- *Choose* takes a set of triggered rules $R' \subseteq R$ and produces a subset of R' indicating those rules eligible for consideration (based on priorities). $\mathit{Choose}(R') = \{r_i \mid r_i \in R'$ and there is no $r_j \in R'$ such that $r_j > r_i \in P\}$.
- *Observable* takes a rule r and indicates whether r's action may be observable. In Starburst, a rule's action may be observable iff it includes a **select** or **rollback** statement.

4 Execution Model

We now define a formal model of execution-time rule processing. The model is based on *execution graphs* and accurately captures the semantics of rule processing described in Section 2. Note that execution graphs are used to discuss and to prove the correctness of our analysis techniques, but they are not part of the analysis itself.

A directed execution graph has a distinguished initial state representing the start of rule processing (at any rule assertion point) and zero or more final states representing termination of rule processing. The paths in the graph represent all possible execution sequences during rule processing; branches in the graph result from choosing different rules to consider when more than one is eligible. (Hence any graph for a totally ordered rule set has no branches.) The graph may have infinitely long paths, possibly due to cycles, and these represent nontermination of rule processing.

More formally, a *state* (node) S in an execution graph has two components: (1) a database state D; (2) a set TR containing each triggered rule and its associated transition tables. We denote this state as $S = (D, TR)$. The initial state I is created by an initial transition, which results from a sequence of user-generated database operations. Hence, $I = (D_I, TR_I)$ where D_I is a data-

[1] Note that, unlike in OPS5, there is no distinction between reading values "positively" and "negatively" in this rule language.

[2] As an example, a rule r_1 might be triggered by insertions, but another rule r_2 might delete all inserted tuples before r_1 is chosen for consideration. Untriggering is rare in practice.

base state and there is some (possibly empty) set of operations $O' \subseteq O$ such that:

$$TR_I = \{r \in R \mid O' \cap \mathit{Triggered\text{-}By}(r) \neq \emptyset\}$$

O' are the operations producing the initial transition, and TR_I contains the rules triggered by those operations. A final state F is some $(D_F, \emptyset)$, since no rules are triggered when rule processing terminates.

Each directed *edge* in an execution graph is labeled with a rule r and represents the consideration of r during rule processing. (This includes determining whether r's condition is true and, if so, executing r's action.) Using definitions from Section 3, the following lemma states certain properties that hold for all execution graphs. The lemma is stated without proof—it follows directly from the semantics of rule processing described in Section 2.

Lemma 4.1 (Properties of Execution Graphs) Consider any execution graph edge from a state (D_1, TR_1) to a state (D_2, TR_2) labeled with a rule r. Then:

- $r \in \mathit{Choose}(TR_1)$
- There is some (possibly empty) set of operations $O' \subseteq \mathit{Performs}(r)$ such that the triggered rules in TR_2 can be derived from the triggered rules in TR_1 by:
  1. removing rule r
  2. removing some subset of the rules in $\mathit{Can\text{-}Untrigger}(O')$
  3. adding all rules $r' \in R$ such that $O' \cap \mathit{Triggered\text{-}By}(r') \neq \emptyset$ □

The operations in O' are those executed by r's action. If r's condition is false then O' is empty. If r's condition is true then O' still may be a proper subset of $\mathit{Performs}(r)$ since, by the semantics of SQL, for most operations there are certain database states on which they have no effect. Finally, note that although rule r is removed in step 1, r may be added again in step 3 if $O' \cap \mathit{Triggered\text{-}By}(r) \neq \emptyset$.

The properties in Lemma 4.1 are guaranteed for all execution graphs. By performing more complex analysis on rule conditions and actions, by incorporating properties of database states, and by considering a variety of special cases, we probably can identify additional properties of execution graphs. Since our analysis techniques are based on execution graph properties, more accurate properties may result in more accurate rule analysis. We believe that the properties used here, although somewhat conservative, are sufficiently accurate to yield strong analysis techniques.

5 Termination

We want to determine whether the rules in R are guaranteed to terminate. That is, we want to determine if for all user-generated operations and initial database states, rule processing always reaches a point at which there are no triggered rules to consider. We take as an assumption that individual rule actions terminate. Hence, in terms of execution graphs, the rules in R are guaranteed to terminate iff all paths in every execution graph for R are finite.

As suggested in [CW90], termination is analyzed by constructing a directed *triggering graph* for the rules in R, denoted TG_R. The nodes in TG_R represent the rules in R and the edges represent the *Triggers* relationship. That is, there is an edge from r_i to r_j in TG_R iff $r_j \in \mathit{Triggers}(r_i)$.

Theorem 5.1 (Termination) If there are no cycles in TG_R then the rules in R are guaranteed to terminate.

Proof: Omitted due to space constraints; see [AWH92].

Hence, to determine whether the rules in R are guaranteed to terminate, triggering graph TG_R is constructed and checked for cycles. Although this may appear to be a very conservative approach, by considering only the known properties of our execution graph model (Lemma 4.1), we see that whenever there is a cycle in the triggering graph, our analysis cannot rule out the possibility that there is an execution graph with an infinite path. Clearly, however, there are a number of special cases in which there is a cycle in the triggering graph but other properties (not captured in Lemma 4.1) guarantee termination. Examples are:

- The action of some rule r on the cycle only deletes from a table t, and no other rules on the cycle insert into t. Eventually r's action has no effect.
- The action of some rule r on the cycle only performs a "monotonic" update (e.g. increments values), guaranteeing that the condition of some rule r' on the cycle eventually becomes false (e.g. some value is less than 10).

Although some such cases may be detected automatically, for now we assume that they are discovered by the user through the interactive analysis process: Once the analyzer has built the triggering graph for the rules in R, the user is notified of all cycles (or strong components). If the user is able to verify that, on each cycle, there is some rule r such that repeated consideration of the rules on the cycle guarantee that r's condition eventually becomes false or r's action eventually has no effect, then the rules in R are guaranteed to terminate.

As part of a case study, we used this approach to establish termination for a set of rules in a power network design application [CW90].

6 Confluence

Next we want to determine whether the rules in R are confluent. That is, we want to determine if the final database state at termination of rule processing can depend on which rule is chosen for consideration when multiple non-prioritized rules are triggered. In terms of execution graphs, the rules in R are confluent if every execution graph for R has at most one final state. (Recall that all final states in an execution graph have an empty set of triggered rules, so two different final states cannot represent the same database state.)

Confluence for production rules is a particularly difficult problem because, in addition to the standard problems associated with confluence [Hue80], we must take into account the interactions between rule triggering and rule priorities. For example, it is not sufficient to simply consider the combined effects of two rule actions; it also

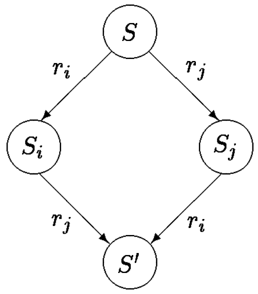


Figure 1: Commutative rules

is necessary to consider all rules that can become triggered, directly or indirectly, by those actions, and the relative ordering of these triggered rules. These issues are discussed as we develop our requirements for confluence in Section 6.3. As preliminaries, we first introduce the notion of *rule commutativity*, and we make a useful observation about execution graphs.

6.1 Rule Commutativity

We say that two rules r_i and r_j are *commutative* (or r_i and r_j *commute*) if, given any state S in any execution graph, considering rule r_i and then rule r_j from state S produces the same execution graph state S' as considering rule r_j and then rule r_i; this is depicted in Figure 1. If this equivalence does not always hold, then r_i and r_j are *noncommutative* (or r_i and r_j *do not commute*).

Each rule clearly commutes with itself. Based on the definitions of Section 3, we give a set of conditions for analyzing whether pairs of distinct rules commute.

Lemma 6.1 For distinct rules r_i and r_j, if any of the following conditions hold then r_i and r_j may be noncommutative; otherwise they are commutative:

1. $r_j \in$ *Triggers*(r_i), i.e. r_i can cause r_j to become triggered
2. $r_j \in$ *Can-Untrigger*(*Performs*(r_i)), i.e. r_i can untrigger r_j
3. $\langle \mathbf{I}, t\rangle$, $\langle \mathbf{D}, t\rangle$, or $\langle \mathbf{U}, t.c\rangle$ is in *Performs*(r_i) and $t.c$ is in *Reads*(r_j) for some $t.c \in C$, i.e. r_i's operations can affect what r_j reads
4. $\langle \mathbf{I}, t\rangle$ is in *Performs*(r_i) and $\langle \mathbf{D}, t\rangle$ or $\langle \mathbf{U}, t.c\rangle$ is in *Performs*(r_j) for some $t \in T$ or $t.c \in C$, i.e. r_i's insertions can affect what r_j updates or deletes[3]
5. $\langle \mathbf{U}, t.c\rangle$ is in both *Performs*(r_i) and *Performs*(r_j), i.e. r_i's updates can affect r_j's updates
6. any of 1–5 with r_i and r_j reversed □

We leave it to the reader to verify that if a pair of rules does not satisfy any of 1–6 then the rules are guaranteed to commute.

The conditions in Lemma 6.1 are somewhat conservative and probably could be refined by performing more complex analysis on rule conditions and actions and by considering a variety of special cases. As two examples of this, consider rules r_i and r_j such that:

1. r_i inserts into a table t and r_j deletes from t, but the tuples inserted by r_i never satisfy the delete condition of r_j, or
2. r_i and r_j update the same table but never the same tuples.

In the first example, r_i and r_j are noncommutative according to condition 4 of Lemma 6.1, but they do actually commute. In the second example, r_i and r_j are noncommutative according to condition 5 but do commute. Although some such cases may be detected automatically, for now we assume that they are specified by the user during the interactive analysis process: We allow the user to declare that pairs of rules that appear noncommutative according to Lemma 6.1 actually do commute. The analysis algorithms then treat these rules as commutative.

6.2 Observation

We say that two rules r_i and r_j are *unordered* if neither $r_i > r_j$ nor $r_j > r_i$ is in P. (Similarly, we say two rules r_i and r_j are *ordered* if $r_i > r_j$ or $r_j > r_i$ is in P.) Based on our execution graph model, we make the following observation about possible states, which is used in the next section to develop our criteria for confluence.

Observation 6.2 Consider any two unordered rules r_i and r_j in R. It is very likely that there is an execution graph with a state that has (at least) two outgoing edges, one labeled r_i and one labeled r_j. (Informally, there is very likely a scenario in which both r_i and r_j are triggered and eligible for consideration. Recall that a triggered rule r is eligible for consideration iff there is no other triggered rule with precedence over r.)

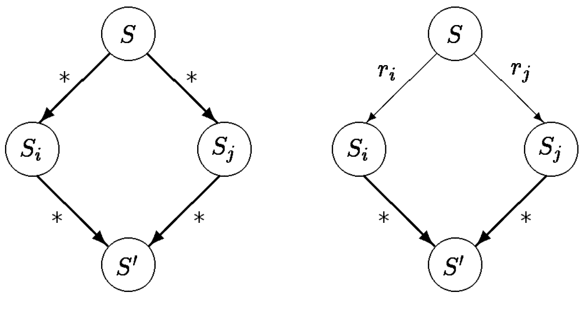
Justification: Let $O' =$ *Triggered-By*$(r_i) \cup$ *Triggered-By*(r_j). Consider an execution graph for which the operations in O' are the initial user-generated operations, so that r_i and r_j are both triggered in the initial state. Consider any path of length 0 or more from the initial state to a state $S = (D, TR)$ in which there are no rules $r \in TR$ such that $r > r_i$ or $r > r_j$ is in P, i.e. there are no triggered rules with precedence over r_i or r_j.[4] State S has at least two outgoing edges, one labeled r_i and one labeled r_j. □

6.3 Analyzing Confluence

We now return to the question of confluence. We want to determine if every execution graph for R is guaranteed to have at most one final state. For two execution graph states S_i and S_j, let $S_i \rightarrow S_j$ denote that there is an edge in the execution graph from state S_i to state S_j and let $S_i \xrightarrow{*} S_j$ denote that there is a path of length 0 or more from S_i to S_j. ($\xrightarrow{*}$ is the reflexive-transitive closure of $\rightarrow$.) Our first Lemma establishes conditions for confluence based on $\xrightarrow{*}$:

[3] In SQL it is possible to delete from or update a table without reading the table, which is why cases 4 and 5 are distinct from case 3.

[4] Such a path does not exist if r_i or r_j is untriggered along all potential paths, or if rules with precedence over r_i or r_j are considered indefinitely along all potential paths. These are highly unlikely (and probably undesirable) circumstances, but are why this is an observation rather than a theorem.



(a) Based on paths (b) Based on edges

Figure 2: Conditions for confluence

Lemma 6.3 (Path Confluence) Consider an arbitrary execution graph EG and suppose that for any three states S, S_i, and S_j in EG such that $S \xrightarrow{*} S_i$ and $S \xrightarrow{*} S_j$, there is a fourth state S' such that $S_i \xrightarrow{*} S'$ and $S_j \xrightarrow{*} S'$ (Figure 2a). Then EG has at most one final state.[5]

Proof: Suppose, for the sake of a contradiction, that EG has two distinct final states, F_1 and F_2. Let I be the initial state, so $I \xrightarrow{*} F_1$ and $I \xrightarrow{*} F_2$. Then, by assumption, there must be a fourth state S such that $F_1 \xrightarrow{*} S$ and $F_2 \xrightarrow{*} S$. Since F_1 and F_2 are both final states, $S = F_1$ and $S = F_2$, contradicting $F_1 \neq F_2$. □

It is quite difficult in general to determine when the supposition of Lemma 6.3 holds, since it is based entirely on arbitrarily long paths. The following Lemma gives a somewhat weaker condition that is easier to verify and implies the supposition of Lemma 6.3; it does, however, add the requirement that rule processing is guaranteed to terminate:

Lemma 6.4 (Edge Confluence) Consider an arbitrary execution graph EG with no infinite paths. Suppose that for any three states S, S_i, and S_j in EG such that $S \rightarrow S_i$ and $S \rightarrow S_j$, there is a fourth state S' such that $S_i \xrightarrow{*} S'$ and $S_j \xrightarrow{*} S'$ (Figure 2b). Then for any three states S, S_i, and S_j in EG such that $S \xrightarrow{*} S_i$ and $S \xrightarrow{*} S_j$, there is a fourth state S' such that $S_i \xrightarrow{*} S'$ and $S_j \xrightarrow{*} S'$.

Proof: Classic result; see e.g. [Hue80].

We use Lemma 6.4 as the basis for our analysis techniques. Based on this Lemma (along with Lemma 6.3), we can guarantee confluence for the rules in R if we know

1. there are no infinite paths in any execution graph for R (i.e., the rules in R are guaranteed to terminate), and
2. in any execution graph for R, for any three states S, S_i, and S_j such that $S \rightarrow S_i$ and $S \rightarrow S_j$, there is a fourth state S' such that $S_i \xrightarrow{*} S'$ and $S_j \xrightarrow{*} S'$.

We assume that the first condition has been established through the analysis techniques of Section 5; we focus our attention on analysis techniques for establishing the second condition.

[5] Sometimes the term confluence is used to denote the supposition of this Lemma [Hue80], which then implies confluence in the sense that we've defined it.

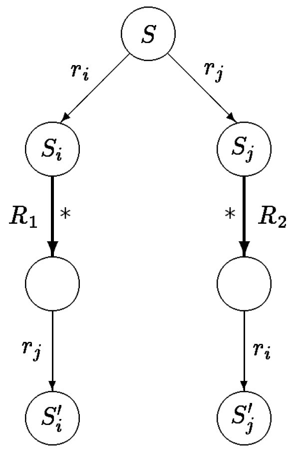


Figure 3: Paths towards common state S'

Consider any execution graph for R and any three states S, S_i, and S_j such that $S \rightarrow S_i$ and $S \rightarrow S_j$. This configuration is produced by every state S that has at least two unordered triggered rules that are eligible for consideration. Let r_i be the rule labeling edge $S \rightarrow S_i$ and r_j be the rule labeling edge $S \rightarrow S_j$, as in Figure 2b. We want to prove that there is a fourth state S' such that $S_i \xrightarrow{*} S'$ and $S_j \xrightarrow{*} S'$. It is tempting to assume that if r_i and r_j are commutative, then r_j can be considered from state S_i and r_i from S_j, producing a common state S' as in Figure 1. Unfortunately, this is not always possible: If r_i causes a rule r with precedence over r_j to become triggered, then r_j is not eligible for consideration in state S_i (similarly for r_i in state S_j). Since the new triggered rule r must be considered before rule r_j, r must commute with r_j. Furthermore, r may cause additional rules with precedence over r_j to become triggered.

With this in mind, we motivate the requirements for the existence of a common state S' that is reachable from both S_i and S_j. We do this by attempting to "build" valid paths from S_i and S_j towards S'; call these paths p_1 and p_2, respectively. From state S_i, triggered rules with precedence over r_j are considered until r_j is eligible; call these rules R_1. Similarly, from S_j triggered rules with precedence over r_i are considered until r_i is eligible; call these rules R_2. After this, r_j can be considered on path p_1 and r_i can be considered on path p_2. Paths p_1 and p_2 up to this point are depicted in Figure 3.

Now suppose that from state S'_i we can continue path p_1 by considering the rules in R_2 (in the same order), i.e. suppose the rules in R_2 are appropriately triggered and eligible. Similarly, suppose that from S'_j we can consider the rules in R_1. Then the same rules are considered along both paths. Consequently, if each rule in $\{r_i\} \cup R_1$ commutes with each rule in $\{r_j\} \cup R_2$, then the two paths are equivalent and reach a common state S'; this is depicted in Figure 4.

Unfortunately, even this scenario is not necessarily valid: There is no guarantee that the rules in R_2 are triggered and eligible from state S'_i; similarly for R_1 and S'_j.

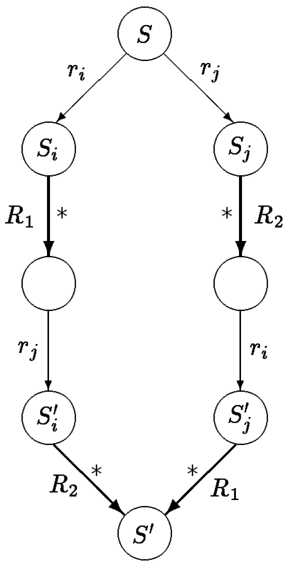


Figure 4: Paths reaching common state S'

(For example, a rule in R_2 may not be eligible from state S_i' because r_j triggered a rule with higher priority.) We can guarantee this, however, if we extend the rules originally considered in R_1 to include all eligible rules with precedence over rules in R_2, and extend the rules in R_2 similarly. Using this mutually recursive definition of R_1 and R_2, the pairwise commutativity of rules in $\{r_i\} \cup R_1$ with rules in $\{r_j\} \cup R_2$ guarantees the existence of state S', and consequently guarantees confluence.

To establish confluence for the rules in R, then, we must consider in this fashion every pair of rules r_i and r_j such that some state in some execution graph for R may have two outgoing edges, one labeled with r_i and one with r_j. Recall Observation 6.2: For any two unordered rules r_i and r_j, it is very likely that there is an execution graph with a state that has two outgoing edges, one labeled r_i and one labeled r_j. Consequently, we consider every pair of unordered rules, and our analysis requirement for confluence is stated as follows.

Definition 6.5 (Confluence Requirement) Consider any pair of unordered rules r_i and r_j in R. Let $R_1 \subseteq R$ and $R_2 \subseteq R$ be constructed by the following algorithm:

$R_1 \leftarrow \{r_i\}$
$R_2 \leftarrow \{r_j\}$
repeat until unchanged:

$$R_1 \leftarrow R_1 \cup \{r \in R \mid r \in \mathit{Triggers}(r_1) \text{ for some } r_1 \in R_1 \text{ and } r > r_2 \in P \text{ for some } r_2 \in R_2 \text{ and } r \neq r_j\}$$

$$R_2 \leftarrow R_2 \cup \{r \in R \mid r \in \mathit{Triggers}(r_2) \text{ for some } r_2 \in R_2 \text{ and } r > r_1 \in P \text{ for some } r_1 \in R_1 \text{ and } r \neq r_i\}$$

For every pair of rules $r_1 \in R_1$ and $r_2 \in R_2$, r_1 and r_2 must commute. □

The following lemma and theorem formally prove that the requirement of Definition 6.5 indeed guarantees confluence.

Lemma 6.6 (Confluence Lemma) Suppose the Confluence Requirement (Definition 6.5) holds for R. Then in any execution graph EG for R, for any three states S, S_i, and S_j in EG such that $S \rightarrow S_i$ and $S \rightarrow S_j$, there is a fourth state S' such that $S_i \xrightarrow{*} S'$ and $S_j \xrightarrow{*} S'$.

Proof: Omitted due to space constraints; see [AWH92]. (The formal proof parallels the motivation shown in Figure 4, although the full construction is slightly more complex.)

Theorem 6.7 (Confluence Theorem) Suppose the Confluence Requirement holds for R and there are no infinite paths in any execution graph for R. Then any execution graph for R has exactly one final state, i.e. the rules in R are confluent.

Proof: Let EG be any execution graph for R. By Confluence Lemma 6.6, for any three states S, S_i, and S_j in EG such that $S \rightarrow S_i$ and $S \rightarrow S_j$, there is a fourth state S' such that $S_i \xrightarrow{*} S'$ and $S_j \xrightarrow{*} S'$. Therefore, by Edge Confluence Lemma 6.4, for any three states S, S_i, and S_j in EG such that $S \xrightarrow{*} S_i$ and $S \xrightarrow{*} S_j$, there is a fourth state S' such that $S_i \xrightarrow{*} S'$ and $S_j \xrightarrow{*} S'$. By Path Confluence Lemma 6.3, EG has at most one final state, hence (since there are no infinite paths) EG has exactly one final state. □

Thus, analyzing whether the rules in R are confluent requires considering each pair of unordered rules r_i and r_j in R: Sets R_1 and R_2 are built from r_i and r_j according to Definition 6.5, and the rules in R_1 and R_2 are checked pairwise for commutativity.

6.4 Using Confluence Analysis

If our analysis determines that the rules in R are not confluent, it can be attributed to pairs of unordered rules r_i and r_j that generate sets R_1 and R_2 such that rules $r_1 \in R_1$ and $r_2 \in R_2$ do not commute. (In the most common case, r_1 and r_2 are r_i and r_j themselves; see Corollary 6.8 below.) With this information, it appears that the user has three possible courses of action towards confluence (short of modifying the rules themselves):

1. Certify that rules r_1 and r_2 actually do commute
2. Specify a user-defined priority between rules r_i and r_j so they no longer must satisfy the Confluence Requirement
3. Remove user-defined priorities so r_1 or r_2 is no longer part of R_1 or R_2

Approach 1 is clearly the best when it is valid. Approach 3 is non-intuitive and in fact useless: removing orderings to eliminate r_1 or r_2 from R_1 or R_2 simply produces a corresponding violation to the Confluence Requirement elsewhere. Hence, if Approach 1 is not applicable (i.e. rules r_1 and r_2 do not commute) then Approach 2 should be used. Note, however, that adding an ordering between rules r_i and r_j does not immediately guarantee confluence—sets R_1 or R_2 may increase for other pairs of rules and indicate that the rule set is still not confluent.[6]

[6]Intuitively, a source of non-confluence can appear to "move around", requiring an iterative process of adding or-

As guidelines for developing confluent rule sets, the following corollaries indicate simple properties that are satisfied by the rules in R if they are found to be confluent using our methods.

Corollary 6.8 If R is found to be confluent and r_i and r_j are unordered rules in R, then r_i and r_j commute.

Proof: Unordered rules r_i and r_j generate sets R_1 and R_2 such that $r_i \in R_1$ and $r_j \in R_2$. Hence, by the Confluence Requirement, r_i and r_j must commute. $\square$

Corollary 6.9 If R is found to be confluent and $P = \emptyset$ (i.e. there are no user-defined priorities between any rules in R), then every pair of rules in R commutes.

Proof: Follows directly from Corollary 6.8. $\square$

Corollary 6.10 If R is found to be confluent and r_i and r_j in R are such that r_i may trigger r_j (or vice-versa), then r_i and r_j are ordered.

Proof: Since $r_j \in \mathit{Triggers}(r_i)$, by our conditions for noncommutativity (Lemma 6.1), r_i and r_j do not commute. Suppose, for the sake of a contradiction, that r_i and r_j are unordered. Then by Corollary 6.8 they must commute. $\square$

Additional similar corollaries certainly exist and provide useful initial tools for the rule programmer.

We used our approach (by hand) to analyze confluence for several medium-sized rule applications. In most cases the rule sets were initially found to be non-confluent. However, for those rule sets that actually were confluent, user specification of rule commutativity eventually allowed confluence to be verified. Furthermore, for some rule sets the analysis uncovered previously undetected sources of non-confluence.

7 Partial Confluence

Confluence may be too strong a requirement for some applications. It sometimes is useful to allow rule set R to be non-confluent for certain "unimportant" (e.g. scratch) tables in the database, but to ensure that R is confluent for other "important" (e.g. data) tables. We call this *partial confluence*, or *confluence with respect to* T', where T' is a subset of the set of tables T in the database schema. In terms of execution graphs, the rules in R are confluent with respect to T' if, given any execution graph EG for R and any two final states $F_1 = (D_1, \emptyset)$ and $F_2 = (D_2, \emptyset)$ in EG, the tables in T' are identical in database states D_1 and D_2. (Partial confluence obviously is implied by confluence, since confluence guarantees at most one final state.)

Partial confluence is analyzed by analyzing confluence for a subset of the rules in R: those rules that can directly or indirectly affect the final value of tables in T'.

derings (or certifying commutativity) until the rule set is made confluent. This happens because our analysis techniques simply detect that confluence requires two rules to be ordered—the user chooses an ordering, and this choice affects which additional rules must be ordered.

Definition 7.1 (Significant Rules) Let $T' \subseteq T$ be a set of tables. The set of rules that are *significant with respect to* T', denoted $\mathit{Sig}(T')$, is computed by the following algorithm:

$$\mathit{Sig}(T') \leftarrow \{r \in R \mid \langle \mathbf{I}, t\rangle,\ \langle \mathbf{D}, t\rangle, \text{ or } \langle \mathbf{U}, t.c\rangle \text{ is in } \mathit{Performs}(r) \text{ for some } t \in T'\}$$

repeat until unchanged:

$$\mathit{Sig}(T') \leftarrow \mathit{Sig}(T') \cup \{r \in R \mid \text{there is an } r' \in \mathit{Sig}(T') \text{ such that } r' \text{ and } r \text{ do not commute}\} \quad \square$$

That is, $\mathit{Sig}(T')$ contains all rules that modify any table in T', along with (recursively) all rules that do not commute with rules in $\mathit{Sig}(T')$. This algorithm determines whether rules commute using our conservative conditions for noncommutativity from Lemma 6.1. Hence, the user can influence the computation of $\mathit{Sig}(T')$ by specifying that pairs of rules that appear noncommutative according to Lemma 6.1 actually do commute.

As in Confluence Theorem 6.7, partial confluence requires that rules are guaranteed to terminate. In this case, however, the rule set under consideration is $\mathit{Sig}(T')$. Thus, before analyzing partial confluence, termination of the rules in $\mathit{Sig}(T')$ must be established using the techniques of Section 5.[7]

Theorem 7.2 (Partial Confluence) Let $T' \subseteq T$ be a set of tables. Suppose the Confluence Requirement (Definition 6.5) holds for the rules in $\mathit{Sig}(T')$ and there are no infinite paths in any execution graph for $\mathit{Sig}(T')$. Then given any two final states F_1 and F_2 in any execution graph for R, the tables in T' are identical in F_1 and F_2, i.e. the rules in R are confluent with respect to T'.

Proof: Omitted due to space constraints; see [AWH92].

Hence, analyzing whether the rules in R are confluent with respect to T' requires first computing $\mathit{Sig}(T')$, then considering each pair of unordered rules r_i and r_j in $\mathit{Sig}(T')$: Sets R_1 and R_2 are built according to Definition 6.5 and checked pairwise for commutativity. If the analysis determines that the rules in R are not partially confluent, then the same interactive approach as that described in Section 6.4 for confluence can be used here to establish partial confluence.

8 Observable Determinism

In some database production rule languages, such as Starburst, the final database state may not be the only effect of rule processing—some rule actions may be visible to the environment (*observable*) while rules are being processed. When this is the case, the user may want to determine whether a rule set is *observably deterministic*, i.e. whether the order and appearance of observable rule actions is the same regardless of which rule is chosen for consideration when multiple non-prioritized rules are

[7] That is, even though the rules in $\mathit{Sig}(T')$ are never processed on their own, it must be established that if they were processed on their own they would terminate. As in Section 6.3, this is necessary for Definition 6.5 to guarantee confluence.

triggered. Note that observable determinism and confluence are orthogonal properties: a rule set may be confluent but not observably deterministic or vice-versa.

We analyze observable determinism using our techniques for partial confluence. Intuitively, we add a fictional table *Obs* to the database, and we pretend that those rules with observable actions also "timestamp and log" their observable actions in table *Obs*. We analyze the resulting rule set for confluence with respect to table *Obs*; if partial confluence holds, then the rule set is observably deterministic.

More formally, recall the definitions of Section 3. Let $T_{obs} = T \cup \{Obs\}$ be an extended set of tables, let $C_{obs} = C \cup \{Obs.c\}$ be an extended set of columns, and let O_{obs} be the corresponding extended set of operations. Let $Reads_{obs}$ and $Performs_{obs}$ extend the definitions of *Reads* and *Performs* as follows. For every $r \in R$ such that $Observable(r)$, add $Obs.c$ to $Reads(r)$ and $\langle \mathbf{I}, Obs\rangle$ to $Performs(r)$. For convenience, we say that a rule r *is observable* if $Observable(r)$.

Theorem 8.1 (Observable Determinism) Suppose, using extended definitions T_{obs}, C_{obs}, O_{obs}, $Reads_{obs}$, and $Performs_{obs}$, that our analysis methods for partial confluence determine that rule set R is confluent with respect to *Obs*. That is, suppose (from Theorem 7.2) that the Confluence Requirement of Definition 6.5 holds for the rules in $Sig(Obs)$ and there are no infinite paths in any execution graph for R. Then the rules in R are observably deterministic.

Proof: By supposition, any hypothetical behavior of the rules in R that is consistent with the definitions of $Reads_{obs}$ and $Performs_{obs}$ is confluent with respect to *Obs*. Consider the following such behavior. Suppose each observable rule r, in addition to its existing actions, inserts a new tuple into *Obs* that contains the current number of tuples in *Obs* (the "timestamp") and a complete description of r's observable actions (the "log"). Since there is a unique final value for *Obs*, the hypothetical tuples written to *Obs* must be identical on all execution paths. Consequently, there is only one possible order and appearance of observable actions, and the rules in R are observably deterministic. □

If, using the analysis methods indicated by this theorem, the rules in R are not found to be observably deterministic, then the same interactive approach as that described in Section 6.4 can be used to establish confluence with respect to *Obs*, and consequently observable determinism. Although this requires the user to be aware of fictional table *Obs*, the use of *Obs* in the analysis techniques is quite intuitive and may actually guide the user in establishing observable determinism.

The following corollary gives a simple property that is satisfied by the observable rules in R if they are found to be deterministic using our methods. Additional useful corollaries certainly exist.

Corollary 8.2 If R is found to be observably deterministic and r_i and r_j are distinct observable rules in R, then r_i and r_j are ordered.[8]

Proof: Since r_i is observable, $Obs.c \in Reads(r_i)$ and $\langle \mathbf{I}, Obs\rangle \in Performs(r_i)$; similarly for r_j. Therefore, by Definition 7.1, r_i and r_j are both in $Sig(Obs)$. In addition, by Lemma 6.1, r_i and r_j satisfy our conditions for noncommutativity. Suppose, for the sake of a contradiction, that r_i and r_j are unordered. Then r_i and r_j generate sets R_1 and R_2 (from Definition 6.5) such that $r_i \in R_1$ and $r_j \in R_2$. Hence, by the Confluence Requirement, r_i and r_j must commute. □

9 Conclusions and Future Work

We have given static analysis methods that determine whether arbitrary sets of database production rules are guaranteed to terminate, are confluent, are partially confluent with respect to a set of tables, or are observably deterministic. Our algorithms are conservative—they may not always detect when a rule set satisfies these properties. However, they isolate the responsible rules when a property is not satisfied, and they determine simple criteria that, if satisfied, guarantee the property. Furthermore, for the cases when these criteria are not satisfied, our methods often can suggest modifications to the rule set that are likely to make the property hold. Consequently, our methods can form the basis of a powerful interactive development environment for database rule programmers.

Although our methods have been designed for the Starburst Rule System, we expect that they can be adapted to accommodate the syntax and semantics of other database rule languages. In particular, the fundamental definitions of Section 3 (*Triggers*, *Performs*, *Choose*, etc.) can simply be redefined for an alternative rule language. Alternative rule processing semantics will probably require that the execution graph model is modified, which consequently will cause algorithms (and proofs) to be modified. However, our fundamental "building blocks" of rule analysis techniques can remain the same: the triggering graph for analyzing termination, the Edge and Path Lemmas for analyzing confluence, the notion of partial confluence, and the use of partial confluence in analyzing observable determinism.

Some technical comparisons can be drawn between this work and the results in [HH91, Ras90, ZH90]. In [HH91], a version of the OPS5 production rule language is considered, and a class of rule sets is identified that (conservatively) guarantees the *unique fixed point property*, which essentially corresponds to our notion of confluence. By defining a mapping between our language and the language in [HH91], we have shown that our confluence requirements properly subsume their fixed point requirements: if a rule set has the unique fixed point property according to [HH91], then our methods determine that the corresponding rule set is confluent, but not always vice-versa. The methods in [HH91] have previously been shown to subsume those in [Ras90, ZH90], hence our approach, although still conservative, appears quite accurate when compared with previous work.

[8] Note that this is not an if and only if condition: orderings between all pairs of observable rules does not necessarily guarantee observable determinism.

Finally, we plan a number of improvements and extensions to this work:

- **Incremental methods**: In our current approach, complete analysis is performed after any change to the rule set. In many cases it is clear that most results of previous analysis are still valid and only incremental additional analysis needs to be performed. We plan to modify our methods to incorporate incremental analysis. At the coarsest level, most rule applications can be partitioned into groups of rules such that, across partitions, rules reference different sets of tables and have no priority ordering. Although rules from different partitions are processed at the same time and their execution may be interleaved, they have no effect on each other. Hence, analysis can be applied separately to each partition, and it needs to be repeated for a partition only when rules in that partition change.
- **Less conservative methods**: As discussed throughout the paper, many of our assumptions, definitions, and algorithms are conservative, and there is room for refinement. This may include more complex analysis of SQL, more accurate properties of our execution model, and a suite of special cases.
- **Restricted user operations**: Our analysis assumes that the user-generated operations that initiate rule processing are arbitrary. However, in some cases it may be known that these will be of a particular type, i.e. users will only perform certain operations on certain tables. This may reduce possible execution paths during rule processing, and consequently may guarantee properties that otherwise do not hold. We plan to extend our methods so that termination, confluence, and observable determinism can be analyzed in the context of limited user-generated operations.
- **Implementation and experimentation**: We plan to implement our algorithms as part of an interactive development environment for the Starburst Rule System. Although we have verified by hand that our methods are indeed useful, implementation will allow practical experimentation with large and realistic rule applications.

Acknowledgements

Thanks to Stefano Ceri and Guy Lohman for helpful comments on an initial draft.

References

[AS91] S. Abiteboul and E. Simon. Fundamental properties of deterministic and nondeterministic extensions of datalog. *Theoretical Computer Science*, 78:137–158, 1991.

[AWH92] A. Aiken, J. Widom, and J.M. Hellerstein. Behavior of database production rules: Termination, confluence, and observable determinism. IBM Research Report RJ 8562, IBM Almaden Research Center, San Jose, California, January 1992.

[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.

[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.

[GJ91] N. Gehani and H.V. Jagadish. Ode as an active database: Constraints and triggers. In *Proceedings of the Seventeenth International Conference on Very Large Data Bases*, pages 327–336, Barcelona, Spain, September 1991.

[H+90] L.M. Haas et al. Starburst mid-flight: As the dust clears. *IEEE Transactions on Knowledge and Data Engineering*, 2(1):143–160, March 1990.

[Han89] E.N. Hanson. An initial report on the design of Ariel: A DBMS with an integrated production rule system. *SIGMOD Record, Special Issue on Rule Management and Processing in Expert Database Systems*, 18(3):12–19, September 1989.

[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.

[Hue80] G. Huet. Confluent reductions: Abstract properties and applications to term rewriting systems. *Journal of the ACM*, 27(4):797–821, October 1980.

[KU91] A.P. Karadimce and S.D. Urban. Diagnosing anomalous rule behavior in databases with integrity maintenance production rules. In *Third Workshop on Foundations of Models and Languages for Data and Objects*, Aigen, Austria, September 1991.

[MD89] D.R. McCarthy and U. Dayal. The architecture of an active database management system. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 215–224, Portland, Oregon, May 1989.

[Ras90] L. Raschid. Maintaining consistency in a stratified production system. In *Proceedings of the AAAI National Conference on Artificial Intelligence*, 1990.

[SJGP90] M. Stonebraker, A. Jhingran, J. Goh, and S. Potamianos. On rules, procedures, caching and views in data base systems. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 281–290, Atlantic City, New Jersey, May 1990.

[WCL91] J. Widom, R.J. Cochrane, and B.G. Lindsay. Implementing set-oriented production rules as an extension to Starburst. In *Proceedings of the Seventeenth International Conference on Very Large Data Bases*, pages 275–285, Barcelona, Spain, September 1991.

[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.

[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.