Static Analysis Techniques for
Predicting the Behavior of Database Production Rules

Alexander Aiken
Jennifer Widom
Joseph M. Hellerstein*

IBM Almaden Research Center
650 Harry Road
San Jose, CA 95120

{aiken,widom}@almaden.ibm.com, hellers@cs.wisc.edu

Abstract

Methods are given for statically analyzing sets of database production rules to determine if
the rules are (1) guaranteed to terminate, (2) guaranteed to produce a unique final database
state, and (3) guaranteed to produce a unique stream of observable actions. If 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.

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, making the database
system active [DW92]. Database production rules provide a general and powerful mechanism for
many database features, including integrity constraint enforcement, derived data maintenance, trigs-
gers, alerters, authorization checking, and versioning. In addition, active database systems provide
a convenient platform for large and efficient knowledge-bases and expert systems. A significant
drawback of active database systems, however, lies in the development of correct rule applications:
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 become a
nightmare for the database rule programmer.

A significant step in aiding the database rule programmer is to provide a facility that statically
analyzes sets of rules, providing 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. However, 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 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.

### 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 active database systems, including Starburst. 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 11 we make some additional, more technical, comparisons, and we 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 [CW90] we presented preliminary 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. An initial presentation of our analysis methods appears in [AWH92]; in this paper we provide additional intuition, refine and extend the methods in [AWH92], include numerous examples, and provide proofs for all lemmas and theorems.

1.2 Outline of Paper

As an introduction to active databases and to establish a basis for our analysis techniques, in Section 2 we give a syntax and semantics for the Starburst production rule language; Section 3 then motivates the termination, confluence, and observable determinism properties in the context of this language. In Section 4 we provide examples of Starburst production rules, illustrating rule sets that do and do not satisfy termination, confluence, and observable determinism. In Section 5 we introduce notation and definitions needed for rule analysis and we define some straightforward preliminary analysis techniques. In Section 6 we present a model of rule processing to be used as the formal basis for our analysis algorithms. Termination analysis is covered in Section 7 and confluence in Section 8. In Section 9 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 10. Finally, in Section 11 we draw conclusions and discuss future work.

2 The Starburst Rule System

We provide a brief overview of the Starburst database production rule language. Examples are given in Section 4; additional examples and further details 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:

\[
\begin{align*}
\text{create rule} & \quad \text{name on table} \\
\text{when} & \quad \text{triggering-operations} \\
[ & \quad \text{if condition}] \\
\text{then} & \quad \text{action} \\
[ & \quad \text{precedes rule-list}] \\
[ & \quad \text{follows rule-list}] 
\end{align*}
\]

The triggering-operations are one or more of inserted, deleted, and updated\((c_1, \ldots, c_n)\), where \(c_1, \ldots, c_n\) are columns of the rule’s table. 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.\footnote{In the current implementation rule conditions are SQL select statements, where the condition is true iff the select statement produces one or more tuples. In the context of rule conditions and for the purposes of rule analysis, predicates and select statements are equivalent [WCL91].} The action specifies an arbitrary sequence of database operations to be executed when the rule is triggered and its condition is true; these operations may be SQL data modification operations (insert, delete, update), SQL data retrieval operations (select), and transaction abort (rollback). 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.\footnote{The system actually orders non-prioritized rules using an algorithm based on rule creation time [ACL91], but this is an implementation feature separate from the semantics of the rule language.}

A rule's condition and action may refer to the current state of the database through top-level and 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 processing points. There is an automatic rule processing point at the end of each transaction, and there may be additional user-specified rule processing points within a transaction. We describe the semantics of rule processing at an arbitrary point. The state change resulting from the user-generated data modification operations executed since the last rule processing point (or start of the transaction) creates the first relevant transition, and some rules are triggered by this transition. As rule actions are executed, additional transitions are created which may trigger additional rules or trigger the same rules additional times. Rule processing follows an iterative algorithm in which:

1. A triggered rule $r$ is selected for consideration such that no other triggered rule has precedence over $r$.
2. $r$'s condition is evaluated.
3. If $r$'s condition is true then $r$'s action is executed.

For step 1 in this algorithm, a rule is triggered if one or more of its triggering operations occurred in the composite transition since the last time the rule was considered, or since the start of the transaction if the rule has not yet been considered. (The effect of this semantics is that each rule
sees each modification exactly once.) Rule processing terminates when a rollback operation is executed or when there are no more triggered rules.

User-invoked rule processing may specify that only a subset of the defined rules should be considered for execution, rather than all rules [Wid92]. Hence, the rule programmer may want to predict the behavior of a subset of the rules. The semantics of rule processing for rule subsets is identical to rule processing in the general case, so our analysis methods can be applied directly to arbitrary rule subsets.

The analysis techniques we present are based on the Starburst rule language and rule processing semantics, but with modifications the methods also could apply to other similar languages; see Section 11.

3 Termination, Confluence, and Observable Determinism

Suppose a set of Starburst rules has been defined using the language described in Section 2. Further suppose that arbitrary data modification operations have been performed, creating an initial triggering transition, and then rule processing is invoked. From the algorithm given in Section 2 it is clear that rule processing may not terminate—rules could trigger each other forever. A set of Starburst rules has the termination property if, for any initial triggering transition, rule processing is guaranteed to terminate, i.e. eventually rollback is executed or there are no more triggered rules.

In step 1 of the rule processing algorithm, a rule $r$ is selected such that no other triggered rule has precedence over $r$. Since the precedence of rules may be only a partial order (indeed, no ordering is required), many rules may be eligible for selection in this step. A set of Starburst rules has the confluence property if, for any initial triggering transition, there is some database state $D$ such that if rule processing terminates it terminates with the database in state $D$, regardless of which eligible rule is selected each time step 1 is performed.

Finally, when rule actions are executed (step 3 in the rule processing algorithm), these actions may be visible to the environment in which the transaction is executing. For example, actions may perform data retrieval, or they may abort the transaction. A set of Starburst rules has the observable determinism property if, for any initial triggering transition, the sequence of select and rollback operations executed during rule processing (including the values returned by select operations) is identical, regardless of which eligible rule is selected each time step 1 is performed.

4 Examples

For examples we consider a simple database schema with three tables:

- emp(id, rank, salary)
- bonus(emp-id, amount)
- sales(emp-id, month, number)

Table emp records each employee's rank and salary, table bonus records a bonus amount to be awarded to each employee, and table sales records each employee's number of sales on a monthly
basis. Because our example rules are quite simple, none includes an if clause; this does not detract from illustrating the salient points of termination, confluence, and observable determinism.

Example 4.1 (nonterminating) Nontermination is illustrated with two (admittedly contrived) rules. The first rule, bonus-rank, states that whenever an employee’s bonus is increased by more than 100, that employee’s rank is increased by 1:

```plaintext
create rule bonus-rank on bonus
when updated(amount)
then update emp
  set rank = rank + 1
  where id in (select emp-id from new-updated, old-updated
               where new-updated.emp-id = old-updated.emp-id
               and new-updated.amount - old-updated.amount > 100)
```

The second rule, rank-bonus, states that whenever an employee’s rank is modified, that employee’s bonus is increased by 10 times the new rank:

```plaintext
create rule rank-bonus on emp
when updated(rank)
then update bonus
  set amount = amount + 10 * (select rank from new-updated
                              where new-updated.id = bonus.emp-id)
  where emp-id in (select id from new-updated)
```

With these two rules, whenever an employee’s rank is modified to greater than 10, or whenever an employee with rank at least 10 receives a bonus greater than 100, rule processing does not terminate—the rules trigger each other forever. □

Example 4.2 (terminating, non-confluent) This is the most subtle and interesting of our examples. We specify three rules that are guaranteed to terminate. We suggest an ordering for the rules that appears to guarantee confluence, then we explain how this ordering is in fact insufficient for confluence. The first rule, good-sales, increases an employee’s salary by 10 whenever that employee posts sales greater than 50 for a month:

```plaintext
create rule good-sales on sales
when inserted
then update emp
  set salary = salary + 10
  where id in (select emp-id from inserted where number > 50)
```

The second rule, great-sales, increases an employee’s rank by 1 whenever that employee posts sales greater than 100 for a month:

```plaintext
create rule great-sales on sales
when inserted
then update emp
  set rank = rank + 1
  where id in (select emp-id from inserted where number > 100)
```
Although rules \textit{good-sales} and \textit{great-sales} may be triggered at the same time, their actions cannot affect each other, so no relative ordering between the rules is needed for confluence. The third rule, \textit{rank_raise}, increases an employee's salary by 10\% whenever that employee's rank reaches 15 (we assume that ranks do not decrease):

\begin{verbatim}
create rule rank-raise on emp
  when updated(rank)
  then update emp
    set salary = 1.1 * salary
    where id in (select id from new-updated where rank = 15)
\end{verbatim}

For confluence it is clear that a relative ordering is required between rule \textit{rank-raise} and rule \textit{good-sales} since, if both rules are triggered at the same time, the order in which their actions execute influences the final salary. We specify that \textit{rank-raise} has priority over \textit{good-sales}, so the 10\% increase given by \textit{rank-raise} does not reflect the increase of 10 given by \textit{good-sales}. It also is clear that a relative ordering is required between rule \textit{rank-raise} and rule \textit{great-sales} since, if both rules are triggered at the same time, executing \textit{great-sales} first could increase an employee's rank to 16 before \textit{rank-raise} awards the 10\% increase. Hence, we add the following clause to rule \textit{rank-raise}:

\begin{verbatim}
  precedes good-sales, great-sales
\end{verbatim}

It turns out that these orderings still are insufficient for confluence. Suppose rules \textit{good-sales} and \textit{great-sales} are triggered at the same time, and there is an employee whose newly posted sales exceeds 100, whose salary is 60, and whose rank is 14. Suppose rule \textit{good-sales} is executed first. Then \textit{good-sales} increases the employee's salary to 70, \textit{great-sales} increases the employee's rank to 15, and finally \textit{rank-raise} is triggered and increases the employee's salary to 77. Now suppose instead that rule \textit{great-sales} is executed first. Then \textit{great-sales} increases the employee's rank to 15, \textit{rank-raise} is triggered and increases the employee's salary to 66, and finally \textit{good-sales} increases the employee's salary to 76. Hence there are two possible final values for the employee's salary, and the rules are not confluent.

The important property to observe here is that there are two rules (\textit{good-sales} and \textit{great-sales}) that appear to be unrelated and therefore appear to need no relative ordering. However, the existence of a third rule (\textit{rank-raise}) that is related to both of the first two rules means that the first two rules must be ordered to achieve confluence. □

Example 4.3 (terminating, confluent, not observably deterministic) Consider rules \textit{good-sales} and \textit{rank-raise} from Example 4.2, ordered so that \textit{rank-raise} precedes \textit{good-sales}. Now add a third rule, \textit{new-rank}, that displays an employee's ID, rank, and salary, along with a "new-rank" message, whenever that employee's rank is modified:

\begin{verbatim}
create rule new-rank on emp
  when updated(rank)
  then select id,rank,salary,"new-rank" from new-updated
\end{verbatim}
These three rules are guaranteed to terminate and they are confluent—the final database state is guaranteed to be unique. However, the rules are not observably deterministic. Suppose all three rules are triggered at the same time. Then the salaries displayed by rule new-rank may take on three different values, depending on whether new-rank is executed before rank-raise, after rank-raise but before good-sales, or after good-sales. □

Example 4.4 (terminating, confluent, observably deterministic) Consider rule bonus-rank from Example 4.1, rule good-sales from Example 4.2, and rule new-rank from Example 4.3, with new-rank specified to follow bonus-rank and good-sales. This set of rules exhibits all three properties: termination, confluence, and observable determinism. □

5 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 11.) Let \( P \) denote the set of user-defined priority orderings on rules in \( R \) (as specified by their \texttt{ precedes} and \texttt{ 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:

\[
O = \{(I,t) \mid t \in T\} \cup \{(D,t) \mid t \in T\} \cup \{(U,t.c) \mid t.c \in C\}.
\]

\((I,t)\) denotes insertions into table \( t \), \((D,t)\) denotes deletions from table \( t \), and \((U,t.c)\) denotes updates to column \( c \) of table \( t \).

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

- \texttt{Triggered-By} takes a rule \( r \) and produces the set of operations in \( O \) that trigger \( r \). \texttt{Triggered-By} is trivial to compute based on rule syntax.

- \texttt{Performs} takes a rule \( r \) and produces the set of operations in \( O \) that may be performed by \( r \)'s action. \texttt{Performs} is trivial to compute based on rule syntax.

- \texttt{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). \texttt{Triggers}(\( r \)) = \{\( r' \in R \mid \texttt{Performs}(r) \cap \texttt{Triggered-By}(r') \neq \emptyset \}\).

- \texttt{Uses} takes a rule \( r \) and produces all columns in \( C \) that may be referenced when evaluating \( r \)'s condition or executing a data modification operation in \( r \)'s action. \texttt{Uses}(\( r \)) contains every \( t.c \) referenced in \( r \)'s condition, every \( t.c \) referenced in the \texttt{where} clause of a delete or update operation in \( r \)'s action, and every \( t.c \) referenced in a nested select expression in an insert, delete, or update operation in \( r \)'s action. In addition, for every \( \langle \text{trans} \rangle.c \) referenced in
this same way, where \(<trans>\) is one of inserted, deleted, new-updated, or old-updated, 
t.c is in \(\text{Uses}(r)\) 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 \(t\).)

- **Can-Untrigger** takes a set of operations \(O^t \subseteq O\) and produces all rules that can be "un-
  triggered" as a result of operations in \(O^t\). 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.\(^3\) \(\text{Can-Untrigger}(O^t) = \{r \in R \mid \langle D, t \rangle \in O^t \text{ and } \langle I, t \rangle \text{ or } \langle U, t, c \rangle \in \text{Triggered-By}(r) \text{ for some } t \in T, t.c \in C\}\).

- **Choose** takes a set of triggered rules \(R^t \subseteq R\) and produces a subset of \(R^t\) indicating those 
  rules eligible for consideration (based on priorities). \(\text{Choose}(R^t) = \{r_i \mid r_i \in R^t \text{ and there is no } r_j \in R^t \text{ such that } r_j > r_i \in P\}\).

- **Rollback** takes a rule \(r\) and indicates whether executing \(r\)'s action is guaranteed to abort the 
  transaction. In Starburst, \(\text{Rollback}(r)\) is true iff one of the operations comprising \(r\)'s action 
  is rollback.

- **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 **rollback** or a top-level **select** statement.

6 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 
these graphs are never actually constructed and they are not part of the analysis itself.

A directed execution graph has a distinguished initial state representing the start of rule pro-
cessing (at any processing point) and zero or more final states representing termination of rule 
processing. The paths in the graph represent all possible execution sequences during rule process-
ing; 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 
ininitely long paths (possibly but not necessarily due to cycles in the graph); these paths represent 
nontermination of rule processing.

More formally, a **state** (node) \(S\) in an execution graph has two components: (1) a database 
state \(D_1\); (2) a set \(TR\) containing each triggered rule and its associated transition tables. We denote 
this state as \(S = (D_1, TR)\). 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 database 
state and there is some set of operations \(O^t \subseteq O\) such that:

\(^3\)As an example, a rule \(\tau_1\) might be triggered by insertions, but another rule \(\tau_2\) might delete all inserted tuples 
before \(\tau_1\) is chosen for consideration. Untriggering is rare in practice.
\[ TR_I = \{ r \in R \mid O' \cap \text{Triggered-By}(r) \neq \emptyset \} \]

\( O' \) is the set of operations producing the initial transition, and \( TR_I \) contains the rules triggered by those operations. A final state \( F \) is some \((DF, \emptyset)\), since no rules are triggered when rule processing terminates. Note that a final state \( F \) may correspond to normal termination or it may correspond to termination due to a rollback statement; in the latter case \( F = (DT, \emptyset) \), where \( DT \) is the database state at the start of the transaction.

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 5, 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 6.1 (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:

- \( r \in \text{Choose}(TR_1) \)
- Either:
  
  1. Rollback\((r)\), \( D_2 = DT \), and \( TR_2 = \emptyset \); or
  2. There is some (possibly empty) set of operations \( O' \subseteq \text{Performs}(r) \) such that the triggered rules in \( TR_2 \) can be derived from the triggered rules in \( TR_1 \) by:
     
     a. removing rule \( r \)
     b. removing some subset of the rules in \( \text{Can-Untrigger}(O') \)
     c. adding all rules \( r' \in R \) such that \( O' \cap \text{Triggered-By}(r') \neq \emptyset \)

Case (1) corresponds to the situation in which \( r \)'s condition is true and its action includes rollback. Case (2) corresponds to the situation in which \( r \)'s condition is false, or \( r \)'s condition is true and its action does not include rollback. In case (2), the operations in \( O' \) are those executed by \( r \)'s action, where \( O' \) is empty if \( r \)'s condition is false. If \( r \)'s condition is true then \( O' \) still may be a proper subset of \( \text{Performs}(r) \) 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 (a), \( r \) may be added again in step (c) if \( O' \cap \text{Triggered-By}(r) \neq \emptyset \).

The properties in Lemma 6.1 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.
7 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 if 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 $TGR$. The nodes in $TGR$ represent all rules $r \in R$ such that $\text{Rollback}(r)$ is false; the edges in $TGR$ represent the $\text{Triggers}$ relationship. That is, there is an edge from $r_i$ to $r_j$ in $TGR$ iff $r_j \in \text{Triggers}(r_i)$. We exclude a rule $r_i$ from the graph if $\text{Rollback}(r_i)$, since if $r_i$ contains rollback then it will not trigger any other rule $r_j$, even if technically $r_j \in \text{Triggers}(r_i)$.

**Theorem 7.1 (Termination)** If there are no cycles in $TGR$ then the rules in $R$ are guaranteed to terminate.

**Proof:** We must prove that all paths in every execution graph for $R$ are finite. Suppose, for the sake of a contradiction, that there are no cycles in graph $TGR$ but there is an infinite path $p$ in some execution graph for $R$. Then, since there are only finitely many rules, some rule $r$ must appear on infinitely many edges on path $p$. By the properties of execution graphs in Lemma 6.1, $r$ must be added to set $TR$ of triggered rules (by step 3) infinitely many times. Hence there must be some operation $o \in \text{Performs}(r_1)$ for only a finite number of rules $r_1$, there must be some rule $r_1$ such that $o \in \text{Performs}(r_1)$ and $r_1$ appears on infinitely many edges on path $p$. Note that, by definition, $r \in \text{Triggers}(r_1)$, so there is an edge from $r_1$ to $r$ in $TGR$. Since $r_1$ appears on infinitely many edges, $r_1$ must be added to set $TR$ infinitely many times. By the same reasoning as above, there is some rule $r_2$ such that $r_1 \in \text{Triggers}(r_2)$ (so there is an edge from $r_2$ to $r_1$ in $TGR$) and $r_2$ appears on infinitely many edges on path $p$. This reasoning continues, generating rules $r_3$, $r_4$, etc. Since, by assumption, there are no cycles in graph $TGR$, this reasoning generates infinitely many rules, a contradiction. $\square$

Hence, to determine whether the rules in $R$ are guaranteed to terminate, triggering graph $TGR$ is constructed and checked for cycles. As an example, consider the triggering graph $TGR$ for rules $R = \{\text{bonus-rank, rank-bonus}\}$ of Example 4.1. There are two nodes in $TGR$, one representing rule $\text{bonus-rank}$ (call the node $br$) and one representing rule $\text{rank-bonus}$ (call the node $rb$). Since $\text{rank-bonus} \in \text{Triggers(\text{bonus-rank})}$, there is an edge in $TGR$ from node $br$ to node $rb$. Similarly, since $\text{bonus-rank} \in \text{Triggers(\text{rank-bonus})}$, there is an edge in $TGR$ from node $rb$ to node $br$. Therefore, there is a cycle in $TGR$, and our analysis determines that this set of rules may not terminate. In Examples 4.2, 4.3, and 4.4, the triggering graphs are acyclic, so our analysis determines that the rules are guaranteed to terminate.

---

4If $\text{Rollback}(r_i)$ and $r_j \in \text{Triggers}(r_i)$, then $r_i$ must include both a data modification operation and a rollback operation in its action. This is allowed syntactically, but semantically it makes little sense—the effect of the data modification operation always will be undone when the rollback operation is executed.
Although our approach to termination may appear to be very conservative, by considering only the known properties of our execution graph model (Lemma 6.1), 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 6.1) 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].

8 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 is necessary to consider all rules that can become triggered, directly or indirectly, by those actions, and the relative ordering of these triggered rules (recall Example 4.2). These issues are discussed further as we develop our requirements for confluence in Section 8.3. As preliminaries, we first introduce the notion of rule commutativity, and we make a useful observation about execution graphs.
8.1 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 5, we give a set of conditions for analyzing whether pairs of distinct rules commute.

**Lemma 8.1** 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. $\langle I, t \rangle$, $\langle D, t \rangle$, or $\langle U, t.c \rangle$ is in $Performs(r_i)$ and $t.c$ is in $Uses(r_j)$ for some $t.c \in C$, i.e. $r_i$'s operations can affect what $r_j$ uses
4. $\langle I, t \rangle$ is in $Performs(r_i)$ and $\langle D, t \rangle$ or $\langle U, t.c \rangle$ 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\(^5\)
5. $\langle U, t.c \rangle$ 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 \(\square\)

It is straightforward to verify that if a pair of rules does not satisfy any of 1–6 then the rules are guaranteed to commute.

As illustration, consider the examples from Section 4:

- **Example 4.1** – Rules **bonus-rank** and **rank-bonus** are noncommutative according to condition 1 of Lemma 8.1, because **rank-bonus** $\in Triggers(\text{bonus-rank})$.

\(^5\)In SQL it is possible to delete from or update a table without referencing columns of the table, which is why cases 4 and 5 are distinct from case 3.
• Example 4.2 - Rules good-sales and great-sales are commutative. Rules good-sales and rank-raise are noncommutative according to condition 5, because \(<U, \text{emp.salary}\>) is in both \(\text{Performs(good-sales)}\) and \(\text{Performs(rank-raise)}\). Rules great-sales and rank-raise are noncommutative according to condition 3, because \(<U, \text{emp.rank}\>) is in \(\text{Performs(great-sales)}\) and \(\text{emp.rank}\) is in \(\text{Uses(rank-raise)}\). (Recall from Section 5 that \(\text{emp.rank}\) is in \(\text{Uses(rank-raise)}\) because new-updated.rank is referenced and \(\text{emp}\) is the triggering table.) Rules great-sales and rank-raise are noncommutative also according to condition 1, because rank-raise \(\in \text{Triggers(great-sales)}\).

• Example 4.3 - Rules good-sales and rank-raise are noncommutative as in Example 4.2. Rules good-sales and new-rank are commutative; rules rank-raise and new-rank also are commutative. (Rule new-rank commutes with rules good-sales and rank-raise because, although new-rank does access column emp.salary, this column is not in \(\text{Uses(new-rank)}\)—it is not used in the rule's condition or in a data modification action.)

• Example 4.4 - Rules bonus-rank and good-sales are commutative. Rules bonus-rank and new-rank are noncommutative according to condition 1, because new-rank \(\in \text{Triggers(bonus-rank)}\). Rules good-sales and new-rank are commutative.

The conditions in Lemma 8.1 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 \(r_j\)

2. \(r_i\) and \(r_j\) update the same table and column but never the same tuples

In the first example, \(r_i\) and \(r_j\) are noncommutative according to condition 4 of Lemma 8.1, 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 8.1 actually do commute. The analysis algorithms then treat these rules as commutative.

8.2 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 8.2 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 $O' = Triggered-Bg(r_i) \cup Triggered-Bg(r_j)$. 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 $r_j$. State $S$ has at least two outgoing edges, one labeled $r_i$ and one labeled $r_j$. □

### 8.3 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 $S_i \Rightarrow S_j$ denote that there is a path of length 0 or more from $S_i$ to $S_j$. ($\Rightarrow$ is the reflexive-transitive closure of $\rightarrow$.)

Our first Lemma establishes conditions for confluence based on $\Rightarrow$:

**Lemma 8.3 (Path Confluence)** Consider an arbitrary execution graph $EG$ and suppose that for any three states $S_i, S_j,$ and $S_k$ in $EG$ such that $S_i \Rightarrow S_j$ and $S_j \Rightarrow S_k$, there is a fourth state $S'$ such that $S_i \Rightarrow S'$ and $S_j \Rightarrow S'$ (Figure 2a). Then $EG$ has at most one final state.\(^6\)

**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 $I \Rightarrow F_1$ and $I \Rightarrow F_2$. Then, by assumption, there must be a fourth state $S$ such that $F_1 \Rightarrow S$ and $F_2 \Rightarrow S$. 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 8.3 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 8.3; it does, however, add the requirement that rule processing is guaranteed to terminate:

**Lemma 8.4 (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 $S_i \Rightarrow S'$ and $S_j \Rightarrow S'$ (Figure 2b). Then 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 $S_i \Rightarrow S'$ and $S_j \Rightarrow S'$.

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

\(^6\)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.

\(^7\)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.
We use Lemma 8.4 as the basis for our analysis techniques. Based on this Lemma (along with Lemma 8.3), 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 \( S_i \stackrel{\ast}{\rightarrow} S' \) and \( S_j \stackrel{\ast}{\rightarrow} S' \).

We assume that the first condition has been established through the analysis techniques of Section 7; we focus our attention on analysis techniques for establishing the second condition.

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 \( S_i \stackrel{\ast}{\rightarrow} S' \) and \( S_j \stackrel{\ast}{\rightarrow} S' \). 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

![Diagram](image-url)
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 \( S'_i \); 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 \( S'_j \). (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 \( S'_i \), 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 8.2: 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 8.5 (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:

\[
\begin{align*}
R_1 & \leftarrow \{r_i\} \\
R_2 & \leftarrow \{r_j\} \\
\text{repeat until unchanged:} \\
R_1 & \leftarrow R_1 \cup \{r \in R \mid r \in \text{Triggers}(r_i) \text{ for some } r_1 \in R_1 \\
& \quad \text{and } r > r_2 \in P \text{ for some } r_2 \in R_2 \text{ and } r \neq r_j \} \\
R_2 & \leftarrow R_2 \cup \{r \in R \mid r \in \text{Triggers}(r_2) \text{ for some } r_2 \in R_2 \\
& \quad \text{and } r > r_1 \in P \text{ for some } r_1 \in R_1 \text{ and } r \neq r_i \} 
\end{align*}
\]

For every pair of rules \( r_1 \in R_1 \) and \( r_2 \in R_2 \), \( r_1 \) and \( r_2 \) must commute. \( \square \)
The following lemma and theorem formally prove that the requirement of Definition 8.5 indeed guarantees confluence.

**Lemma 8.6 (Confluence Lemma)** Suppose the Confluence Requirement (Definition 8.5) 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 \( S_i \rightarrow S' \) and \( S_j \rightarrow S' \).

**Proof:** See Appendix A.1. (The formal proof parallels the motivation shown in Figure 4, although the full construction is slightly more complex.)

**Theorem 8.7 (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 8.6, 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 \( S_i \rightarrow S' \) and \( S_j \rightarrow S' \). Therefore, by Edge Confluence Lemma 8.4, 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 \( S_i \rightarrow S' \) and \( S_j \rightarrow S' \). By Path Confluence Lemma 8.3, \( EG \) has at most one final state, hence (since there are no infinite paths) \( EG \) has exactly one final state. \( \square \)

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 8.5, and the rules in \( R_1 \) and \( R_2 \) are checked pairwise for commutativity.

### 8.4 Examples

Consider Example 4.2 in which there are three rules: **good-sales**, **great-sales**, and **rank-raise**. The only pair of unordered rules is **good-sales** and **great-sales**. Letting \( r_i = \text{good-sales} \) and
\( r_j = \text{great-sales} \), we construct sets \( R_1 \) and \( R_2 \) of Definition 8.5. \( R_1 = \{ \text{good-sales} \} \), since there are no rules in \( \text{Triggers(\text{good-sales})} \). \( R_2 = \{ \text{great-sales, rank-raise} \} \), since \( \text{rank-raise} \in \text{Triggers(\text{great-sales})} \) and \( \text{rank-raise} > \text{good-sales} \). Now, since \( \text{good-sales} \in R_1 \) and \( \text{rank-raise} \in R_2 \) do not commute (recall Section 8.1), then according to Definition 8.5 we cannot conclude that this set of rules is confluent. To see this in terms of execution graphs, consider a state \( S \) in which \( \text{good-sales} \) and \( \text{great-sales} \) are both triggered. If \( \text{good-sales} \) is considered first, then path \( p_1 \) from \( S \) to a final state considers rules \( \text{good-sales, great-sales, and rank-raise} \), in that order. If \( \text{great-sales} \) is considered first, then path \( p_2 \) from \( S \) to a final state considers rules \( \text{great-sales, rank-raise, and good-sales} \), in that order. Although rules \( \text{good-sales} \) and \( \text{great-sales} \) do commute, rules \( \text{good-sales} \) and \( \text{rank-raise} \) do not. Hence paths \( p_1 \) and \( p_2 \) may lead to different final states.

Now consider Example 4.3 in which there are three rules: \( \text{good-sales, rank-raise, and new-rank} \). There are two pairs of unordered rules: \( \text{good-sales/new-rank} \) and \( \text{rank-raise/new-rank} \). For both pairs, sets \( R_1 \) and \( R_2 \) of Definition 8.5 contain only the rules themselves, and the rules commute. Hence by Theorem 8.7 the rules are confluent.

Finally consider Example 4.4 in which there are three rules: \( \text{bonus-rank, good-sales, and new-rank} \). The only pair of unordered rules is \( \text{bonus-rank} \) and \( \text{good-sales} \). Letting \( r_i = \text{bonus-rank} \) and \( r_j = \text{good-sales} \), we construct sets \( R_1 \) and \( R_2 \) of Definition 8.5; consider \( R_2 \) first. \( R_2 = \{ \text{good-sales} \} \), since there are no rules in \( \text{Triggers(\text{good-sales})} \). \( R_1 = \{ \text{bonus-rank} \} \), since although \( \text{new-rank} \in \text{Triggers(\text{bonus-rank})} \), there is no rule \( r_2 \in R_2 \) such that \( \text{new-rank} > r_2 \). Since \( \text{good-sales} \) and \( \text{bonus-rank} \) commute, the rules are confluent.

### 8.5 Using Confluence Analysis

If our analysis determines that the rules in a set \( R \) 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 8.8 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.\(^6\)

As initial guidelines for developing confluent rule sets, the following Corollaries indicate simple properties that must be satisfied by a set of rules \( R \) if the rules in \( R \) are found to be confluent using our methods.

**Corollary 8.8** 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. \( \Box \)

**Corollary 8.9** 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 8.8. \( \Box \)

**Corollary 8.10** 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 \( r_j \in \text{Triggers}(r_i) \), by our conditions for noncommutativity (Lemma 8.1), \( r_i \) and \( r_j \) do not commute. Suppose, for the sake of a contradiction, that \( r_i \) and \( r_j \) are unordered. Then by Corollary 8.8 they must commute. \( \Box \)

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

We used our approach 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, i.e. confluence analysis successfully revealed errors in rule programming.

### 9 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 \( T' \)

\(^6\)Intuitively, a source of non-confluence can appear to "move around", requiring an iterative process of adding orderings (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.
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 \( T' \).

**Definition 9.1 (Significant Rules)** Let \( T' \subseteq T \) be a set of tables. The set of rules that are **significant with respect to** \( T' \), denoted \( \text{Sig}(T') \), is computed by the following algorithm:

\[
\text{Sig}(T') \leftarrow \{ r \in R | (I, t), (D, t), \text{or } (U, t, c) \text{ is in } \text{Performs}(r) \text{ for some } t \in T' \}
\]

repeat until unchanged:

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

That is, \( \text{Sig}(T') \) contains all rules that modify any table in \( T' \), along with (recursively) all rules that do not commute with rules in \( \text{Sig}(T') \). This algorithm determines whether rules commute using our conservative conditions for noncommutativity from Lemma 8.1. Hence, the user can influence the computation of \( \text{Sig}(T') \) by specifying that pairs of rules that appear noncommutative according to Lemma 8.1 actually do commute.

As in Confluence Theorem 8.7, partial confluence requires that rules are guaranteed to terminate. In this case, however, the rule set under consideration is \( \text{Sig}(T') \). Thus, before analyzing partial confluence, termination of the rules in \( \text{Sig}(T') \) must be established using the techniques of Section 7.\(^9\)

**Theorem 9.2 (Partial Confluence)** Let \( T' \subseteq T \) be a set of tables. Suppose the Confluence Requirement (Definition 8.5) holds for the rules in \( \text{Sig}(T') \) and there are no infinite paths in any execution graph for \( \text{Sig}(T') \). 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:** See Appendix A.2.

Hence, analyzing whether the rules in \( R \) are confluent with respect to \( T' \) requires first computing \( \text{Sig}(T') \), then considering each pair of unordered rules \( r_i \) and \( r_j \) in \( \text{Sig}(T') \): Sets \( R_1 \) and \( R_2 \) are built according to Definition 8.5 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 8.5 for confluence can be used here to establish partial confluence. Examples of partial confluence analysis are given in Section 10.1 below.

## 10 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**)

\(^9\)That is, even though the rules in \( \text{Sig}(T') \) are never processed on their own, it must be established that if they were processed on their own they would terminate. As in Section 8.3, this is necessary for Definition 8.5 to guarantee confluence.
while rules are being processed. When this is the case, the user may want to determine whether a rule set is \textit{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 triggered. Note that observable determinism and confluence are orthogonal properties: a rule set may be confluent but not observably deterministic or vice versa. (E.g., the rules in Example 4.2 are observably deterministic but non-confluent, while the rules in Example 4.3 are confluent but not observably deterministic.)

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

More formally, recall the definitions of Section 5. 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 $Uses_{obs}$ and $Performs_{obs}$ extend the definitions of $Uses$ and $Performs$ as follows. For every $r \in R$ such that $Observable(r)$, add $Obs.c$ to $Uses(r)$ and $(I, Obs)$ to $Performs(r)$. In addition, for every $r \in R$ such that $Observable(r)$, add to $Uses(r)$ every $t.c$ referenced in a top-level select operation in $r$'s action and, for every $(trans).c$ referenced in such a select operation, where $trans$ is one of inserted, deleted, new-updated, or old-updated, add $t.c$ to $Uses(r)$ for $r$'s triggering table $t$. Hereafter, for convenience we say that a rule $r$ is \textit{observable} if $Observable(r)$.

**Theorem 10.1 (Observable Determinism)** Suppose, using extended definitions $T_{obs}$, $C_{obs}$, $O_{obs}$, $Uses_{obs}$, 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 9.2) that the Confluence Requirement of Definition 8.5 holds for the rules in $Sig(\{Obs\})$ and there are no infinite paths in any execution graph for $Sig(\{Obs\})$. 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 $Uses_{obs}$ 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. $\square$

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 8.5 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 10.2** 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.\(^{10}\)

**Proof:** Since $r_i$ is observable, $\text{Obs.c} \in \text{Uses}(r_i)$ and $\langle I, \text{Obs} \rangle \in \text{Performs}(r_i)$; similarly for $r_j$. Therefore, by Definition 9.1, $r_i$ and $r_j$ are both in $\text{Sig}\{\{\text{Obs}\}\}$. In addition, by Lemma 8.1, $r_i$ and $r_j$ satisfy our conditions for noncommutativity. Suppose, for the sake of a contradiction, that $r_i$ and $r_j$ are unordered. $r_i$ and $r_j$ generate sets $R_1$ and $R_2$ (from Definition 8.5) 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, a contradiction. \(\boxdot\)

### 10.1 Examples

In Examples 4.1 and 4.2 there are no observable rules. Hence, in both cases $\text{Sig}\{\{\text{Obs}\}\} = \emptyset$, the rules are partially confluent with respect $\text{Obs}$ (vacuously by Theorem 9.2), and the rules are observably deterministic (by Theorem 10.1).

Now consider Example 4.3 in which there are three rules: $\text{good-sales}$, $\text{rank-raise}$, and $\text{new-rank}$. Since $\text{new-rank}$ is observable, $\text{Uses}_{\text{obs}}$ and $\text{Performs}_{\text{obs}}$ are derived from $\text{Uses}$ and $\text{Performs}$ as follows: $\text{Uses}(\text{new-rank})$ is extended to include $\text{Obs.c}$, $\text{emp.id}$, $\text{emp.rank}$, and $\text{emp.salary}$; $\text{Performs}(\text{new-rank})$ is extended to include $\langle I, \text{Obs} \rangle$. With these extensions, rule $\text{new-rank}$ no longer commutes with rule $\text{good-sales}$ (by condition 3 of Lemma 8.1) or with rule $\text{rank-raise}$ (also by condition 3). By Definition 9.1 of significant rules, $\text{Sig}\{\{\text{Obs}\}\} = \{\text{new-rank}, \text{good-sales}, \text{rank-raise}\}$. Hence, to analyze observable determinism, we analyze confluence for all three rules. When we considered confluence for these rules in Section 8.4, without the extended definitions of $\text{Uses}$ and $\text{Performs}$, we found them to be confluent. However, with the extended definitions this is not the case: unordered rules $\text{good-sales}$ and $\text{new-rank}$ do not commute, so by Corollary 8.8 we cannot guarantee confluence; similarly for unordered rules $\text{rank-raise}$ and $\text{new-rank}$. Therefore we determine that the rules are not observably deterministic.

Finally consider Example 4.4 in which there are three rules: $\text{bonus-rank}$, $\text{good-sales}$, and $\text{new-rank}$. $\text{Uses}_{\text{obs}}$ and $\text{Performs}_{\text{obs}}$ are extended for observable rule $\text{new-rank}$ as in the previous example. With these extensions, rules $\text{good-sales}$ and $\text{new-rank}$ no longer commute. By Definition 9.1, $\text{Sig}\{\{\text{Obs}\}\} = \{\text{new-rank}, \text{bonus-rank}, \text{good-sales}\}$. Hence, to analyze observable determinism, we analyze confluence for all three rules. The only pair of unordered rules in the set is $\text{bonus-rank}$ and $\text{good-sales}$. As in Section 8.4, $R_1$ and $R_2$ are constructed according to Definition 8.5, then Theorem 8.7 is applied to conclude that the rules are confluent. Therefore the rules are observably deterministic.

\(^{10}\)Note that this is not an if and only if condition: orderings between all pairs of observable rules does not necessarily guarantee observable determinism.
11 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 5 (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 (reasonable) 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.

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. the user 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.

### A Appendix

Here we provide the two (lengthy) proofs omitted from the body of the paper.

#### A.1 Confluence

To prove Lemma 8.6, we first introduce some additional notation and definitions. Let $S$ and $S'$ be execution graph states and let $r$ be a rule in $R$. We write $S \leadsto S'$ if $S'$ follows from $S$ by consideration of rule $r$, whether or not $r$ is triggered and eligible in $S$. If $r$ is not triggered in $S$, then the consideration of $r$ in $S$ is an identity transition, i.e. $S \leadsto S$. (This corresponds with the intuition that a non-triggered rule can have no effect.) With this notation we give a precise definition of commutativity.

**Definition A.1 (Commutativity)** Two rules $r_i$ and $r_j$ *commute* if for all execution graph states $S$, $S \leadsto S_i \leadsto S'$ if and only if $S \leadsto S_j \leadsto S'$.

If rule $r$ is triggered and eligible in state $S$, then we write $S \rightarrow S'$, which also denotes an execution graph edge (as defined in Section 8.3). Recall that $\rightarrow$ is the reflexive-transitive closure of $\rightarrow$.

Let a *path* be a sequence of rule considerations $S_0 \leadsto S_1 \leadsto S_2 \ldots S_{n-1} \leadsto S_n$.\(^\text{11}\) In the proofs below, the start state $S_0$ always is fixed and obvious from context, so we generally abbreviate a path as just the sequence of considered rules $\langle r_1, \ldots, r_n \rangle$. Two paths $P$ and $Q$ are equivalent, denoted $P \equiv Q$, if they begin in the same state $S_0$ and end in the same state $S_n$. A path is *valid* if all

\(^{11}\)Note that a path as defined here is not necessarily a path in an execution graph, since there are no edges in execution graphs that correspond to consideration of ineligible rules.
rules are eligible when they are considered; i.e. for all i, 1 ≤ i ≤ n, \( S_{i-1} \rightarrow_i S_i \). Hence, a valid path is a path in an execution graph. Finally, given two sequences of rules \( A \) and \( B \), \( A; B \) is the path consisting of consideration of the rules in \( A \) followed by consideration of the rules in \( B \).

The proof of Lemma 8.6 uses the following simple lemma about paths. For a path \( P \), let \( \overline{P} \) denote \( P \) with all considerations of non-triggered rules deleted.

**Lemma A.2** Let \( P \) be a path and assume that for each rule in \( P \), either the rule is not triggered when it is considered, or the rule is eligible when it is considered. Then \( \overline{P} \equiv P \) and \( \overline{P} \) is a valid path.

**Proof:** Obvious.

The following two lemmas are used several times in the proof of Lemma 8.6.

**Lemma A.3** Let \( r_i \), \( r_j \), and \( r \) be rules such that \( r_i \) and \( r_j \) are unordered, and let \( R_1 \) and \( R_2 \) be the sets of rules constructed from \( r_i \) and \( r_j \) in Definition 8.5. If \( r \in (R_1 \cup R_2) \setminus \{r_i, r_j\} \), then either \( r > r_i \) or \( r > r_j \).

**Proof:** Recall the algorithm in Definition 8.5 for constructing \( R_1 \) and \( R_2 \). The proof is by induction on the number of loop iterations required by this algorithm to add \( r \) to either \( R_1 \) or \( R_2 \). As the base case, let the number of iterations be 0. Then \( R_1 = \{r_i\} \) and \( R_2 = \{r_j\} \), so the result holds vacuously. Assume the result holds for \( n \) iterations. If \( r \) is added in iteration \( n + 1 \), then by definition there is some \( r' \) added previously such that \( r > r' \). If \( r' = r_i \) or \( r' = r_j \) then \( r > r_i \) or \( r > r_j \). Otherwise, by the induction hypothesis \( r' > r_i \) or \( r' > r_j \), so by transitivity \( r > r_i \) or \( r > r_j \). □

**Lemma A.4** Let \( r_i \) and \( r_j \) be unordered rules, let \( R_1 \) and \( R_2 \) be the sets of rules constructed from \( r_i \) and \( r_j \) in Definition 8.5, and let \( r \) and \( r' \) be rules such that \( r > r' \) and \( r' \in R_1 \cup R_2 \). Then \( r \neq r_i \) and \( r \neq r_j \).

**Proof:** For the sake of a contradiction, suppose \( r = r_i \). Since \( r > r' \), \( r' \neq r_i \) (since a rule cannot have higher priority than itself) and \( r' \neq r_j \) (since \( r_i \) and \( r_j \) are unordered). Hence, by Lemma A.3, either \( r' > r_i \) or \( r' > r_j \). By transitivity, then, either \( r > r_i \) (contradicting the fact that rules cannot have higher priority than themselves) or \( r > r_j \) (contradicting the assumption that \( r_i \) and \( r_j \) are unordered). Thus \( r \neq r_i \). A parallel argument shows \( r \neq r_j \). □

**Lemma 8.6 (Confluence Lemma)** Suppose the Confluence Requirement (Definition 8.5) 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 \( S_i \leadsto S' \) and \( S_j \leadsto S' \).

**Proof:** Let \( r_i \) and \( r_j \) label edges \( S \rightarrow S_i \) and \( S \rightarrow S_j \) respectively, i.e. \( S \stackrel{r_i}{\rightarrow} S_i \) and \( S \stackrel{r_j}{\rightarrow} S_j \), where \( r_i \) and \( r_j \) are distinct unordered rules.\(^{12}\) We must prove that there is a state \( S' \) such that \( S_i \leadsto S' \)

\(^{12}\)Other than that, rules \( r_i \) and \( r_j \) are chosen arbitrarily, since by Observation 8.2 any pair is very likely to be possible.
and \( S_j \not\rightarrow S' \). We do this by exhibiting valid paths \( \overline{P}_1 = S \xrightarrow{r_1} S_i \not\rightarrow S' \) and \( \overline{P}_2 = S \xrightarrow{r_2} S_j \not\rightarrow S' \). The proof involves three steps. First, we give an algorithm to construct initial paths \( P_1 \) and \( P_2 \). Second, we extend \( P_1 \) and \( P_2 \) with some additional rules so that every rule in \( P_1 \) and \( P_2 \) is either eligible or not triggered when it is considered; by Lemma A.2, at this point \( \overline{P}_1 \) and \( \overline{P}_2 \) are valid, \( P_1 \equiv \overline{P}_1 \), and \( P_2 \equiv \overline{P}_2 \). Third, we show that \( \overline{P}_1 \) and \( \overline{P}_2 \) have the same last state, i.e. \( \overline{P}_1 \equiv \overline{P}_2 \), which proves the lemma.

Let \( R_1 \) and \( R_2 \) be the sets of rules constructed from \( r_i \) and \( r_j \) in Definition 8.5, let \( R_1' = R_1 - \{r_i\} \), and let \( R_2' = R_2 - \{r_j\} \). The following algorithm constructs initial paths \( P_1 \) and \( P_2 \).

\[
\begin{align*}
A &\leftarrow \langle \rangle \\
\text{while} &\text{ there is a rule } r \in R_1' \text{ eligible in the last state of path } \langle r_i \rangle; \ A \text{ do } A \leftarrow A; \langle r \rangle \\
B &\leftarrow \langle \rangle \\
\text{while} &\text{ there is a rule } r \in R_2' \text{ eligible in the last state of path } \langle r_j \rangle; \ B \text{ do } B \leftarrow B; \langle r \rangle \\
P_1 &\leftarrow \langle r_i \rangle; A; \langle r_j \rangle; B \\
P_2 &\leftarrow \langle r_j \rangle; B; \langle r_i \rangle; A
\end{align*}
\]

Before proceeding, we establish an important property of paths \( P_1 \) and \( P_2 \). Consider path \( P_1 \) and let \( S_0 \) be the last state of path \( \langle r_i \rangle; A \). We show that there is no rule \( r \) such that \( r \in R_1' \) and \( r \) is triggered in \( S_0 \). (A symmetric argument shows that on path \( P_2 \) there is no rule \( r \) such that \( r \in R_2' \) and \( r \) is triggered in the last state of \( \langle r_j \rangle; B \).) Clearly, there is no rule \( r \) such that \( r \in R_1 \) and \( r \) is eligible in \( S_0 \), because this would violate the termination condition of the first while loop in the algorithm. Suppose \( r \in R_1' \) and \( r \) is triggered but not eligible in \( S_0 \). Then there is some \( r' \) such that \( r' > r \) and \( r' \) is eligible in \( S_0 \). We show \( r' \in R_1' \), a contradiction. By \( R_1' = R_1 - \{r_i\} \) and Definition 8.5 of \( R_1 \), \( r' \in R_1' \) if:

1. there is a rule \( r_1 \in R_1 \) such that \( r' \in \text{Triggers}(r_1) \),
2. there is a rule \( r_2 \in R_2 \) such that \( r' > r_2 \), and
3. \( r' \neq r_i \) and \( r' \neq r_j \).

For (1), note that by Lemma A.3, \( r' > r_i \) or \( r' > r_j \). Thus, \( r' \) could not be triggered in the initial state \( S \), or else one of \( r_i \) or \( r_j \) would not be eligible in \( S \). So \( r' \) must be triggered by some rule in \( \langle r_i \rangle; A \), which implies \( r' \in \text{Triggers}(r_1) \) for some \( r_1 \in R_1 \). For (2), note that \( r \in R_1' \) implies that there is some \( r_2 \in R_2 \) such that \( r > r_2 \); by transitivity \( r' > r_2 \). For (3), \( r' > r \) and \( r \in R_1' \) together imply \( r \neq r_i \) and \( r \neq r_j \) by Lemma A.4. Therefore \( r' \in R_1' \) and \( r' \) is eligible in \( S_0 \). But, as noted above, this is a contradiction, since the while loop adds all eligible rules in \( R_1' \). We conclude that there are no rules in \( R_1' \) triggered in \( S_0 \). Similarly, there are no rules in \( R_2' \) triggered in the last state of \( \langle r_j \rangle; B \).

As the second step of the proof, we show that \( \overline{P}_1 \) and \( \overline{P}_2 \) are valid. To do this, we modify \( P_1 \) and \( P_2 \) so that every rule is either eligible or not triggered when it is considered—that is, there should be no rules that are triggered but not eligible when they are considered. We show how to modify path \( P_1 \); the construction for \( P_2 \) is symmetric. In path \( P_1 \), every rule in the portion \( \langle r_i \rangle; A \) is eligible, because \( r_i \) is eligible in the initial state \( S \) and every rule added by while loop is eligible.
Thus, we consider latter portion \(\langle r_j \rangle; B\). As above, let \(S_0\) be the last state of \(\langle r_i \rangle; A\). Assume, for
the sake of a contradiction, that \(r_j\) is triggered in \(S_0\) but is not eligible. Then there must be a rule
\(r\) such that \(r > r_j\) and \(r\) is eligible in \(S_0\). We show \(r \in R'_1\), a contradiction. Consider (1), (2), and
(3) above. For (1), note that \(r\) cannot be triggered in the initial state \(S\) since \(r_j\) is eligible in \(S\) and
\(r > r_j\). So \(r\) is triggered by a rule in \(\langle r_i \rangle; A\), which implies \(r \in \text{Triggers}(r_1)\) for some \(r_1 \in R_1\). For
(2), note that \(r > r_j\) and \(r_j \in R_2\). For (3), \(r \neq r_i\) and \(r \neq r_j\) because \(r > r_j\). Thus, \(r \in R'_1\). But \(r\)
cannot be in \(R'_1\), because there are no eligible rules in \(R'_1\) at \(S_0\). We conclude that if \(r_j\) is triggered
in \(S_0\), then \(r_j\) is eligible.

Now we must guarantee that rules in \(B\) are either eligible or not triggered when they are
considered on path \(P_1\). For this property, it may be necessary to add more rules from \(R'_2\) to \(B\).
We simultaneously consider paths \(P_1 = \langle r_i \rangle; A; \langle r_j \rangle; B\) and \(P_2 = \langle r_j \rangle; B; \langle r_i \rangle; A\). Let \(r\) be a rule in
\(B\), let \(S_1\) be the state where \(r\) is considered in \(P_1\), and let \(S_2\) be the state where \(r\) is considered in
\(P_2\). By construction, \(r\) is eligible in \(S_2\), because the second while loop adds only eligible rules to
\(B\). Suppose \(r\) is triggered but not eligible in \(S_1\). Then there is some rule \(r'\) such that \(r' > r\) and \(r'\)
is eligible in \(S_1\). Clearly \(r'\) is not triggered in \(S_2\), or else \(r\) could not be eligible in \(S_2\). So we can
insert \(r'\) before \(r\) in \(B\) with two effects: an eligible rule is added to \(P_1\) and a non-triggered rule is
added to \(P_2\).

This observation motivates the following modification to \(B\): Repeatedly select the first rule \(r\)
from \(B\) that is triggered but not eligible in the state \(S_1\) where \(r\) is considered in \(P_1\). Insert a rule
\(r'\) before \(r\) in \(B\), where \(r' > r\) and \(r'\) is eligible in \(S_1\). By repeating this procedure for all rules with
precedence over \(r\), eventually \(r\) becomes an eligible rule in \(P_1\). Thus, eventually all rules in \(P_1\) are
either triggered and eligible or not triggered when they are considered. This procedure terminates
because at each step \(P_1\) is extended by an eligible rule; since (by assumption) rule processing always
terminates, only a finite number of rules can be added. Note that when this procedure terminates,
\(P_1\) is valid, and all rules added to \(B\) are not triggered when they are considered in \(P_2\).

Before proceeding to the third step, we show that for every \(r'\) added to \(B\) by the procedure
above, \(r' \in R'_2\). In \(P_1\), \(r'\) can be triggered in three ways: it is triggered in the initial state \(S_1\), it
is triggered by a rule in \(\langle r_i \rangle; A\), or it is triggered by a rule in \(\langle r_j \rangle; B\). We consider each of these
separately:

- Suppose \(r'\) is triggered in the initial state \(S_1\). We know \(r' > r\) and \(r \in R'_2\), so by Lemma A.3
  and transitivity, \(r' > r_i\) or \(r' > r_j\). But \(r_i\) and \(r_j\) are eligible in \(S\), so \(r'\) cannot be triggered
  in \(S\).

- Suppose \(r'\) is triggered by a rule in \(\langle r_i \rangle; A\). To derive a contradiction, we first show \(r' \in R'_1\).
  For (1), every rule in \(\langle r_i \rangle; A\) is in \(R_1\), implying that \(r' \in \text{Triggers}(r_1)\) for some \(r_1 \in R_1\). For
  (2), \(r' > r\) and \(r \in R_2\). For (3), \(r' > r\) and \(r \in R_2\) together imply \(r' \neq r_i\) and \(r' \neq r_j\) by
  Lemma A.4. Thus, \(r' \in R'_1\). But there are no eligible rules of \(R'_1\) in the last state of \(\langle r_i \rangle; A\).
  Hence \(r'\) is not triggered by a rule in \(\langle r_i \rangle; A\).

- Then \(r'\) must be triggered by a rule in \(\langle r_j \rangle; B\). This implies \(r' \in R'_2\). For (1), every rule in
  \(\langle r_j \rangle; B\) is in \(R_2\), implying \(r' \in \text{Triggers}(r_2)\) for some \(r_2 \in R_2\). For (2), because \(r \in R'_2\) there

28
is a rule \( r_1 \) such that \( r_1 \in R_1 \) and \( r > r_1 \); by transitivity \( r' > r_1 \). For \((3), r' > r \) and \( r \in R_2 \) together imply \( r' \neq r \) and \( r' \neq r_2 \) by Lemma A.4.

For a path \( P \), let the set of rules appearing on \( P \) be denoted \( \text{Rules}(P) \). We have shown that rules can be added to \( R \) such that \( \overline{P_1} \) is valid, \( \text{Rules}(A) \subseteq R_1 \), \( \text{Rules}(B) \subseteq R_2 \), and the modifications to \( R \) add only non-triggered rules to \( P_2 \). A symmetric argument adds rules to \( A \) such that \( \overline{P_2} \) is valid, \( \text{Rules}(A) \subseteq R_1 \), \( \text{Rules}(B) \subseteq R_2 \), and the modifications to \( A \) add only non-triggered rules to \( P_1 \). Thus, we can construct \( P_1 \) and \( P_2 \) such that \( P_1 = \langle r_1 \rangle ; A ; \langle r_2 \rangle ; B ; \langle r_3 \rangle \); \( P_2 = \langle r_2 \rangle ; B ; \langle r_1 \rangle ; A \), \( \overline{P_1} \) and \( \overline{P_2} \) are valid, \( \text{Rules}(A) \subseteq R_1 \), and \( \text{Rules}(B) \subseteq R_2 \).

To complete the proof, we show \( \overline{P_1} \equiv \overline{P_2} \), i.e. \( \overline{P_1} \) and \( \overline{P_2} \) have the same last state. By Lemma A.2, \( P_1 \equiv \overline{P_1} \) and \( P_2 \equiv \overline{P_2} \), so it suffices to show \( P_1 \equiv P_2 \). Consider any path with two consecutive rules, one from \( R_1 \) and the other from \( R_2 \). Since \( R_1 \) and \( R_2 \) are pairwise commutative, we can interchange the two rules without changing the last state. Path \( \langle r_1 \rangle ; A \) consists entirely of rules from \( R_1 \), while path \( \langle r_2 \rangle ; B \) consists entirely of rules from \( R_2 \). Hence, by repeatedly interchanging consecutive rules from \( R_1 \) and \( R_2 \), we can prove \( \langle r_1 \rangle ; A ; \langle r_2 \rangle ; B \equiv \langle r_2 \rangle ; B ; \langle r_1 \rangle ; A \), and consequently \( P_1 \equiv P_2 \). □

### A.2 Partial Confluence

To prove Theorem 9.2, we use some notation introduced in Appendix A.1 and give some additional notation and definitions. Our method for establishing partial confluence uses the confluence of a set of rules \( \text{Sig}(T') \subseteq R \) to prove partial confluence for the rules in \( R \). Confluence is a property of execution graphs, but \( \text{Sig}(T') \) and \( R \) may not have the same execution graphs. Thus, to prove the correctness of our method, we need to translate between execution graphs of \( \text{Sig}(T') \) and execution graphs of \( R \). As a first step, we subscript \( \sim \) and \( \to \) with sets of rules. Let \( R' \subseteq R \). We write \( S \sim_{R'} S' \) if \( S' \) follows from \( S \) by consideration of rule \( r \) when the set of rules is \( R' \). We write \( S \to_{R'} S' \) if, additionally, \( r \) is triggered and eligible in state \( S \). Since \( R \) denotes the entire set of rules, for consistency with previous sections we write \( S \sim S' \) and \( S \to S' \) for \( S \sim_{R} S' \) and \( S \to_{R} S' \), respectively.

Let \( P = S_0 \sim_{R'} S_1 \sim_{R'} \cdots \sim_{R'} S_n \) be any path. \( P \) is a path with respect to \( R' \) if each \( r_i \) is in \( R' \) and for each state \( S_i = (D_i, TR_i) \), the set of triggered rules in \( TR_i \) is a subset of \( R' \). If \( S_{i-1} \to_{R'} S_i \), \( 1 \leq i \leq n \), then \( P \) is a valid path with respect to \( R' \), and \( P \) is a path in an execution graph for \( R' \) (recall Section A.1). Let \( S \) be any state. A rule \( r \) is eligible with respect to \( R' \) in \( S \) if \( r \in R' \), \( r \) is triggered in \( S \), and there is no rule \( r' \) such that \( r' \in R' \), \( r' > r \), and \( r' \) is triggered in \( S \). The following lemma provides a way to convert certain paths with respect to \( R \) into valid paths with respect to \( R' \).

**Lemma A.5** Let \( R' \subseteq R \), let \( (D_0, TR_0) \sim_{R'} (D_n, TR_n) \) be a path (with respect to \( R \)), and suppose \( r_i \) is eligible with respect to \( R' \) in state \( (D_{i-1}, TR_{i-1}) \), \( 1 \leq i \leq n \). Then \( (D_0, TR_0 \cap R') \to_{R'} (D_n, TR_n \cap R') \), i.e. \( (D_0, TR_0 \cap R') \sim_{R'} (D_n, TR_n \cap R') \) is a valid path with respect to \( R' \).
**Proof:** Let $TR'_i = TR_i \cap R'$. Rule $r_i$ is eligible in $(D_{i-1}, TR'_i)$, $1 \leq i \leq n$, since $r_i$ is eligible with respect to $R'$ in $(D_{i-1}, TR_{i-1})$. Therefore, $(D_{i-1}, TR'_i) \Rightarrow_{R'} (D_i, TR'_i)$, $1 \leq i \leq n$. □

The following lemma provides the key construction for proving Theorem 9.2. This lemma shows that any path in an execution graph for $R$ can be transformed by commuting rules into an equivalent path $RS$, $RN$ such that $RS$ contains only rules from $\text{Sig}(T')$, $RN$ contains only rules from $R - \text{Sig}(T')$, and all rules in $RS$ are eligible with respect to $\text{Sig}(T')$.

**Lemma A.6** Let $T' \subseteq T$ be a set of tables and assume there are no infinite paths in any execution graph for $\text{Sig}(T')$. If $P$ is a path in an execution graph for $R$ ending in a final state then there exists a path $P'$ such that

1. $P' = RS; RN$, where Rules$(RS) \subseteq \text{Sig}(T')$ and Rules$(RN) \subseteq R - \text{Sig}(T')$,
2. $P' \equiv P$,
3. in $P'$, every rule in $\text{Sig}(T')$ is eligible with respect to $\text{Sig}(T')$ when it is considered, and
4. $TR \cap \text{Sig}(T') = \emptyset$, where $(D, TR)$ is the last state of $RS$.

**Proof:** Let path $P'$ be constructed by the following algorithm:

\[
P' \leftarrow P
\]
\[
\text{while } P' \neq RS; RN \text{ where Rules}(RS) \subseteq \text{Sig}(T') \text{ and Rules}(RN) \subseteq R - \text{Sig}(T') \text{ do}
\]
\[
\quad \text{let } P' = RS; RN; \langle r_i \rangle; A;
\]
\[
\quad \text{where Rules}(RS) \subseteq \text{Sig}(T'), \text{Rules}(RN) \subseteq R - \text{Sig}(T'), \vert RN \vert > 0, \text{ and } r_i \in \text{Sig}(T')
\]
\[
\quad \text{if } r_i \text{ is eligible with respect to } \text{Sig}(T') \text{ in last state of } RS \text{ then}
\]
\[
\quad \quad [a] \quad P' \leftarrow RS; \langle r_i \rangle; RN; A
\]
\[
\quad \text{else if } r_i \text{ is not triggered in the last state of } RS \text{ then}
\]
\[
\quad \quad [b] \quad P' \leftarrow RS; RN; A
\]
\[
\quad \text{else}
\]
\[
\quad \quad \text{let } r_j > r_i \text{ be eligible with respect to } \text{Sig}(T') \text{ in the last state of } RS
\]
\[
\quad \quad [c] \quad P' \leftarrow RS; \langle r_j \rangle; RN; \langle r_i \rangle; A
\]
\[
\quad \text{while } P' = RS; RN \text{ and } r \in \text{Sig}(T') \text{ is eligible with respect to } \text{Sig}(T') \text{ in the last state of } RS \text{ do}
\]
\[
\quad \quad P' \leftarrow RS; (r); RN
\]

We must show that (1)–(4) hold and that the algorithm for constructing $P'$ always terminates. We first show that (1)–(3) hold after the first while loop and that the loop terminates. We then show that the second while loop preserves (1)–(3), establishes (4), and terminates.

Consider the first while loop. Note that if the loop condition is true, then $P'$ must have the form described by the first let clause; i.e. $P'$ must begin with a (possibly empty) set of rules $RS$ from $\text{Sig}(T')$, followed by a non-empty set of rules $RN$ from $R - \text{Sig}(T')$, followed by at least one rule $r_i$ from $\text{Sig}(T')$. From the termination condition of this loop, it is clear that (1) holds when the loop terminates. We prove by induction on the number of iterations that (2) and (3) also hold when the loop terminates. For the base case, (2) holds after 0 iterations since $P' = P$. Clearly (3) holds as well, since $P$ is a valid path for $R$. For the induction step, let $P'_n = RS_n; RN_n; \langle r_i \rangle; A_n$ be
Let \( P'_{n+1} \) be the result of interchanging \( r_i \) and \( RN \) (branch [a]). Recall from Definition 9.1 of \( \text{Sig}(T') \) that rules in \( R - \text{Sig}(T') \) commute with rules in \( \text{Sig}(T') \); therefore \( r_i \) commutes with all rules in \( RN \). Hence:

\[
P'_n = RS_n; RN_n; \langle r_i \rangle; A_n
\]

This shows that (2) holds. For (3), we must show that every rule in \( \text{Sig}(T') \) is eligible with respect to \( \text{Sig}(T') \) when it is considered in \( P'_{n+1} \). A rule in \( RS_n \) is considered in the same state in \( P'_{n+1} \) as in \( P_n \), so rules in \( RS_n \) are eligible with respect to \( \text{Sig}(T') \) when considered in \( P'_{n+1} \). By the condition for the [a] branch, \( r_i \) is eligible with respect to \( \text{Sig}(T') \) in the last state of \( RS_n \). By definition, no rules in \( RN \) are in \( \text{Sig}(T') \). Finally, because \( r_i \) commutes with rules in \( RN \), the first state of \( A_n \) is the same in \( P'_{n} \) and \( P'_{n+1} \), so each rule in \( A_n \) is considered in the same state in \( P'_{n} \) and \( P'_{n+1} \). Therefore, rules in \( A_n \) that are also in \( \text{Sig}(T') \) are eligible with respect to \( \text{Sig}(T') \) in \( P'_{n+1} \).

Now suppose \( P'_{n+1} = RS_n; RN_n; A_n \) (branch [b]). Let \( S \) be the last state of \( RS_n \). By the condition for the [b] branch, we know that \( r_i \) is not triggered in \( S \), i.e. \( S \not \approx S \). Hence:

\[
P'_n = RS_n; RN_n; \langle r_i \rangle; A_n
\]

This shows that (2) holds. For (3), we must show that every rule in \( \text{Sig}(T') \) is eligible with respect to \( \text{Sig}(T') \) when it is considered in \( P'_{n+1} \). A rule in \( RS_n \) is considered in the same state in \( P'_{n+1} \) as in \( P_n \), so rules in \( RS_n \) are eligible with respect to \( \text{Sig}(T') \) when considered in \( P'_{n+1} \). By definition, no rules in \( RN \) are in \( \text{Sig}(T') \). Finally, by the equivalences above, the first state of \( A_n \) is the same in \( P'_{n} \) and \( P'_{n+1} \), so each rule in \( A_n \) is considered in the same state in \( P'_{n} \) and \( P'_{n+1} \). Therefore, rules in \( A_n \) that are also in \( \text{Sig}(T') \) are eligible with respect to \( \text{Sig}(T') \) in \( P'_{n+1} \).

For the last case, suppose \( P'_{n+1} = RS_n; \langle r_j \rangle; RN_n; \langle r_i \rangle; A_n \) (branch [c]). Let \( S \) be the last state of \( RS_n \) and let \( S' \) be the last state of \( RN_n \). The new rule \( r_j \) must exist, since if \( r_i \) is triggered but not eligible with respect to \( \text{Sig}(T') \) in \( S \), then there is a rule \( r_j \) such that \( r_j > r_i \) and \( r_j \) is eligible with respect to \( \text{Sig}(T') \) in \( S \). Now, \( r_j \) cannot be triggered in \( S' \), or else \( r_i \) is not eligible with respect to \( \text{Sig}(T') \) in \( S' \). Note also that \( r_j \) commutes with rules in \( RN_n \), since \( r_j \in \text{Sig}(T') \) and \( \text{Rules}(RN_n) \subseteq R - \text{Sig}(T') \). Hence:

\[
P'_n = RS_n; RN_n; \langle r_j \rangle; A_n
\]

This shows that (2) holds. For (3), we must show that every rule in \( \text{Sig}(T') \) is eligible with respect to \( \text{Sig}(T') \) when it is considered in \( P'_{n+1} \). A rule in \( RS_n \) is considered in the same state in \( P'_{n+1} \) as in \( P_n \), so rules in \( RS_n \) are eligible with respect to \( \text{Sig}(T') \) when considered in \( P'_{n+1} \). By definition, no rules in \( RN \) are in \( \text{Sig}(T') \). Finally, by the equivalences above, the first state of \( A_n \) is the same in \( P'_{n} \) and \( P'_{n+1} \), so each rule in \( A_n \) is considered in the same state in \( P'_{n} \) and \( P'_{n+1} \). Therefore, rules in \( A_n \) that are also in \( \text{Sig}(T') \) are eligible with respect to \( \text{Sig}(T') \) in \( P'_{n+1} \).

For the last case, suppose \( P'_{n+1} = RS_n; \langle r_j \rangle; RN_n; \langle r_i \rangle; A_n \) (branch [c]). Let \( S \) be the last state of \( RS_n \) and let \( S' \) be the last state of \( RN_n \). The new rule \( r_j \) must exist, since if \( r_i \) is triggered but not eligible with respect to \( \text{Sig}(T') \) in \( S \), then there is a rule \( r_j \) such that \( r_j > r_i \) and \( r_j \) is eligible with respect to \( \text{Sig}(T') \) in \( S \). Now, \( r_j \) cannot be triggered in \( S' \), or else \( r_i \) is not eligible with respect to \( \text{Sig}(T') \) in \( S' \). Note also that \( r_j \) commutes with rules in \( RN_n \), since \( r_j \in \text{Sig}(T') \) and \( \text{Rules}(RN_n) \subseteq R - \text{Sig}(T') \). Hence:

\[
P'_n = RS_n; RN_n; \langle r_j \rangle; A_n
\]
This shows that (2) holds. For (3), we must show that every rule in \( \text{Sig}(T') \) is eligible with respect to \( \text{Sig}(T') \) when it is considered in \( P'_{n+1} \). A rule in \( RS_n \) is considered in the same state in \( P'_{n+1} \) as in \( P'_n \), so rules in \( RS_n \) are eligible with respect to \( \text{Sig}(T') \) when considered in \( P'_{n+1} \). By the choice of \( r_j \) in the let clause preceding \([c]_j \), \( r_j \) is eligible with respect to \( \text{Sig}(T') \) in the last state of \( RS_n \).

By definition, no rules in \( RN_n \) are in \( \text{Sig}(T') \). Finally, by the equivalences above, the first state of \( \langle r_i \rangle \); \( A_n \) is the same in \( P'_n \) and \( P'_{n+1} \), so each rule in \( \langle r_i \rangle \); \( A_n \) is considered in the same state in \( P'_n \) and \( P'_{n+1} \). Therefore, rules in \( \langle r_i \rangle \); \( A_n \) that are also in \( \text{Sig}(T') \) are eligible with respect to \( \text{Sig}(T') \) in \( P'_{n+1} \).

We have shown that (1)–(3) hold after the first while loop. Next we show that the first while loop always terminates. In iteration \( n \), either \( RS_n \) is extended by one rule, or a rule not in \( RS_n \) is deleted from the path. Hence, to prove termination it suffices to show that \( RS_n \) cannot be infinite. By (3) and the fact that \( \text{Rules}(RS_n) \subseteq \text{Sig}(T') \), we know that every rule in \( RS_n \) is eligible with respect to \( \text{Sig}(T') \). Therefore, by Lemma A.5, there is a path \( RS'_n \) that is valid with respect \( \text{Sig}(T') \) and \( |RS'_n| = |RS_n| \). By assumption, there are no infinite paths in any execution graph for \( \text{Sig}(T') \); hence \( RS'_n \) must be finite, implying that \( RS_n \) is finite as well. We conclude that the first while loop terminates.

Now consider the second while loop. Condition (1) obviously holds throughout loop execution. We prove by induction on the number of loop iterations that (2) and (3) also hold. The result is trivial for 0 iterations. Let \( P'_{n} = RS; RN \) be \( P' \) after \( n \) iterations of the second while loop. Assume that there is a rule \( r \in \text{Sig}(T') \) eligible with respect to \( \text{Sig}(T') \) in the last state of \( RS \). Recall that no rules are triggered in the last state of \( P \) because \( P \) ends in a final state. Since \( P'_{n} = P \), no rules are triggered in the last state of \( P'_{n} \). Therefore:

\[
\begin{align*}
P'_n &= RS; RN \\
     &\equiv RS; RN; \langle r \rangle & \text{since } r \text{ is not triggered after } RN \\
     &\equiv RS; \langle r \rangle; RN & \text{by commutativity of } r \text{ with rules in } RN \\
     &= P'_{n+1}
\end{align*}
\]

This shows that (2) holds. For (3), note that all rules in \( RS \) are considered in the same state in both \( P'_n \) and \( P'_{n+1} \), by definition \( r \) is eligible with respect to \( \text{Sig}(T') \) in the last state of \( RS \), and no rules in \( RN \) are in \( \text{Sig}(T') \). Finally, note that the second while loop does not terminate until (4) holds. To complete the proof, we must show that the second while loop always terminates. This directly parallels the proof of termination for the first while loop. 

**Theorem 9.2 (Partial Confluence)** Let \( T' \subseteq T \) be a set of tables. Suppose the Confluence Requirement (Definition 8.5) holds for the rules in \( \text{Sig}(T') \) and there are no infinite paths in any execution graph for \( \text{Sig}(T') \). 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:** Let \( P_1 \) and \( P_2 \) be any two execution graph paths leading to final states \( F_1 \) and \( F_2 \), respectively. Let \( P'_1 = \langle RS_1; RN_1 \rangle \) and \( P'_2 = \langle RS_2; RN_2 \rangle \) be the paths of Lemma A.6 for \( P_1 \) and \( P_2 \), respectively. By part (2) of Lemma A.6, \( P'_1 \equiv P_1 \) and \( P'_2 \equiv P_2 \). We show that the tables in \( T' \) are identical in the last states of \( RS_1 \) and \( RS_2 \). Because none of the rules in \( RN_1 \) or \( RN_2 \) modify the
tables in $T'$ (by Definition 9.1 of $Sig(T')$), it then follows that the tables in $T'$ are identical in the last states of $P_1$ and $P_2$. Consequently, the tables in $T'$ are identical in the last states of $P_1$ and $P_2$, which are $F_1$ and $F_2$.

Consider $RS_1$. By Lemma A.5 and part (3) of Lemma A.6, a path $RS_1'$ that is valid with respect to $Sig(T')$ can be constructed from $RS_1$. Let $(D_1, TR_1)$ be the last state of $RS_1$ and let $(D_1, TR_1')$ be the last state of $RS_1'$. By Lemma A.5 and part (4) of Lemma A.6, $TR_1' = TR_1 \cap Sig(T') = \emptyset$, so $RS_1'$ ends in final state $(D_1, \emptyset)$. A symmetric argument shows that a path $RS_2'$ that is valid with respect to $Sig(T')$ can be constructed from $RS_2$, where $RS_2$ ends in a state $(D_2, TR_2)$ and $RS_2'$ ends in final state $(D_2, \emptyset)$. Paths $RS_1'$ and $RS_2'$ have the same start state, which is produced by the same initial operations producing the start state of paths $P_1$ and $P_2$. Hence, since paths $RS_1'$ and $RS_2'$ both end in a final state, $RS_1'$ and $RS_2'$ are two paths in an execution graph for $Sig(T')$. By assumption, Definition 8.5 holds for the rules in $Sig(T')$ and there are no infinite paths in any execution graph for $Sig(T')$. Hence, by Confluence Theorem 8.7, every execution graph for $Sig(T')$ has exactly one final state. Therefore $D_1 = D_2$, which shows that the tables in $T'$ are identical in the last states of $RS_1$ and $RS_2$. □

References


