# University of Florida

Computer and Information Science and Engineering

## A Confluent Rule Execution Model for Active Databases

**S-K. Kim**
**S. Chakravarthy**

EMAIL: sharma@cis.ufl.edu

WWW: http://www.cis.ufl.edu/~sharma

**Tech. Report UF-CIS-TR-95-032**
**(Submitted for publication)**

October 1995

(This work is partly supported by the Office of Naval Research and the Navy Command, Control and Ocean Surveillance Center RDT&E Division, and by the Rome Laboratory.)


**Computer and Information Science and Engineering Department**
**E301 Computer Science and Engineering Building**
**University of Florida, PO Box 116120**
**Gainesville, Florida 32611-6120**

# Contents

# A Confluent Rule Execution Model for Active Databases*

**Seung-Kyum Kim** **Sharma Chakravarthy**

Database Systems Research and Development Center
Department of Computer and Information Science and Engineering
University of Florida, Gainesville, FL 32611
Email: {skk,sharma}@cis.ufl.edu

**Abstract**

In this paper we propose a new rule execution model along with priority specification schemes to achieve confluent rule execution in active databases. Our model employs prioritization to resolve conflicts between rules, and uses a rule scheduler based on the topological sort to achieve correct confluent rule executions. Given a rule set, a trigger graph and a dependency graph are built from the information obtained by analyzing the rule set at compile time. The two graphs are combined to form a priority graph, on which the user is requested to specify priorities only if there exists dependencies in the dependency graph. From the priority graph, an execution graph is derived for every user transaction that triggers one or more rules. The rule scheduler uses the execution graph. Our model also correctly handles the situation where trigger paths of rules triggered by a user transaction are overlapping, which existing models are unable to handle. We prove that our model achieves maximum parallelism in rule executions.

## 1 Introduction

Incorporating ECA rules (Event-Condition-Action rules) enhances the functionality of traditional database systems significantly [C+89, GJ91, AMC93, HW93]. Also, ECA rules provide flexible alternatives for implementing many database features, such as integrity constraint enforcement, that are traditionally hard-wired into a DBMS [CW90, SJGP90, WF90, CW91, WCL91].

An ECA-rule consists of three parts: *event*, *condition*, and *action*. Execution of ECA rules goes through three phases: event detection, condition test, and execution of action. An event can be a data manipulation or retrieval operation, a method invocation in Object-Oriented databases, a signal from timer or the users, or a combination thereof. An active database system monitors occurrences of events pre-specified by ECA rules. Once specified events have occurred, the condition part of the relevant rule is tested. If the test is satisfied, the rule's action part can be executed. In Sentinel [CKAK94], a rule is said to be *triggered* when the rule has passed the event detection phase; that is, when one or more events which the rule is waiting for have occurred. When an ECA rule has passed the condition test phase, it is said to be *eligible* for execution.[1] In this paper,

*This work is partly supported by the Office of Naval Research and the Navy Command, Control and Ocean Surveillance Center RDT&E Division, and by the Rome Laboratory.

[1] The definition of *trigger* is blurred as condition-action rules such as the production rule [BFKM85] have evolved to ECA-rules. A condition-action rule is triggered and eligible for execution when the current database state satisfies the specified condition, whereas for an ECA rule to be ready to execute, it has to pass separate event detection and condition test phases.

we use the word "trigger" to describe the eligible rules assuming that the condition part has been satisfied or it is nil.

The ECA rule execution model (rule execution models in general) has to address several issues. First, multiple rules can be triggered and eligible for execution at the same time. For example, suppose that two rules $r_i$ and $r_j$ are defined, respectively, on two events $E_1$ and $(E_1 \nabla E_2)$ and there are no conditions specified for the rules, where $(E_1 \nabla E_2)$ is a disjunction of two component events $E_1$ and $E_2$. A disjunction occurs when either component event occurs [CKAK94]. Now, if event $E_1$ occurs, it will trigger both $r_i$ and $r_j$. As addressed in [AWH92], multiple triggered rules pose problems when different execution orders can produce different final database states. If an active database system randomly chooses a rule to execute (out of several triggered rules), the final database state is *nondeterministic*. This adds to the problem of understanding applications that trigger rules. Priorities are generally used to deal with conflict resolution of rules [SHP88, C$^+$89, ACL91, Han92]. When multiple conflicting rules are triggered at the same time, a rule with the highest priority is selected for execution.

[AWH92] focuses on testing whether or not given a rule set has the *confluence* property. A rule set is said to be confluent if any permutation of the rules yields the same final database state when the rules are executed. If a rule set turns out to be not confluent, either the rule set is rewritten to remove the conflicts or priorities are explicitly defined over the conflicting rules. Then, the new rule set is retested to see if it has the confluence property. A problem with this approach is that it tends to repeat the time-consuming test process until the rule set eventually becomes confluent. Also, it has not been shown as to how confluence can be guaranteed as priorities between conflicting rules are added to the system.

There are other subtle problems with the ECA rule execution model. Suppose that $r_i$ and $r_j$ mentioned previously have condition parts. As event $E_1$ occurs, the two rules pass the event detection phase. Assume that both rules have passed the condition test and ready to execute their action part. It is possible that the execution of one rule's action, say $r_i$'s, can invalidate $r_j$'s condition that was already tested to be true; that is, $r_i$ can *untrigger* $r_j$. Apart from the issue of whether or not the condition test should be delayed up to the point (or retested at the point) just before execution of the action part, if one rule untriggers other rules, it is very likely that the rule set is not confluent. The opposite situation can also happen. Suppose the condition of $r_i$ was not met. So its action part would not be executed. But execution of $r_j$'s action could change database state so that $r_i$'s condition could be satisfied this time. Therefore, if $r_j$ executes first and $r_i$'s condition is tested after that, $r_i$ will be able to execute too. Again, execution order of the two rules makes a difference. Instead of proposing a more rigorous rule execution model to deal with the anomalies, we consider such rules as conflicting with one another so that the rule programmer can be informed of these rules. This view will allow us to cover the problem within the framework of confluent rule executions.

In this paper we explore problems of confluent rule executions, which deal with obtaining a unique final database state for any initial set of rules that is triggered by a user transaction. We show that previous rule execution models are inadequate for confluent rule executions (Section 2), and propose a new rule execution model that guarantees confluent executions (Sectione 3, 4, and part of 5). We also show that our model is a perfect fit for parallel rule executions (Section 5). Conclusions are in Section 6.


Figure 1: Rule execution graphs

## 2 Limitations of the Earlier Rule Execution Models

Early rule execution models such as one used in OPS5 [BFKM85] deal with problems of confluent rule executions only in terms of conflict resolution. When multiple rules are triggered (and eligible for execution), the rule scheduler selects a rule to execute according to a certain set of criteria such as recency of trigger time, complexity of conditions. Although this scheme is used in the AI domain, users in the database domain prefer deterministic states from the executions of transactions. Furthermore, the conflict resolution approach is not a complete answer to the confluence problem since it is based on dynamic components such as recency of trigger time and does not support concurrent execution. For the above reasons, we do not consider this approach in our work.

A somewhat different approach taken in active database systems such as Starburst [AWH92, ACL91] and Postgres [SHP88, SJGP90] is to statically assign execution priorities over rules. In these systems if multiple rules are triggered, a rule with the highest priority among them is executed first. However, rule execution models in these systems cannot guarantee confluent rule executions unless all the rules (not only conflicting ones) are totally ordered. This problem is illustrated in the following examples.

**Example 2.1** Figure 1(a) shows nondeterministic behavior of rule execution even when all conflicting rules are ordered. In the figure solid arrows represent trigger relationships. Dashed lines represent conflicts and an arrow on a dashed line indicates priority between two conflicting rules. As shown, two pairs of rules are conflicting: $(r_2, r_5)$ and $(r_3, r_5)$. The conflicting rules are ordered in such a way that $r_2$ precedes $r_5$ and $r_5$ precedes $r_3$ when the pairs of conflicting rules are triggered at the same time. Now suppose $r_1$ and $r_4$ are triggered by the user transaction at the same time. (Note that these rules are denoted by solid circles in the graph.) In a rule execution model such as Starburst, one of $r_1$ and $r_4$ will be randomly selected for execution since there is no conflict between them. Suppose $r_4$ is executed first, then it will trigger $r_5$. Yet there is no order between $r_1$ and $r_5$ which are ready to execute. So $r_5$ may go first, and its execution will trigger $r_6$. Then, $r_6$, $r_1$, $r_2$, and $r_3$ may follow. Including this, two of legitimate execution sequences for the rule set are following: (1) $\langle r_4 \cdot r_5 \cdot r_6 \cdot r_1 \cdot r_2 \cdot r_3 \rangle$ and (2) $\langle r_1 \cdot r_2 \cdot r_3 \cdot r_4 \cdot r_5 \cdot r_6 \rangle$. Note that relative orders of two conflicting rules, $r_2$ and $r_5$ (as well as $r_3$ and $r_5$) in the two rule execution sequences are different, thereby unable to guarantee confluent execution. ◁

**Example 2.2** Figure 2 illustrates another situation where the previous rule execution models fail to achieve confluent rule executions. There is a dependency between $r_k$ and $r_l$ and $r_k$ has priority over $r_l$. In this example, $r_i$ and $r_j$ are triggered by the user transaction. Note also that $r_i$ is an ancestor of $r_j$ in trigger relationship and thereby trigger paths originated from both rules overlap one another. Given that priority, the following two (and more) sequences of rule executions are

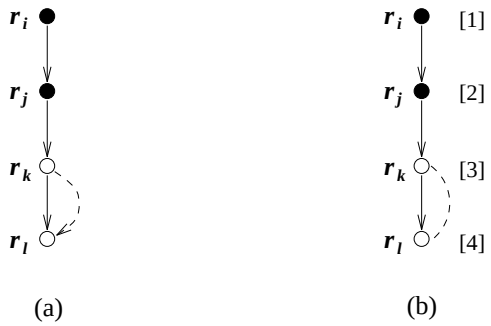


Figure 2: Overlapped trigger paths

possible: $\langle r_i \cdot r_j \cdot r_k \cdot r_l \cdot r_j \cdot r_k \cdot r_l \rangle$ and $\langle r_i \cdot r_j \cdot r_k \cdot r_j \cdot r_k \cdot r_l \cdot r_l \rangle$. Now relative orders of two conflicting rules $r_k$ and $r_l$ in the two execution sequences are different. Therefore confluent rule executions cannot be guaranteed in the given situation. ◁

As the previous examples suggest, a problem that the extant active rule execution models fail to address properly is that even though two rules are not directly conflicting each other, they may trigger other rules that are directly conflicting. Depending on the execution order of triggering rules, directly conflicting rules may be executed in a different order from what the user specified, likely resulting non-confluent rule executions. Unless all the direct conflicts are removed by rewriting the rules, one possible remedy for this problem, implied in Starburst [AWH92], would be to regard the indirectly conflicting rules as conflicting ones. Figure 1(b) illustrates how conflicts of Figure 1(a) are propagated toward ancestor rules in trigger relationship if this approach is taken. An undesirable consequence of propagating conflicts is that it severely limits parallel rule execution. In addition, it is not always clear how to propagate conflicts in some cases as Figure 2(a) shows.

Another problem which the previous rule execution models does not handle is shown in Example 2.2 where trigger paths of rules triggered by the user transaction overlap. In fact, this new situation poses additional problems for priority specification. That is, any static priority schemes specified before rules' execution cannot range over all possible permutations of conflicting rules execution, since one cannot anticipate which rules will be triggered by the user transaction how many times. For instance, given the rule set of Figure 2(a), there can be two distinct final database states which result from rule execution sequences, $\langle r_i \cdot r_j \cdot r_k \cdot r_l \cdot r_j \cdot r_k \cdot r_l \rangle$ and $\langle r_i \cdot r_j \cdot r_k \cdot r_j \cdot r_k \cdot r_l \cdot r_l \rangle$. All other legitimate rule execution sequences are equivalent to one of the two sequences in terms of final database states. However, if $r_j$ is triggered twice and $r_i$ once by the user transaction, the number of distinct final database states increases up to five. As $r_i$ and $r_j$ are triggered more number of time, the number of final database states increases exponentially. Therefore, it is not realistic to provide every possible alternative for these cases. Rather, a much less general scheme of priority specification, which provides only some specific alternatives, needs to be considered.

Figure 2(b) shows one way of specifying priority for the rules in Figure 2(a), which is similar to the priority scheme adopted in Postgres [SHP88]. Numbers in brackets denote absolute priorities associated with rules. A larger number denotes a higher priority. This priority specification guarantees confluent rule executions although non-conflicting rules ($r_i$ and $r_j$) too need to be assigned priorities. Note that the given priority specification is (unnecessarily) strong it effectively imposes a serial execution order $\langle r_j \cdot r_k \cdot r_l \cdot r_i \cdot r_j \cdot r_k \cdot r_l \rangle$, thereby ruling out any parallel rule executions. For instance, one instance of $r_j$ could run in parallel with a series of $r_i$ and $r_j$ without affecting the final database state.

In the rest of the paper, we develop a novel rule execution model and a priority scheme that not only ensures confluent rule executions but also allows greater parallelism.

# 3 Definitions and Extensions

## 3.1 Rule Execution Sequence (RES) and Rule Commutativity

Informally, a rule execution sequence (RES) is a sequence of rules that the system can execute when a user transaction triggers at least one rule in the sequence. To characterize RESs, we first define partial RESs. Throughout this paper, $R$ denotes *system rule set*, a set of rules defined in the system by the user. $D$ denotes a set of all possible database states determined by the database schema. $(d_j, R_k)$, $d_j \in D$ and $R_k \subseteq R$, denotes a pair of a database state and a triggered rule set. If $R_k$ is a set of rules directly triggered by a user transaction, it is termed *UTRS* which stands for User-Triggered-Rule-Set. UTRS is, in fact, a multiset since as we shall see later, multiple instances of a rule can be in it. $\mathcal{S}$ denotes the set of all partial RESs (see below) defined over $R$ and $D$.

**Partial RES** Given $R$ and $D$, for a nonempty set of triggered rules, $R_k \subseteq R$ and a database state $d_j \in D$, a *partial RES*, $\sigma$ is defined to be a sequence of rules that connects pairs of a database state and a triggered rule set as follows:

$$\sigma = \langle (d_j, R_k) \stackrel{r_i}{\rightarrow} (d_{j+1}, R_{k+1}) \stackrel{r_{i+1}}{\rightarrow} \cdots \stackrel{r_{i+m-1}}{\rightarrow} (d_{j+m}, R_{k+m}) \rangle$$

where $d_{j+l} \in D$ $(1 \leq l \leq m)$ is a new database state obtained by the execution of $r_{i+l-1}$, each rule $r_{i+l}$ $(0 \leq l < m)$ is in a triggered rule set $R_{k+l}$, and eligible for execution in $d_{j+l}$, i.e., $d_{j+l}$ evaluates the rule's condition test to true. Each triggered rule set $R_{k+l} \subseteq R$ $(1 \leq l \leq m)$ is built as $R_{k+l} = ((R_{k+l-1} - \{r_{i+l-1}\}) - Ru_{k+l}) \cup Rt_{k+l}$, where $Ru_{k+l}$ is a set of rules untriggered by $r_{i+l-1}$ and $Rt_{k+l}$ is a set of rules triggered by $r_{i+l-1}$.[2] ◁

In this paper, if only sequences of rule executions are of interest, for simplicity we write a partial RES without associated database states and triggered rule sets. For example, the partial RES above can be denoted as $\sigma = \langle r_i \cdot r_{i+1} \cdots r_{i+m-1} \rangle$, which we already used for partial RESs in the previous sections. Among partial RESs, we are interested in some, called complete RESs (or simply, RESs), that satisfy certain conditions.

**Complete RES** Given $R$ and $D$, for a nonempty set $R_k \subseteq R$ which is a set of rules triggered by a user transaction (i.e., UTRS) and $d_j \in D$ is a database state produced by operations in the user transaction, a *complete RES* (or *RES*), $\sigma$ is defined to be a partial RES:

$$\sigma = \langle (d_j, R_k) \stackrel{r_i}{\rightarrow} (d_{j+1}, R_{k+1}) \stackrel{r_{i+1}}{\rightarrow} \cdots \stackrel{r_{i+m-1}}{\rightarrow} (d_{j+m}, R_{k+m} = \emptyset) \rangle$$

where no triggered (and eligible) rules remain after execution of the last rule $r_{i+m-1}$ (i.e., $R_{k+m} = \emptyset$). ◁

Note that given $R_k$ and $d_j$, there may be multiple different RESs, even in a case where there is only one rule in $R_k$, and those RESs do not necessarily have the same set of rules executed, since a rule's triggering/untriggering other rules may be dependent on the current database state. In this paper we use *rule schedule* in informal settings interchangeably with complete RES.

[2] Subscripts, $i, i+1, \cdots, i+m-1$, attached to rules, are intended to mean that they are $m$ rules that need not be distinct (similarly for $d$'s and $R$'s). They do not represent any sequential order of rules with respect to subscript numbers. That is, they should not be interpreted as, for instances, $r_{10}, r_{11}, r_{12} \cdots$, in case where $r_i$ is $r_{10}$. For a precise denotation, we could use $i_0, i_1, \cdots, i_{m-1}$, instead. However, we have opted for the less precise notation in favor of simplicity throughout this paper.

**Rule shuffling** Given a partial RES $\sigma_1$, two rules $r_i$ and $r_j$ in $\sigma_1$ can exchange their positions provided $r_j \in R_y$, yielding a different partial RES $\sigma_2$ as below:

$$\sigma_1 = \langle (d_x, R_y) \stackrel{r_i}{\rightarrow} (d_k, R_l) \stackrel{r_j}{\rightarrow} (d_u, R_v) \rangle$$

$$\sigma_2 = \langle (d_x, R_y) \stackrel{r_j}{\rightarrow} (d_m, R_n) \stackrel{r_i}{\rightarrow} (d_s, R_t) \rangle$$

◁

Next we define an important property of rules that is used to show if a system rule set is confluent. Two rules are defined to be commutative if shuffling them always yields the same result.

**Rule commutativity** Given $R$ and $D$, two rules $r_i, r_j \in R$ are defined to be *commutative*, if for all $R_y \subseteq R$, where $r_i, r_j \in R_y$, and for all database state $d_x \in D$, the following two partial RESs can be defined:

$$\langle (d_x, R_y) \stackrel{r_i}{\rightarrow} (d_k, R_l) \stackrel{r_j}{\rightarrow} (d_u, R_v) \rangle$$

$$\langle (d_x, R_y) \stackrel{r_j}{\rightarrow} (d_m, R_n) \stackrel{r_i}{\rightarrow} (d_u, R_v) \rangle$$

where $d_x, d_k, d_m, d_u \in D$ need not be distinct and likewise $R_y, R_l, R_n, R_v \subseteq R$ need not be distinct. ◁

Note that any rule is trivially commutative to itself.

**Equivalent partial RESs** Two partial RESs $\sigma_i$ and $\sigma_j$ are defined to be *equivalent* ($\equiv$) if:

1. $\sigma_i$ and $\sigma_j$ begin with the same pair of database state and triggered rule set, and end with the same pair of database state and triggered rule set; and
2. in $\sigma_i$ and $\sigma_j$ the same set of rules is triggered, possibly in different orders. ◁

The two partial RESs shown in the definition of rule commutativity are equivalent. In fact, the rule commutativity is used to prove that two or more partial RESs are equivalent, and the equivalence of partial RESs is, in turn, used to show whether given a system rule set is confluent or not. Incidentally, it should be noted that without condition 2), the equivalence definition can still be valid. However, with our static analysis method it is not possible to identify all such equivalent partial RESs. To make presentation of this paper coherent, we chose a more restrictive form of equivalence.

The equivalence of partial RESs naturally lends itself to definition of equivalence classes of partial RESs. For given $R$ and $D$, the set of all partial RESs, $\mathcal{S}$ is partitioned into disjoint classes by the equivalence relation ($\equiv$). All partial RESs belonging to an equivalence class have the same final result, i.e., the same database state and the triggered rule set.

**Equivalence class of partial RESs** For a partial RES, $\sigma \in \mathcal{S}$, the *equivalence class* of $\sigma$ is the set $S_\sigma$ defined as follows:

$$S_\sigma = \{\gamma \in \mathcal{S} \mid \gamma \equiv \sigma\}.$$

◁

Of partial RESs belonging to the same equivalence class, for the discussion in this paper we define *canonical partial RES*, or *canonical RES* for short, to be a partial RES that comes first when all the partial RESs are sorted by their rules' indices in lexicographical order. For instance, assuming that an equivalence class includes only three partial RESs, $\sigma_i = \langle r_1 \cdot r_2 \cdot r_4 \cdot r_3 \rangle$, $\sigma_j = \langle r_1 \cdot r_4 \cdot r_2 \cdot r_3 \rangle$, and

$\sigma_k = \langle r_1 \cdot r_2 \cdot r_3 \cdot r_4 \rangle$, $\sigma_k$ is the canonical RES representing that equivalence class (by lexicographically sorting the sets:, $(1,2,4,3)$, $(1,2,3,4)$ and $(1,4,2,3)$). Our prime interest is the equivalence class of complete RESs.

**Confluent Rule Set** Given $R$ and $D$, if there exists only one equivalence class of complete RESs for every nonempty set $R' \subseteq R$ and every $d \in D$, $R$ is defined to be *confluent*. ◁

### 3.2 Dependencies and Dependency Graph

If a different execution sequence of the same rules can produce a different final database state, it is because of certain interactions between rules and between rules and the environment. If we assume the execution environment to be fixed and there is no interference from the user while rules are executing, the interactions between rules must be the sole reason for non-confluent rule executions. Based on this, we define rules' interactions responsible for non-confluence as *dependencies* between rules, much like those of the concurrency control in transaction processing. We define two kind of dependencies.

**Data dependency** Two distinct rules $r_i$ and $r_j$ are defined to have *data dependency* with each other if $r_i$ writes in its action part to a data object that $r_j$ reads or writes in its action part, or vice versa. ◁

**Untrigger dependency** Two distinct rules $r_i$ and $r_j$ are defined to have *untrigger dependency* each other if $r_i$ writes in its action part to a data object that $r_j$ reads in its condition part or vice versa. ◁

If two rules have a data dependency with each other, the input to one rule can be altered by the other rule's action. Thus it is very likely that the affected rule would behave differently. The data dependency can also mean that one rule's output can be overridden by the other rule's output. This also has a bearing on the final outcome. If there is no data dependency, two rules act independently. Therefore, there should be no difference in the final outcome due to different relative execution order of the two rules.

On the other hand, if there is untrigger dependency between two rules $r_i$ and $r_j$, this implies that one rule's action can change the condition which determines whether the other rule is to execute or not. If the affected rule, say $r_i$, has already executed first, it is unrealistic to revoke the effect of $r_i$. As a result, both $r_i$ and $r_j$ will execute in this case. However, if the affecting rule $r_j$ executes first, it can prevent $r_i$ from executing. Since it is assumed that there are no read-only rules, the two different execution sequences can result in different database states even though there is no data dependency.[3]

From the above observation, it is clear that the absence of data dependency and untrigger dependency between two rules is a sufficient condition for the two rules to be commutative. (The reverse is not necessarily true.) If there exists either dependency between two rules, the rules are said to *conflict* with each other. Obviously, conflicting rules are non-commutative.

[3]It should be noted that whether or not the untrigger dependency can indeed affect confluent execution depends on rule execution model employed by an active database system. If the rule execution model does not re-check the condition part of a rule just before it executes the action part of that rule, then no rule is untriggered. In such a case, it can appear that the untrigger dependency is no longer a problem and only data dependency matters.

**Lemma 1** *Given a partial RES $\sigma$, a new partial RES $\sigma'$ obtained by freely shuffling rules in $\sigma$ is equivalent to $\sigma$, as long as relative orders of conflicting rules in both RESs are equal if there are any conflicting rules.*

**Proof of lemma 1** Suppose $\sigma$ and $\sigma'$ are not equivalent despite the same relative orders of conflicting rules in them. Then, there must be one or more pairs of non-conflicting rules in $\sigma$ that can be shuffled but result in a different (non-equivalent) partial RES. These non-conflicting rules are, then, conflicting and should have the same relative orders in $\sigma$ and $\sigma'$, which is a contradiction. □

Below, we define dependency graph that represents dependencies between rules in the system rule set.

**Dependency graph** Given a system rule set $R$, a *dependency graph*, $DG = (R, E_D)$ is an undirected graph where $R$ is a node set in which a node has one-to-one mapping to a rule and $E_D$ is a dependency edge set. For any two nodes $u$ and $v$ in $R$, a *dependency edge*, $(u, v)$ is in $E_D$ if and only if there is data dependency or untrigger dependency (or both) between $u$ and $v$. ◁

A dependency graph is non-transitive; that is, $(u, v)$ and $(v, w)$ in $E_D$ do not imply $(u, w)$ in $E_D$. Edges in a dependency graph represent only direct dependencies. An indirect (transitive) dependency is represented by a path consisting of a set of connected dependency edges.

### 3.3 Trigger Graph

A *trigger graph* ($TG$) is an *acyclic* directed graph representing trigger relationships between rules within given a system rule set $R$. For a system rule set $R$, $TG = (R, E_T)$ has a node set $R$ and a trigger edge set $E_T$. The node set is equivalent to the rule set. For any two rules (i.e., nodes) $r_i$ and $r_j$ in $R$, trigger edge set $E_T$ contains a directed edge, called *trigger edge*, $(r_i \xrightarrow{T} r_j)$, if and only if $r_i$ *can trigger* $r_j$. It is defined that $r_i$ can trigger $r_j$ if execution of $r_i$'s action can give rise to *an* event that is referenced in the event specification of $r_j$.[4]

Note that for rules $r_i$ and $r_j$ above, it is possible that $r_j$ is not triggered by $r_i$ at run time if $r_i$'s action part contains a conditional statement. Nevertheless we conservatively maintain a trigger edge if there is any possibility of $r_i$'s triggering $r_j$. In addition, we are assuming that a trigger graph is *acyclic* to guarantee termination of rule executions [AWH92]. If a trigger graph contains a cycle, it is possible that once a rule in the cycle is triggered all the rules in the cycle keep triggering the next rule indefinitely. We also assume that there exists a separate mechanism for detecting cycles in a trigger graph so that the rule programmer can rewrite the rules in such a case.

A *trigger path* in a trigger graph is a path starting from any node and ending at a reachable leaf node. Incidentally, it should be noted that a trigger relationship between two rules does not necessarily imply a dependency between the rules. For instance, given a trigger edge $(r_i \xrightarrow{T} r_j)$, if $r_i$ for sure triggers $r_j$ and no other rules are triggered from $r_i$ and $r_j$, there are only two possible partial RESs for the two rules, $\langle r_i \cdot r_j \cdot r_j \rangle$ and $\langle r_j \cdot r_i \cdot r_j \rangle$. If there is no data or untrigger dependency between $r_i$ and $r_j$ (i.e, the two rules are commutative), the two RESs are equivalent despite the trigger edge.

[4] We admit that this definition of "can trigger" is rather crude. In Sentinel, for example, if a rule is waiting for an occurrence of $(E_1; E_2)$, which is a composite event *sequence* and occurs when $E_2$ occurs provided $E_1$ occurred before, the occurrence of $E_1$ alone never triggers that rule. In our current work, however, we do not pursue this issue any further. (For event specifications in Sentinel, see [CKAK94].)

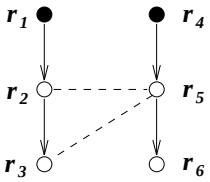


Figure 3: A conflicting rule set

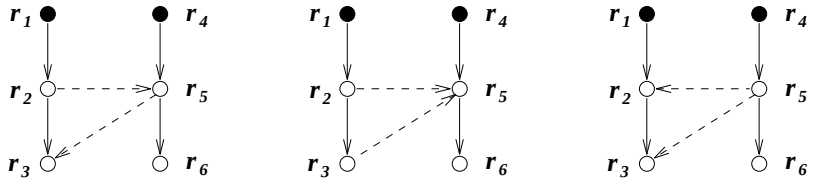


Figure 4: Priority graphs for Figure 3

# 4 Confluence and Priority Specification

In this section we present basic ideas that give us a handle for dealing with conflicting rules in order to obtain confluent rules executions. We consider simple cases first. When there are $n$ distinct rules to execute and $m$ pairs of conflicting rules among them, intuitively, the maximum number of different final database states that can result from all differing RESs is conservatively bounded by $2^m$, since each pair of conflicting rules can possibly produce two different final database states by changing their relative order.

**Example 4.1** Figure 3 is a redrawing of Figure 1(a) with removal of directions on dependency edges $(r_2, r_5)$ and $(r_3, r_5)$. Note that $r_1$ and $r_4$, denoted by solid nodes in the graph, are in UTRS, a set of rules initially triggered by a user transaction. Assuming that all the six rules are executed, all complete RESs that can be generated should be equal to a set of possible merged sequences of two partial RESs $\langle r_1 \cdot r_2 \cdot r_3 \rangle$ and $\langle r_4 \cdot r_5 \cdot r_6 \rangle$. Then, all the possible merged (now complete) RESs can be partitioned into up to four groups by relative orders between $r_2$ and $r_5$ and between $r_3$ and $r_5$ as follows: (1) $(r_2 \rightarrow r_5)$ $(r_3 \leftarrow r_5)$, (2) $(r_2 \rightarrow r_5)$ $(r_3 \rightarrow r_5)$, (3) $(r_2 \leftarrow r_5)$ $(r_3 \leftarrow r_5)$, and (4) $(r_2 \leftarrow r_5)$ $(r_3 \rightarrow r_5)$. However, since there exists an inherent order between $r_2$ and $r_3$, i.e., $(r_2 \rightarrow r_3)$, dictated by a trigger relationship, no merged RESs can contain combination (4) due to a cycle being formed. Combination (4) is dropped from consideration. Since cumulative effect of all the other rules are the same regardless of their execution order, the three combinations are the only factors that can make a difference in the final database state. Therefore, in this example, up to three distinct final database states can be produced by all possible complete RESs. ◁

Using the three possible orderings of conflicting rules in Example 4.1, we can assign directions to dependency edges in the graph of Figure 3. Resulting graphs, which we call *priority graphs*, are shown in Figure 4. These priority graphs present how priorities can be specified over conflicting rules in order to make rule executions confluent. Also, importantly, they represent partial orders that the rule scheduler needs to follow as it schedules rule executions. As we shall see in the following section, the rule scheduler basically uses a topological sort algorithm working on a subgraph of priority graph, and this demands the priority graph to be *acyclic*.

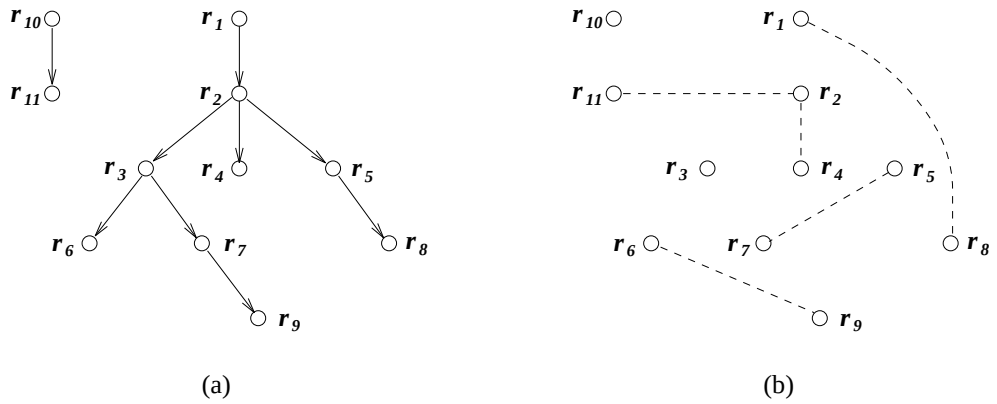


Figure 5: A pair of trigger graph and dependency graph

**Example 4.2** All possible topological sorts on priority graph of Figure 4(a) correspond to an equivalence class represented by a canonical RES, $\sigma_1 = \langle r_1 \cdot \bar{r_2} \cdot r_4 \cdot \bar{r_5} \cdot \bar{r_3} \cdot r_6 \rangle$ – for clarity we use $\bar{}$ to denote conflicting rules as $\bar{r_2}$. Note that a RES, $\langle r_1 \cdot r_4 \cdot \bar{r_2} \cdot \bar{r_5} \cdot \bar{r_3} \cdot r_6 \rangle$ is equivalently converted to $\sigma_1$ by shuffling $r_2$ and $r_4$, which are commutative. Similarly, $\sigma_2 = \langle r_1 \cdot \bar{r_2} \cdot \bar{r_3} \cdot r_4 \cdot \bar{r_5} \cdot r_6 \rangle$ and $\sigma_3 = \langle r_1 \cdot r_4 \cdot \bar{r_5} \cdot \bar{r_2} \cdot \bar{r_3} \cdot r_6 \rangle$ represent equivalence classes obtained when the topological sort is carried out on priority graphs of Figure 4(b) and Figure 4((c), respectively. ◁

The formal definition of the priority graph is given below.

**Priority graph** Given trigger graph $TG = (R, E_T)$ and dependency graph $DG = (R, E_D)$, *priority graph*, $PG = (R, E_P)$ is a directed *acyclic* multigraph formed by merging $TG$ and $DG$, where $R$ is a node set defined as before and $E_P$ is a priority edge set. For any two distinct nodes $u, v \in R$, $(u \xrightarrow{T} v) \in E_P$ if and only if $(u \xrightarrow{T} v) \in E_T$, and either $(u \xrightarrow{D} v) \in E_P$ or $(v \xrightarrow{D} u) \in E_P$ if and only if $(u, v) \in E_D$. $(u \xrightarrow{D} v)$ (or $(v \xrightarrow{D} u)$) is called *directed dependency edge* to distinguish it from undirected ones in $E_D$, and the direction of the edge is given by the user. ◁

For a $PG$, a trigger edge is depicted by a solid arrow line while a directed dependency edge is depicted by a dashed arrow line. A $PG$ is defined to be a multigraph because it can have more than one edge (actually two edges, i.e., a trigger edge and a directed dependency edge) between two nodes. It should be noted that if a node $u$ is an ancestor of a node $v$ in $TG$ and there is a dependency edge $(u, v)$ in $DG$, the corresponding directed dependency edge in $PG$ is automatically set to $(u \xrightarrow{D} v)$, not to form a cycle in $PG$.

**Example 4.3** Figures 5(a) and (b) show a trigger graph and its dependency graph counterpart respectively. Figure 6 shows a priority graph built out of the two graphs. Note that directions of dependency edges are determined by the user as a way of specifying priorities between conflicting rules. However, direction of dependency edge between $r_1$ and $r_8$ (between $r_2$ and $r_4$ as well) is set by the system as shown in the graph because $r_1$ is an ancestor of $r_8$ in a trigger path. ◁

Note that given a system rule set $R$, only one $PG$ is built at compile time. The $PG$ contains nodes for all rules in $R$. When an active database system is running, subsets of $R$ will be triggered and executed dynamically. In order to schedule those rules, the rule scheduler builds a subgraph of $PG$, called *execution graph*, when a user transaction triggers rules.

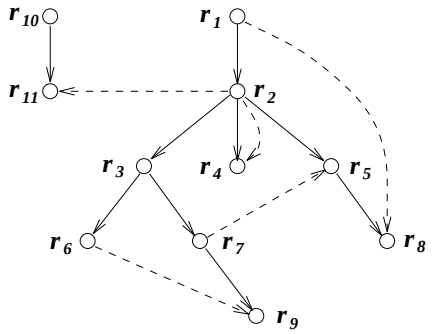


Figure 6: A priority graph

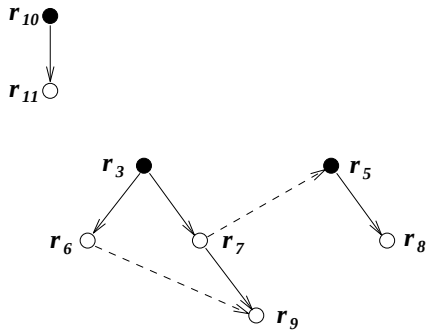


Figure 7: An execution graph

**Execution graph** Given a system rule set $R$, a priority graph $PG = (R, E_P)$, and a UTRS $R' \subseteq R$, an *execution graph* $EG = (R_E, E_E)$ is a subgraph of $PG$ where $R_E$ is a node set and $E_E$ is an edge set. $R_E$ is recursively defined as $R_E = \{r \mid r \in R' \vee (\exists r' \exists (r' \xrightarrow{T} r)\,(r' \in R_E \wedge (r' \xrightarrow{T} r) \in E_P)\}$. For any two distinct nodes $u, v \in R_E$, $(u \xrightarrow{T} v) \in E_E$ if $(u \xrightarrow{T} v) \in E_P$ and $(u \xrightarrow{D} v) \in E_E$ if $(u \xrightarrow{D} v) \in E_P$. ◁

Simply stated, the node set $R_E$ consists of a UTRS plus those rules that are reachable from rules in the UTRS through trigger paths in $PG$. The edge set $E_E$ is a set of trigger and directed dependency edges that connect nodes in $R_E$.

**Example 4.4** Figure 7 shows an execution graph derived from a priority graph of Figure 6 when a UTRS has rules $r_3$, $r_5$, and $r_{10}$. A rule schedule can be obtained by performing the topological sort on the execution graph. The canonical RES for the equivalence class represented by the execution graph is $\langle r_3 \cdot r_6 \cdot r_7 \cdot r_5 \cdot r_8 \cdot r_9 \cdot r_{10} \cdot r_{11} \rangle$. Note that priority graphs shown earlier in Figure 4 are, in fact, execution graphs as well. ◁

**Lemma 2** *Given an execution graph $EG = (R_E, E_E)$, a set of rule execution sequences corresponding to all feasible topological sorts on $EG$ constitutes an equivalence class,* independent *of the initial database state.*

**Proof of lemma 2** Since $EG$ is acyclic and all pairs of conflicting rules in $EG$ are ordered (i.e., have an edge between them), all topological sorts on $EG$ should have the same relative orders of

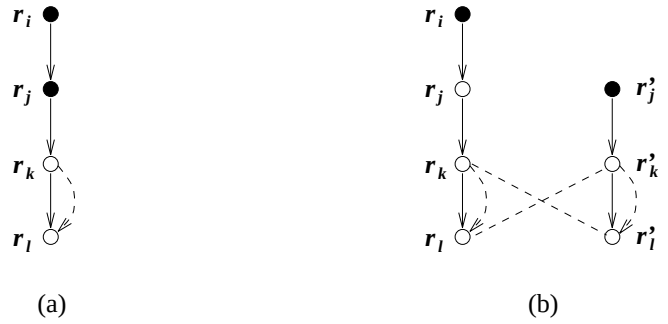


Figure 8: Overlapping trigger paths and extended execution graph

conflicting rules. Then, by Lemma 1, RESs represented by the topological sorts are equivalent each other. Also, since Lemma 1 holds without any premise on initial database states, Lemma 2 also holds regardless of initial database states. □

# 5 Strict Order-Preserving Rule Execution Model

Thus far, only simple cases were taken into account. Specifically, trigger graphs are of a tree structure, all rules in a UTRS are distinct, and no parent/child relationship exists among the rules. As a result, no trigger paths in the execution graph overlapped with one another. When rules in a UTRS have overlapping trigger paths, the priority graph and execution graph defined in the previous section does not capture the semantics. For example, consider Figure 8(a) which is the same as Figure 2(a). When $r_i$ and $r_j$ are triggered by a transaction, both rules instantiate their own trigger paths, and these trigger paths overlap with each other [5]he overlapping of trigger paths is not directly visible in Figure 8(a). Treating the graph as an execution graph yields two partial RESs : $\langle r_i \cdot r_j \cdot \bar{r_k} \cdot \bar{r_l} \rangle$ and $\langle r_j \cdot \bar{r_k} \cdot \bar{r_l} \rangle$. Therefore, the rule schedule or alternately the complete RES should be a merged RES of the two partial RES's. A possible merge RES is $\langle r_i \cdot r_j \cdot r_j \cdot \bar{r_k} \cdot \bar{r_l} \cdot \bar{r_k} \cdot \bar{r_l} \rangle$. In order to accomplish, we need to (i) obtain a complete RES using topological sort and (ii) determine the equivalence class of an execution. These issues are addressed in this section.

## 5.1 Extended Execution Graph

In order to understand the effect of overlapping trigger paths, we introduce an extended execution graph, used only for illustration purposes. Recall that given a rule set along with its trigger graph, each rule in a UTRS instantiates a subgraph of the trigger graph, whose sole root is that rule. However it is possible to derive an execution graph, from a given priority graph and a UTRS, such that every rule in the UTRS becomes *a* root node in the resulting extended execution graph.

**Example 5.1** Figure 8(b) shows the extended execution graph of Figure 8(a). In the extended execution graph, $r_j$ and $r'_j$ (similarly other rules as well) are the same rule and only represent different instantiations. Since a dependency exists between rules $r_k$ and $r_l$, it may be present between all instantiations of $r_k$ and $r_l$ as shown in Figure 8(b). Directions of dependency edges in an extended execution graph may be either inferred from the priority graph or specified by the user. Figure 9 shows three different acyclic orderings of relevant dependency edges of Figure 8(b).

[5]T

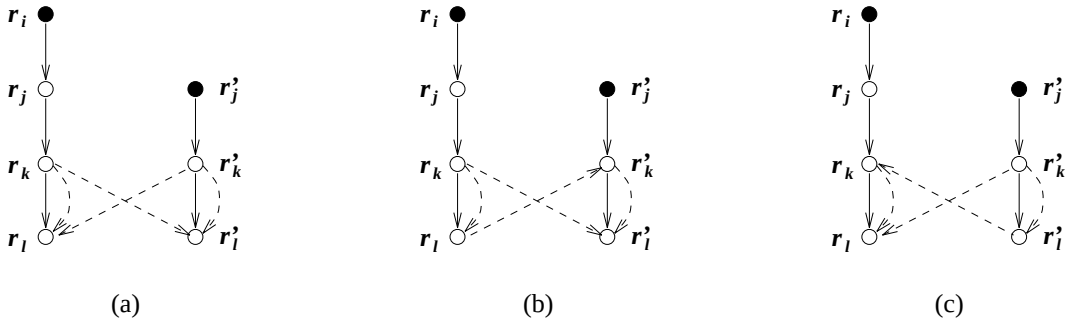


Figure 9: Three different orderings of dependency edges in Figure 8(b)

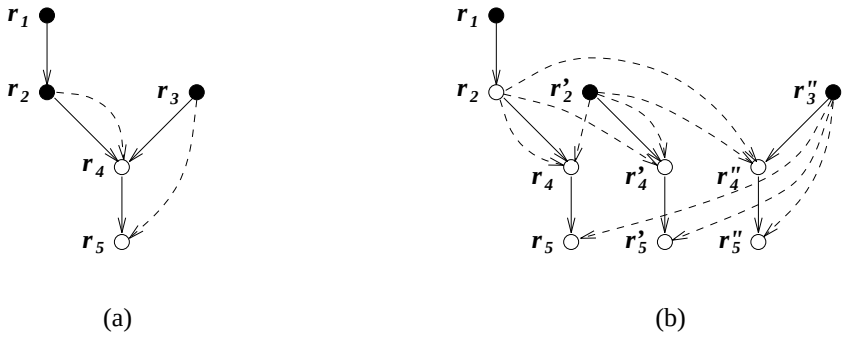


Figure 10: Extended execution graph in strict order-preserving executions

Once an acyclic extended execution graph is given, the rule scheduler can schedule rule executions using topological sort. All possible topological sorts constitute one equivalence class. ◁

The extended execution graph, however, *cannot* be used for priority specification; it is created only at rule execution time and priorities need to be specified before that. Thus we need to find alternative ways to interpret a priority graph.

## 5.2 Strict Order-Preserving Executions

One way to derive an extended execution graph is to faithfully follow what the user specifies in the priority graph, i.e., priorities between conflicting rules. In *strict order-preserving executions*, if rule $r_i$ has precedence over rule $r_j$ in a priority graph, all instances of $r_i$ precede all instances of $r_j$ in resulting rule schedules. Given a priority graph, an extended execution graph is obtained by simply adding directed dependency edges in the priority graph to duplicated overlapping trigger paths. This scheme provides a simple solution for overlapping trigger paths, regardless of the number of times trigger paths overlap.

**Example 5.2** Figure 10(a) shows a priority graph where rules $r_1$, $r_2$, and $r_3$ are in UTRS and their overlapping trigger paths are denoted by partial RESs, $\langle r_1 \cdot \bar{r_2} \cdot \bar{r_4} \cdot \bar{r_5} \rangle$, $\langle \bar{r_2} \cdot \bar{r_4} \cdot \bar{r_5} \rangle$, and $\langle \bar{r_3} \cdot \bar{r_4} \cdot \bar{r_5} \rangle$. Figure 10(b) illustrates how an extended execution graph is built using strict order-preserving executions. First, overlapping trigger paths are separated. Second, any dependency edges connecting a rule in an overlapping trigger path with a rule in any trigger path is also introduced in the extended execution graph. For example, $(r_2 \stackrel{D}{\rightarrow} r'_4)$ and $(r''_3 \stackrel{D}{\rightarrow} r'_5)$. Given the

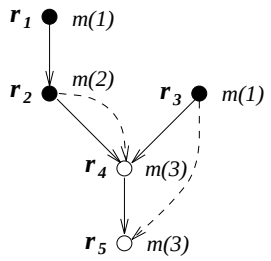


Figure 11: Extended execution graph with rule_counts

extended execution graph, the rule scheduler can schedule rule execution by performing a topological sort. All feasible topological sorts should constitute one equivalence class since in the topological sorts, executions of all conflicting rules are ordered in the same ways. The canonical RES for the extended execution graph of Figure 10(b) is $\langle r_1 \cdot \bar{r_2} \cdot \bar{r_2} \cdot \bar{r_3} \cdot \bar{r_4} \cdot \bar{r_4} \cdot \bar{r_4} \cdot \bar{r_5} \cdot \bar{r_5} \cdot \bar{r_5} \rangle$. (Note that $r_i$, $r_i'$, and $r_i''$ are the same rule.) $\lhd$

### 5.3 Implementation

In order to implement a rule scheduler conforming to strict order-preserving execution, one could directly use extended execution graphs shown in Figure 10(b). However, a simpler way for deriving an execution graph without duplicating every overlapping trigger path exists. Consider the extended execution graph of Figure 10(b). In the strict order-preserving execution, directions of all dependency edges incoming to and outgoing from overlapping trigger paths are all the same, as shown in the graph. Therefore, it is unnecessary to duplicate overlapping trigger paths. We, instead, add a rule_count to each node of a plain execution graph. A rule_count attached to a node indicates how many rules in UTRS share the trigger path which the node (i.e., rule) belongs to. Figure 11 depicts how the plain execution graph of Figure 10(a) is extended using rule_counts. In the new graph, $m(i)$ denotes the sharing of the trigger by $i$ instances of the associated rule.

Now the new extended execution graph can be used with a minor modification to perform the topological sort. Whenever a rule is scheduled, its rule_count in the execution graph is decreased by 1. If the rule_count reaches 0, the node and outgoing edges are removed from the execution graph.

Figure 12 describes an algorithm, *Build_EG()* for building an execution graph for given a priority graph ($PG$) and a UTRS. It uses $M[\,]$, an array of size of the system rule set $R$, to hold rule_count values of rules in $EG$. If rule $r_i$ is (to be) in $EG$, $M[i]$ represents the rule_count attached to $r_i$. *Build_EG()* calls a subroutine, *DFS_Copy_Tree()* for every instance in UTRS. Remember that UTRS is a multiset. It can have multiple instances of the same rule due to multiple triggers. *DFS_Copy_Tree()* traverses the $PG$ in the depth-first search fashion and copies a portion of the $PG$ that are reachable through trigger edges in $PG$. It also increases rule_count of each node that it visits. The execution graph of Figure 11 is obtained by applying *Build_EG()* to the priority graph of Figure 10(a) with $UTRS = \{r_1, r_2, r_3\}$.

Once an execution graph is built by *Build_EG()*, the rule scheduler can schedule rule executions. It is possible that some rules in a trigger path in an execution graph not be triggered at all by its parent rule at run time. In such a case, other rules in other trigger paths, having incoming dependency edges from that rule, should not wait. Furthermore, if a rule is not triggered, all its descendant rules in the trigger path will not be triggered either. This should be applied recursively down trigger paths.

```
Given PG and UTRS:

Build_EG(){
EG = ∅;
initialize array M[ ] to 0's; // M[ ] is rule_count array
for every r_i ∈ UTRS do
    call DFS_Copy_Tree(r_i);
}

DFS_Copy_Tree(r_i){
if (r_i ∉ EG) then
    copy r_i into EG;
M[i]++; // increase rule_count of node r_i
// copy trigger edges
for all r_j such that ∃(r_i --T--> r_j) ∈ PG do{
    call DFS_Copy_Tree(r_j);
    if ((r_i --T--> r_j) ∉ EG) then
        copy (r_i --T--> r_j) into EG;
    }
// copy dependency edges
for all r_j such that ∃(r_i --D--> r_j) ∈ PG do{
    if (r_j ∈ EG) and ((r_i --D--> r_j) ∉ EG) then
        copy (r_i --D--> r_j) into EG;
    }
}
```

Figure 12: Algorithm – Build_EG()

Figure 13 describes the rule scheduling algorithm, *Schedule()*, which is a modified version of topological sort. Given an execution graph $EG$, it arbitrarily selects a node (i.e., a rule) with indegree 0. Since $EG$ is acyclic, there should always be at least one node with indegree 0. After executing the selected rule, the scheduler decreases the node's rule_count by 1. If a node's rule_count reaches 0, the node is removed along with any trigger and dependency edges outgoing from the node. However, before the removal, it checks whether the executed rule has triggered child rules in trigger paths. If there are child rules that are not triggered, then *Schedule()* calls a subroutine *DFS_Dec_M()* for those child rules. *DFS_Dec_M()* traverses down $EG$ in a depth-first fashion and decreases rule_count of each node it visits by 1. If a node's rule_count becomes 0, it removes the node and all outgoing edges.

**Theorem 1** *The strict order-preserving rule execution model guarantees confluent rule executions.*

**Proof of theorem 1** Based on Lemma 2, algorithms *Build_EG()* and *Schedule()* together serve as a constructive proof for the theorem since by the algorithms, overlapping trigger paths are separated, effectively making them ordinary acyclic graphs, and the topological sort is performed on the graphs. □

```
Given EG:

Schedule(){
while (EG ≠ ∅) do{
    choose a node r_i with indegree 0;
    execute r_i;
    M[i]--; // decrease rule_count of node r_i
    for all r_j such that ∃(r_i --T--> r_j) ∈ EG do
        if (r_j was not triggered by execution of r_i) then
            call DFS_Dec_M(r_j);
    if ( M[i] = 0 ) then
        delete node r_i and edges (r_i --T--> r_k) and (r_i --D--> r_l), for any k and l, from EG;
    }
}

DFS_Dec_M(r_j){
M[j]--;
for all r_k such that ∃(r_j --T--> r_k) do
    call DFS_Dec_M(r_k);
if ( M[j] = 0 ) then
    delete node r_j and edges (r_j --T--> r_l) and (r_j --D--> r_m), for any l and m, from EG;
// don't need to delete dependency edges incoming to r_j!
}
```

Figure 13: Algorithm – Schedule()

### 5.4 Parallel Rule Execution

The execution graph naturally allows parallel execution of rules. In the extended form, such as Figure 10(b), all rules with indegree 0 can be launched in parallel for execution. Since there should be no dependency edges between nodes with indegree 0 in an execution graph,[6] relative execution order of those independent rules does not affect the final outcome. Note also that multiple instances of the same rule can be scheduled for execution at the same time.

In an execution graph with rule_counts, which we use for our work, all rules with indegree 0 are scheduled simultaneously as many times as the rule_count associated with the rules. In Figure 11, for instance, after scheduling and executing each one instance of $r_1$ and $r_3$ in parallel, two instances of $r_2$ can be scheduled for execution since rule_count of $r_2$ is 2. In order to implement the parallel rule executions, we have to make some changes to the algorithm of Figure 13 which we will not elaborate in this paper. Whenever execution of one instance of a rule is completed, the associated rule_count need be decreased by 1 and its child rules have to be checked to see whether they are triggered or not by the parent rule. If some are not triggered, *DFS_Dec_M()* should be called to recursively decrease rule_counts along trigger paths. Since the rule_count array *M[]* and execution

[6] Note that an execution graph reduces is size as rules are executed.

graph are shared data structures, some locking mechanism need be used to avoid update anomalies within the data structures.

One important measure in parallel processing is the degree of parallelism. In the active rule system, the maximum parallelism is bounded by dependencies between rules in the system rule set. For instance, if all the rules are independent of each other, ideally all triggered rules can be executed in parallel. As dependencies between rules increase, the degree of parallelism would decrease. However, other components too can restrict parallelism. As discussed in Section 2, improper priority specification and rule execution model may execute a given rule set serially which could be executed in parallel. Specifically in our work, two components can hamper parallelism, all resulted from static analysis. First, precision of dependency analysis between two rules can affect parallelism. Even though there is data dependency between two rules, they can be commutative in reality. Being unable to detect such a hidden commutativity results in a false dependency edge in an execution graph, likely costing parallelism. Second, precision of trigger relationship analysis can similarly affect parallelism. If we know for sure that one rule triggers another rule, the trigger edge between the two rules can be deleted after all rule_count values are computed. This way, the two rules can be scheduled in parallel if there is no other path connecting them in the resultant execution graph. Using static analysis, we cannot completely avoid uncertain trigger edges, and presence of the uncertain trigger edges can cost parallelism. However, ignoring the loss caused by imprecision of static analysis, the strict order-preserving rule executions exploit the maximum parallelism existing in a given system rule set. We state it in Theorem 2.

**Theorem 2** *Using the strict order-preserving rule executions, the active rule execution model achieve the maximum parallelism within limitations of static trigger and dependency analysis.*

**Proof of theorem 2** Given any acyclic extended execution graph, there are two kinds of edges; trigger edges and dependency edges. We first assume that no trigger edges can be removed, that is, they are all uncertain trigger edges. Now suppose there are superfluous dependency edges in the execution graph whose absence does not affect the final database state. (Therefore we can remove them safely to increase parallelism.) There can be only two types of dependency edges in an acyclic execution graph. Given any dependency edge $(r_i \xrightarrow{D} r_j)$, $r_i$ is either a proper ancestor of $r_j$ in a trigger path containing both $r_i$ and $r_j$ or not an ancestor of $r_j$ in any trigger path. In the first case, even though the dependency edge is redundant in terms of representation, removal of that edge does not allow $r_j$ to execute before $r_i$ since $r_j$ is yet to be triggered by $r_i$ or its descendant. Thus, removal of the first type of dependency edges has no effect of increasing parallelism. In the second case, if $(r_i \xrightarrow{D} r_j)$ is the only dependency path that can be interconnected by trigger edges and dependency edges to connect $r_i$ to $r_j$, obviously this cannot be removed at any rate. If there exist other dependency paths connecting $r_i$ to $r_j$ whose lengths are longer than $(r_i \xrightarrow{D} r_j)$ (of course, if such paths exist, they should be longer than one-edge path $(r_i \xrightarrow{D} r_j)$), $(r_i \xrightarrow{D} r_j)$ is redundant, but again, removal of the dependency edge deos not allow $r_j$ to execute before $r_i$. By applying this argument to all dependency edges present in the extended execution graph, we can see that dependency edges are either necessary or redundant, but removal of redundant edges does not increase parallelism. Since the execution graph with rule_counts are equivalent to the extended execution graph under the strict order-preserving rule executions, we can conclude that the rule scheduler exploit the maximum parallelism inherent in the system rule set. □

# 6 Conclusions

In this paper we have proposed a new active-rule execution model, along with priority specification schemes, to achieve confluent rule execution in active databases. We employ prioritization to resolve conflicts between rules. Ideally, by prioritizing executions of conflicting rules, whose different relative execution orders can yield different database states, one can achieve confluent rule execution. It is necessary, however, to prioritize as few rules as possible primarily for two reasons: i) prioritizing many rules in a meaningful way is itself a challenge, and ii) the less rules prioritized increases the potential for parallel execution of rules. Prioritizing conflicting rules only, on the other hand, may yield incorrect results; execution of seemingly independent rules can trigger and execute conflicting rules in the wrong way. Also, additional problems arise when rules (in the same trigger path) are triggered and result in overlapped trigger paths. Unlike previous rule execution models, our model uses a rule scheduler based on the topological sort to respect all the specified priorities if applicable. This way, rules being triggered and executed from a user transaction can follow the execution sequence imposed by priority specifications to make their execution confluent. We also have proposed strict order-preserving rule execution to deal with overlapping trigger paths. In strict order-preserving rule execution, when a *part of* or the *whole* trigger path is multiply executed and there are priorities between rules in the trigger path, the rules are executed in such a way that no rule appears before rules with higher priorities if any. It has been shown that our rule execution model can exploit maximum parallelism in rule execution.

In order to handle overlapping trigger paths, we have explored other alternatives than the strict order-preserving rule execution. Serial trigger-path execution and serializable trigger-path execution are them, but due to space limit they are omitted in this paper. The full description of these alternative is found in [KC95].

We are currently investigating other issues not addressed in this paper. One issue is the precision of data dependency (or dependency in general). Our definition in Section 3.2 may be too coarse as some rules might be commutative despite presence of the defined dependencies. Another issue is related to coupling modes defined in HiPAC [C+89]. In this paper, we assumed the *immediate* coupling mode between. The semantics of confluent rule execution in the *deferred* and *detached* coupling modes needs to be addressed.

# References


[ACL91] R. Agrawal, R. Cochrane, and B. Lindsay. On maintaining priorities in a production rule system. In *Proceedings International Conference on Very Large Data Bases*, pages 479–487, Barcelona, Spain, 1991.

[AMC93] E. Anwar, L. Maugis, and S. Chakravarthy. A new perspective on rule support for object-oriented databases. In *Proceedings International Conference on Management of Data*, pages 99–108, Washington, D.C., May 1993.

[AWH92] A. Aiken, J. Widom, and J. Hellerstein. Behavior of database production rules: Termination, confluence, and observable determinism. In *Proceedings International Conference on Management of Data*, pages 59–68, San Diego, CA, 1992.

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

[C+89] S. Chakravarthy et al. Hipac: A research project in active, time-constrained database management (final report). Technical Report XAIT-89-02, Xerox Advanced Information Technology, Cambridge, MA, Aug. 1989.

[CKAK94] S. Chakravarthy, V. Krishnaprasad, E. Anwar, and S.-K. Kim. Composite events for active databases: Semantics, contexts, and detection. In *Proceedings International Conference on Very Large Data Bases*, pages 606–617, Santiago, Chile, Sep. 1994.

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

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

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

[Han92] E. Hanson. The design and implementation of the Ariel active database rule system. Technical Report UF-CIS-018-92, CIS Department, University of Florida, Gainesville, FL 32611, 1992.

[HW93] E. Hanson and J. Widom. An overview of production rules in database systems. *The Knowledge Engineering Review*, 8(3):121–143, Sep. 1993.

[KC95] Seung-Kyum Kim and S. Chakravarthy. A confluent rule execution model for active databases. Technical Report UF-CISE-, CISE Department, University of Florida, Gainesville, FL 32611, 1995.

[SHP88] M. Stonebraker, E. Hanson, and S. Potamianos. The POSTGRES rule manager. *IEEE Transactions on Software Engineering*, 14(7):897–907, July 1988.

[SJGP90] M. Stonebraker, A. Jhingran, J. Goh, and S. Potamianos. On rules, procedures, caching and views in database systems. In *Proceedings International Conference on Management of Data*, pages 281–290, Atlantic City, NJ, 1990.

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

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