On Maintaining Priorities in a Production Rule System

Rakesh Agrawal
ragrawal@ibm.com

Roberta Cochrane*
bobbie@cs.umd.edu

Bruce Lindsay
bruce@ibm.com

IBM Almaden Research Center
San Jose, California 95120

Abstract

We present a priority system which is particularly suited for production rules coupled to databases. In this system, there are default priorities between all rules and overriding user-defined priorities between particular rules. Rule processing using this system is *repeatable*: for a given set of rules and priorities, the rules are considered for execution in the same order if the same set of transactions is executed twice on the same initial database state. The rule order *adheres* to the default order as closely as possible: rules are considered in the same order as the default order unless user-defined precedence constraints force an inversion.

We present data structures and efficient algorithms for implementing such a priority system. We show how the data structures can be incrementally maintained as user-defined priorities are altered. We also discuss how the proposed scheme can be extended to build a multi-level hierarchical priority system.

1 Introduction

Incorporation of production rules into database systems has recently received considerable attention [6, 7, 8, 11, 19, 21, 25, 26, 27, 30]. A central issue in production rule systems is *conflict resolution* [20, 14]. Given that two or more rules are triggered, a conflict resolution mechanism determines which rule is considered first for execution. Some rule systems (for example, Postgres [26]) require that the rule definer specify an absolute numeric priority to conflicting rules which is used to resolve conflicts at run time. Other systems (for example, OPS5 [9]) use a combination of some static properties of the rules (such as the complexity of the antecedents) and some dynamic properties of the data (such as the recency of the tuples satisfying the rules) to determine relative priority. In the case that no criterion resolves the conflict, a rule is chosen randomly, making the rule system *non-deterministic*.

Non-determinism in production rule systems has led to systems that have turned out to be much more complex and unwieldy than had been expected [15], which in turn has inspired research into *deterministic* production rule systems [12, 24, 32]. Although not necessarily appropriate for all applications, deterministic production rule systems are more easily understood, maintained, and extended. They are particularly useful for rule bases coupled to databases, since the primary purpose of a rule base in such an environment is to automate deterministic activities [12].

We propose a new priority system for deterministic production rule systems that has the following attributes:

1. *Default Priorities*. The rules in the production rule system have *default relative priorities* that are a function of the *static* properties of the rules. This function, p, defines a *default total order* over the production rules. A function yielding the creation timestamp of the rules (assuming creation timestamps are unique) is an example of such a function which gives higher priority to older rules. Production order rules, described in [20], provide other examples of such a function. We represent the default total order by $\xrightarrow{d}$ such that, given two rules R and S, if $p(R) < p(S)$ then $R \xrightarrow{d} S$. Default priorities may be user-specified or induced by the system.

2. *User-Defined Priorities*. The user may explicitly

*Current address: Computer Science Department, University of Maryland, College Park, Maryland 20742

Proceedings of the 17th International Conference on Very Large Data Bases, Barcelona, Spain, September 1991

specify relative priorities between particular rules by defining a *precedes* relationship between them. If the user has specified that rule R *precedes* S, and if both R and S have been triggered, then R is considered first for execution, regardless of the default total ordering. User-defined priorities are transitive; that is, if R precedes S and S precedes T, then R precedes T even if S is not triggered. Cycles are not permitted in the user-defined priorities. $R \Rightarrow S$ represents that rule R has user-defined priority over S. We assume for convenience that for every rule R, $R \Rightarrow R$. If there are k rules T_k (k could be 0) such that $R \Rightarrow T_1 \Rightarrow T_2 \Rightarrow \ldots T_k \Rightarrow S$, we say $R \stackrel{*}{\Rightarrow} S$.

User-defined priorities override default priorities. The user may define priorities at the time of rule definition or separately. User-defined priorities are dynamic — they may be dropped and added at any point during the existence of the rule set.

Precedence relationships are a natural way of expressing user-defined priorities [29] because they increase rule autonomy [20]: they do not force the rule designer to know about all the rules in the system. Such relationships are also often the result of rule analysis [24] and rule generation [28], which specify only the precedences that must be satisfied.

3. *Repeatability*. If the same set of transactions is executed twice with the same database state, the same set of rules, and the same user-defined and default priorities between the rules, then all rules are considered in the same order. This repeatability property is important since it is essential for a system to have predictable behavior. The repeatability property can be guaranteed if, given a default total order $\stackrel{d}{\rightarrow}$ over a set of rules $\mathcal{R}$ and an overriding partial order $\stackrel{*}{\Rightarrow}$ over a subset of rules in $\mathcal{R}$, we can obtain a new unique total order. The new total order is represented by $\stackrel{n}{\rightarrow}$.

   The repeatability property is stricter than the *determinism* property considered in [12, 24, 32]. For example, [12] only requires that the production system have a unique fixed point, whereas the repeatability property insists that the computation path to the fixed point is also unique. However, [12] places constraints on rule sets to realize production systems with unique fixed points. The repeatability property guarantees determinism without constraining rule sets. Also, just having a unique fixed point can be inadequate for applications having side effects (an action external to the database, for example), and we need the stronger repeatability property.

4. *Adherence to Default Order*. The new total order $\stackrel{n}{\rightarrow}$ adheres to the default order to the extent permissible within user-defined precedence constraints. Starting with the first rule in the default order, the rules are put in the new order in the same order as the default order unless a user-defined priority forces a rule to come earlier. Consider, for example, the rule system consisting of rules R_0, R_1, R_2, and R_3, where the subscripts associated with the rules also denote their timestamps. Assume that the default order is to order the rules in increasing order of their timestamps, and the user-defined priorities are $R_3 \Rightarrow R_0$ and $R_2 \Rightarrow R_1$. If it weren't for $R_3 \Rightarrow R_0$, the adherence property would require that R_0 come before any other rule in $\stackrel{n}{\rightarrow}$, as R_0 is the first rule in the default order. However, due to user-defined priority of R_3 over R_0, R_3 comes first and then R_0. Having placed R_0, the adherence property requires that R_1 be placed next in $\stackrel{n}{\rightarrow}$. However, the user-defined priority of R_2 over R_1 forces that R_2 be placed before R_1, and thus $R_3 \stackrel{n}{\rightarrow} R_0 \stackrel{n}{\rightarrow} R_2 \stackrel{n}{\rightarrow} R_1$ is the new total order.

   Formally, for $\stackrel{n}{\rightarrow}$ to adhere to $\stackrel{d}{\rightarrow}$, it must be that $R \stackrel{d}{\rightarrow} S$ and $S \stackrel{n}{\rightarrow} R$ if and only if i) $S \stackrel{*}{\Rightarrow} R$, or ii) $S \stackrel{*}{\not\Rightarrow} R$ and $\exists T$ such that $S \stackrel{*}{\Rightarrow} T$, $R \stackrel{*}{\not\Rightarrow} T$, and $T \stackrel{d}{\rightarrow} U$ for all U such that $R \stackrel{*}{\Rightarrow} U$ and $S \stackrel{*}{\not\Rightarrow} U$. Otherwise, $R \stackrel{d}{\rightarrow} S$ and $R \stackrel{n}{\rightarrow} S$. In other words, if R precedes S in the default total order then their ordering is reversed in the new total order if and only if the user has specified that S must precede R or that S must precede a rule T that precedes in the default order all the rules that R must precede. Any rule that has been specified by the user to follow both R and S is ignored in this decision.

   The adherence property has a relationship to *inversions* [16] in the sense that $\stackrel{n}{\rightarrow}$ is an inversion of $\stackrel{d}{\rightarrow}$ that satisfies user-defined precedence constraints. In addition, the adherence property requires that this inversion be such that, starting with the first item in $\stackrel{d}{\rightarrow}$, items have the same order in $\stackrel{n}{\rightarrow}$ as in $\stackrel{d}{\rightarrow}$ unless a user-defined precedence dictates otherwise. This requirement resembles the priority-driven deadline scheduling of jobs in real-time systems [18, 4]. However, in deadline scheduling, if the deadline for a task is missed, the task may not be scheduled at all. On the contrary, rules are never dropped in rule systems (although a higher priority rule may cancel the firing of a lower priority rule).

The priority system proposed in this paper is the result of an effort to define a priority system for the Starburst Production Rule System [30]. An initial design [29] allowed the user to define relative priorities between some rules and required the rule system to be repeatable. However, the algorithm for determining ordering between rules in [29] can lead to cycles in the rule priorities and, hence, does not produce a total order. Letting $ts(R)$ represent the creation time of rule R, the ordering between two rules R and S in [29] is determined as follows:

1. If $R \stackrel{*}{\Rightarrow} S$ and $R \neq S$, then $R \stackrel{n}{\rightarrow} S$.
2. If $S \stackrel{*}{\Rightarrow} R$ and $S \neq R$, then $S \stackrel{n}{\rightarrow} R$.
3. Otherwise, if $ts(R) < ts(S)$, then $R \stackrel{n}{\rightarrow} S$ else $S \stackrel{n}{\rightarrow} R$.

However, consider rules R_0, R_1, and R_2, such that $ts(R_0) = 0$, $ts(R_1) = 1$, $ts(R_2) = 2$, and $R_2 \Rightarrow R_0$. $R_0 \stackrel{n}{\rightarrow} R_1$ since $R_0 \stackrel{*}{\not\Rightarrow} R_1$, $R_1 \stackrel{*}{\not\Rightarrow} R_0$, and $ts(R_0) < ts(R_1)$. Similarly, $R_1 \stackrel{n}{\rightarrow} R_2$. Also $R_2 \stackrel{n}{\rightarrow} R_0$, since $R_2 \Rightarrow R_0$. Thus, $R_0 \stackrel{n}{\rightarrow} R_1 \stackrel{n}{\rightarrow} R_2 \stackrel{n}{\rightarrow} R_0$, a cycle.

The problem of task allocation with precedence relations [5, 17, 31] has similarities to the priority problem considered in this paper. Task allocation with precedence relations also considers the effect of precedence relations between modules on task scheduling. The precedence relations put constraints on the final order, and the total order satisfies the partial order imposed by these relations. However, conflicts are resolved using dynamic information about the jobs, which does not necessarily impose a total order. We, on the other hand, use the adherence property to arrive at the unique total order.

The organization of the remainder of the paper is as follows. In Section 2, we present an algorithm for determining the order between two rules given a default total order over a set of rules and an overriding partial order over some rules in this set. We show that this algorithm leads to a new total order that *adheres* to the default order and guarantees *repeatability*. Section 3 discusses efficient implementation of this algorithm. Section 4 describes how changes in the user-defined priorities between rules can be handled incrementally.

Section 5 shows how our scheme can be extended to build a hierarchical priority system. Related rules are grouped into *rule classes*, as in [13]. User-defined priorities are specified separately for rules in each class and also for rule classes themselves. This scheme extends naturally to multi-level hierarchies. We conclude with a summary in Section 6.

2 Rule Ordering

Definition 1 (Distinguished Rule) Given two rules R and S and an ordering function p that determines the default total order, the *distinguished rule* for R with respect to S and p, $d(R)_{S,p}$, is defined as follows:

1. If $S \stackrel{*}{\Rightarrow} R$, then $d(R)_{S,p} = R$.
2. If $S \stackrel{*}{\not\Rightarrow} R$, then $d(R)_{S,p}$ is defined to be the rule T such that all of the following hold:
   (a) $R \stackrel{*}{\Rightarrow} T$,
   (b) $S \stackrel{*}{\not\Rightarrow} T$, and
   (c) $\forall\ U$ such that $R \stackrel{*}{\Rightarrow} U$ and $S \stackrel{*}{\not\Rightarrow} U$, $p(U) \geq p(T)$.

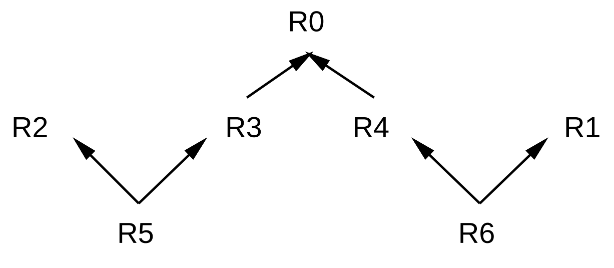
For example, assuming that p is the function yielding the creation time of a rule and that $S \stackrel{*}{\not\Rightarrow} R$, the distinguished rule for rule R with respect to rule S is the oldest rule T that R must precede and that S does not precede in the user-defined priority ordering. Note that $d(R)_{S,p}$ always exists and is unique. Also, $d(R)_{S,p}$ could be R itself.

Algorithm 1 (Relative Rule Ordering) Given two rules R and S and an ordering function p that determines the default total order, applying the following two steps in order determines the relative ordering between two rules R and S:

1. If $R \stackrel{*}{\Rightarrow} S$ and $R \neq S$, then $R \stackrel{n}{\rightarrow} S$. If $S \stackrel{*}{\Rightarrow} R$ and $S \neq R$, then $S \stackrel{n}{\rightarrow} R$.
2. Otherwise, let U be $d(R)_{S,p}$ and let V be $d(S)_{R,p}$. If $p(U) < p(V)$, then $R \stackrel{n}{\rightarrow} S$; otherwise, $S \stackrel{n}{\rightarrow} R$.

That is, the relative ordering between two rules is determined by the user-defined priority (direct or transitive) between them when there is one. Otherwise, the relative ordering is determined by the relative value of the default ordering function p for their respective distinguished rules.

For example, let the rule system consist of rules R_0, R_1, R_2, R_3, R_4, R_5, and R_6, where the subscripts associated with the rules also denote their creation time. Let the default total order be determined by the creation time of the rules, and let the user-defined priorities be as illustrated below. Then, $R_6 \stackrel{n}{\rightarrow} R_5$ because $d(R_6)_{R_5,ts} = R_1$, $d(R_5)_{R_6,ts} = R_2$, and $ts(R_1) < ts(R_2)$.



Theorem 1 () (Repeatability and adherence of the relative rule ordering algorithm) Given a set of rules $\mathcal{R}$, the pairwise application of the relative rule ordering algorithm over rules in $\mathcal{R}$ generates a repeatable and adherent total order.

Proof: See Appendix A. □

3 Implementation

We now discuss how the relative rule ordering algorithm can be implemented efficiently.

Definition 2 (Rule Ordering Graph) The rule ordering graph G for a given ordering function p and a given set of production rules $\mathcal{R}$ is obtained as follows:

1. Corresponding to each rule R in $\mathcal{R}$, create a node R in G. Associate with node R the value of the default ordering function $p(R)$.
2. For each user defined priority $R \Rightarrow S$, create an arc from node R to node S in G.
3. Create an arc from every node R to itself.

The following are the data structures for the rule ordering algorithm:

1. Obtain the rule ordering graph G for the given rule set $\mathcal{R}$.
2. Compute the transitive closure G^* of graph G [2].
3. Sort the successors of every node R in G^* in ascending order of their ordering function values.

Given two rules `r1` and `r2`, the following function returns the rule that has the higher precedence:

```
function precedence(rule r1, rule r2)
                              returns rule
 {
  loop
   {
    s1 = next(successor(r1));
    s2 = next(successor(r2));

    // first use the user-defined
    // precedence, if any

    if (s1 == r2)
        return  r1;
    else if (s2 == r1)
        return  r2;

    // now use the default precedence

    else if (p(s1) < p(s2))
        return  r1;
    else if (p(s2) < p(s1))
        return  r2;
    else //  p(s2) == p(s1)
        continue;
    }
  }
```

It may not be immediately obvious why the loop in the above `precedence` function always terminates before the shorter of the successor lists of `r1` and `r2` runs out. Also, if `r2` $\Rightarrow$ `r1`, then `r1` is not necessarily the first rule in the successor list of `r2` — there may some other rule `s2` that comes before `r1` in the successor list of `r2`. Why does the loop return the correct answer even in this case? The following theorem comes to our rescue:

Theorem 2 (Correctness of the precedence function) The function `precedence` generates the same relative ordering between two rules as the relative rule ordering algorithm.

Proof: See Appendix B. □

The total order $\stackrel{n}{\rightarrow}$ may be constructed by sorting all the rules in $\mathcal{R}$, using for comparison the `precedence` function.

4 Addition and Deletion of Rules and Rule Priorities

Rule systems are not static. Rules are continuously added and deleted, and user-defined priorities between existing rules are altered. One alternative is to form a new rule ordering graph G and compute its transitive closure G^*, every time rules and/or priorities are added or deleted. However, instead of recomputing G^* from scratch, we can incrementally update G^*. The problem of incrementally updating compressed transitive closure has been considered in [1] and that of incrementally updating path information in [3]. The techniques we present here apply to the general problem of incremental maintenance of complete transitive closure of acyclic directed graphs with sorted successors.

4.1 Incremental Additions

Addition of a new rule R simply results in the creation of a new node R in G^*. Also, there is an arc from R to R in G^*.

When the user wants to add a new priority for rule R over rule S, we need to first ensure that $S \overset{*}{\Rightarrow} R$ does not already exist; otherwise, the creation of $R \Rightarrow S$ will cause a cycle in the user-defined priorities. If $R \Rightarrow S$ is a legal addition, then the successor list of every predecessor of R needs to be updated, as they can now reach S and all the successors of S.

The following procedure incrementally updates G^* when a new user-defined priority $R \Rightarrow S$ is added:

```
// Addition of the user-defined priority,
// R => S

procedure addpriority(rule R, rule S)
 {
  // check for potential cycle in the
  // user-defined priorities

  if R is a successor of S in G*
   {
    disallow priority of R over S;
    return;
   }

  // legal user-defined priority ---
  // update data structures

  L = successors(S); // new reachable
                     // successors;
                     // S is included
                     // in successors(S)
  add(R,L);
 }
```

```
// Recursive procedure that adds to R and all
// its predecessors the rules in L

procedure add(rule R, list L)
 {
  // omit from L those rules that are
  // already in the successor list of R

  L = L - successors(R);

  // Add rules in L to the successor list
  // of R and its predecessors,
  // maintaining the correct order

  if L is not empty
   {
    // update the successor list of R
    successors(R) = successors(R) + L;
                    // maintain order

    // update the successor list of
    // predecessors of R
    for all P such that
      P is an immediate predecessor of R do
        add(P, L)
   }
 }
```

The recursion terminates when all the predecessors of R have been updated. It is possible that the add procedure is not executed for some predecessor P if L becomes empty for all its successors.

Multiple visits to a predecessor of R can be avoided by some book-keeping. The first time a predecessor is visited, a bit is set for this rule indicating that this rule has already been visited. Now, before calling add for a predecessor P, this bit is tested to ensure that P has not been already visited.

4.2 Incremental Deletions

Deletion of a user-defined priority $R \Rightarrow S$ does not necessarily imply that S and all its successors should be deleted from the successor list of R and all its predecessors — there may be alternative paths.

The following procedure incrementally updates G^* when a user-defined priority $R \Rightarrow S$ is deleted:

```
// Deletion of the user-defined priority,
// R => S

procedure deletepriority(rule R, rule S)
 {
  L =  successors(S);  // rules potentially
               // unreachable from R via S;
          // S is included in successors(S)
```

```
  delete(R, S, L);
 }
```

```
// Recursive procedure that deletes from R
// and all its predecessors the rules in L
// that are not anymore reachable from them

procedure delete(rule R, rule S, list L)
 {
  // omit from L those rules for which
  // alternative path exists

  L = L -
    successors(immediate-successors(R) - S);

  // Delete rules in L from the successor
  // list of R and its predecessors

  if L is not empty
   {
    // update the successor list of R
    successors(R) = successors(R) - L;

    // update the successor list of
    // predecessors of R
    for all P such that
      P is an immediate predecessor of R do
        delete(P, R, L)
   }
 }
```

As in the case of incremental addition, the recursion terminates when all the predecessors of R have been updated. It is possible that the `delete` procedure is not executed for some predecessor P of R if L becomes empty for at least one node on every path from P to R.

However, it is incorrect to apply the marking optimization discussed with `addpriority` to avoid multiple visits to a predecessor of R. The reason is that, to propagate the addition of a rule l in L to some predecessor P of R, it is sufficient to add l to one of the successors of P and then let P inherit l from this successor. However, to propagate the deletion of a rule l in L to some predecessor P of R, l must not be reachable from any successor of P. If l is only reachable from P through R, then l will only be deleted from P on the last visit to node P.

Deletion of a rule R results in the deletion of all incoming arcs into R and all outgoing arcs from R in G. The `deletepriority` procedure can be applied for each such arc, followed by the deletion of the node R in G.

5 Hierarchical Priority System

Rules are sometimes grouped into rule classes, as in [13]. Rule classes are useful for structuring problem-solving by allowing related rules to be bundled into a separate class. User-defined priorities may be specified between rule classes and between rules within a class. The algorithm presented in Section 3 can be extended to handle such a hierarchical priority system:

1. Create a *class ordering graph* $\mathcal{CG}$ as follows:
   i. For every rule class C, create a node C in $\mathcal{CG}$. Associate with a node C a value which is the smallest of the value of the application of the default ordering function p on all the rules in C.
   ii. Create an arc C to D in $\mathcal{CG}$ if the rule class C has been specified to have a priority over the rule class D.
   iii. For every rule class C, create an arc from C to C in $\mathcal{CG}$.
2. Compute the transitive closure $\mathcal{CG}^*$ of $\mathcal{CG}$.
3. Create a rule ordering graph G and its transitive closure G^* separately for each rule class as in Section 3.
4. Now to determine the relative precedence between two rules, use $\mathcal{CG}^*$ if they belong to different rule classes, and use the corresponding G^* if they belong to the same rule class.

The preceding algorithm can be extended in a straightforward manner to handle multi-level class hierarchies. However, a limitation of this algorithm is that it does not directly admit the user-defined precedence between rules in different classes.

6 Summary

We presented a priority system that is incrementally maintainable for combining user-defined priorities with default priorities. Such priority systems are becoming increasingly important in integrating production systems with database systems which require deterministic behavior. Precedence relationships are a natural way of expressing user-priorities [29] because they increase rule autonomy [20]: they do not force the rule designer to know about all the rules in the system. Such relationships are also often the result of rule analysis [24] and rule generation [28], which specify only the precedences that must be satisfied.

Rule processing using this priority system is repeatable: for a given set of rules and priorities, the rules are considered for execution in the same order if the same set of transactions is executed twice on the same initial database state. The rule order *adheres* to the default order as closely as possible: rules are considered in the same order as the default order unless user-defined precedence constraints force an inversion.

We also presented data structures and efficient algorithms for implementing such a priority system. User-defined priorities are dynamic — new priorities may be added and existing priorities may be deleted or altered. We showed how data structures required for priority determination can be incrementally maintained. Finally, we showed how the proposed scheme can be extended to build a multi-level hierarchical priority system.

We are considering the implementation of this priority system in the Starburst extensible database system [10].

7 Acknowledgements

Thanks to Joe Hellerstein, Tomasz Imielinski, and Jennifer Widom for helpful discussions and suggestions, and to Laura Haas and Guy Lohman for feedback.

A Appendix – Correctness of the relative rule ordering algorithm

In this appendix, we prove that the relation $\stackrel{n}{\to}$ defined by the relative rule ordering algorithm is a repeatable, adherent, total ordering of any given of set rules $\mathcal{R}$ by proving several lemmas. Repeatability is satisfied by the fact that $\stackrel{n}{\to}$ satisfies a total order, and adherence has its own lemma. Let $(\Rightarrow\Leftarrow)$ denote contradiction.

Lemma 1 (Uniqueness) If R and S are two distinct rules in $\mathcal{R}$ and $R \stackrel{n}{\to} S$, then $S \stackrel{n}{\not\to} R$.

Proof: Suppose $R \stackrel{n}{\to} S$ and $S \stackrel{n}{\to} R$ for two distinct rules in $\mathcal{R}$. Now, $R \stackrel{*}{\Rightarrow} S$ and $S \stackrel{*}{\Rightarrow} R$ cannot both hold since there are no cycles in the user-defined priorities. If $R \stackrel{*}{\Rightarrow} S$, then $R \stackrel{n}{\to} S$, and $S \stackrel{n}{\not\to} R$ since the first step of the algorithm is always applied first. A similar argument holds if $S \stackrel{*}{\Rightarrow} R$. So $p(d(S)_{R,p}) < p(d(R)_{S,p})$ and $p(d(S)_{R,p}) > p(d(R)_{S,p})$ must both be true. This is not possible since $<$ is unique $(\Rightarrow\Leftarrow)$. □

Lemma 2 (Totality) Between every pair of distinct rules R and S in $\mathcal{R}$, either $R \stackrel{n}{\to} S$ or $S \stackrel{n}{\to} R$.

Proof: Consider two distinct rules R and S in $\mathcal{R}$. If there is a user-defined priority between these two rules, then obviously $\stackrel{n}{\to}$ holds between these two rules. If there is not a user-defined priority between the rules, then $d(R)_{S,p}$ and $d(S)_{R,p}$ determine their relative ordering. Now, $p(d(R)_{S,p}) < p(d(S)_{R,p})$ or $p(d(R)_{S,p}) > p(d(S)_{R,p})$ since $d(R)_{S,p}$ and $d(S)_{R,p}$ are distinct and p is a total order. Therefore, either $R \stackrel{n}{\to} S$ or $S \stackrel{n}{\to} R$. □

Lemma 3 (Adherence) $p(R) < p(S)$ and $S \stackrel{n}{\to} R$ if and only if i) $S \stackrel{*}{\Rightarrow} R$, or ii) $S \stackrel{*}{\Rightarrow} T$ and $p(T) < p(U)$, $\forall$ U such that $R \stackrel{*}{\Rightarrow} U$ and $S \stackrel{*}{\not\Rightarrow} U$. Otherwise, $p(R) < p(S)$ and $R \stackrel{n}{\to} S$.

Proof: (if) Suppose $p(R) < p(S)$ and $S \stackrel{n}{\to} R$. By definition of $S \stackrel{n}{\to} R$, either $S \stackrel{*}{\Rightarrow} R$ satisfying (i), or $p(d(S)_{R,p}) < p(d(R)_{S,p})$. Now, $S \stackrel{*}{\Rightarrow} d(S)_{R,p}$, $R \stackrel{*}{\Rightarrow} d(R)_{S,p}$, $S \stackrel{*}{\not\Rightarrow} d(R)_{S,p}$ and $\forall$ U such that $R \stackrel{*}{\Rightarrow} U$ and $S \stackrel{*}{\not\Rightarrow} U$, $p(d(R)_{S,p}) <= U$, so $p(d(S)_{R,p}) < U$, satisfying (ii).

(only if) Suppose $S \overset{*}{\Rightarrow} R$. Then, by definition, $S \overset{n}{\rightarrow} R$ even if $p(R) < p(S)$.
Suppose (ii) is satisfied. Then $p(d(S)_{R,p}) <= p(T)$, and $d(R)_{S,p}$ is the U with the minimal value of $p(U)$. So $p(d(S)_{R,p}) < p(d(R)_{S,p})$ and $S \overset{n}{\rightarrow} R$.
Therefore, $\overset{n}{\rightarrow}$ is adherent. □

Lemma 4 (Transitivity) If R, S, and T are distinct rules in $\mathcal{R}$ such that $R \overset{n}{\rightarrow} S$ and $S \overset{n}{\rightarrow} T$, then $R \overset{n}{\rightarrow} T$.

Proof: Suppose R, S, and T are distinct rules in $\mathcal{R}$ such that $R \overset{n}{\rightarrow} S$ and $S \overset{n}{\rightarrow} T$. Then exactly one of the following holds:

1. $R \overset{*}{\Rightarrow} S$ and $S \overset{*}{\Rightarrow} T$.
   Then $R \overset{*}{\Rightarrow} T$ since user-defined priorities are transitive and acyclic.
   Hence, $R \overset{n}{\rightarrow} T$.

2. $R \overset{*}{\Rightarrow} S$ and $p(d(S)_{T,p}) < p(d(T)_{S,p})$.
   Now, $T \overset{*}{\not\Rightarrow} R$, otherwise $T \overset{n}{\rightarrow} S$.
   Since $R \overset{*}{\Rightarrow} S$, $p(d(R)_{T,p}) <= p(d(S)_{T,p})$ and $p(d(T)_{S,p}) <= p(d(T)_{R,p})$. So, $p(d(R)_{T,p}) < p(d(T)_{R,p})$.
   Hence, $R \overset{n}{\rightarrow} T$.

3. $p(d(R)_{S,p}) < p(d(S)_{R,p})$ and $S \overset{*}{\Rightarrow} T$.
   $T \overset{*}{\not\Rightarrow} R$, otherwise $S \overset{n}{\rightarrow} R$.
   Since $S \overset{*}{\Rightarrow} T$, $p(d(S)_{R,p}) <= p(d(T)_{R,p})$ and $p(d(R)_{T,p} <= p(d(R)_{S,p}$. So, $p(d(R)_{T,p} < p(d(T)_{R,p})$.
   Hence $R \overset{*}{\Rightarrow} T$.

4. $p(d(R)_{S,p}) < p(d(S)_{R,p})$ and $p(d(S)_{T,p}) < p(d(T)_{S,p})$. In order to prove this case, we first show that $T \overset{*}{\not\Rightarrow} R$, and then prove by contradiction that $p(d(R)_{T,p}) < p(d(T)_{R,p})$. The following observation is useful:

   Observation 1 $\forall X \ni p(X) < p(d(P)_Q)$ if $P \overset{*}{\Rightarrow} X$, then $Q \overset{*}{\Rightarrow} X$.

   Suppose $T \overset{*}{\Rightarrow} R$. Then $p(d(T)_{S,p}) <= p(d(R)_{S,p})$ and $p(d(S)_{R,p}) <= p(d(S)_{T,p})$, so $p(d(T)_{S,p}) < p(d(S)_{T,p})$ ($\Rightarrow\Leftarrow$). So, $T \overset{*}{\not\Rightarrow} R$.

   Suppose $p(d(R)_{T,p}) > p(d(T)_{R,p})$.
   Consider $p(d(S)_{R,p})$ and $p(d(S)_{T,p})$.

   (a) Suppose $p(d(S)_{R,p}) <= p(d(S)_{T,p})$. Then $p(d(R)_{S,p}) < p(d(S)_{T,p} < p(d(T)_{S,p})$.
   Further consider $p(d(R)_{T,p})$ and $p(d(R)_{S,p})$.

      i. Suppose $p(d(R)_{T,p}) > p(d(R)_{S,p})$. Now, $R \overset{*}{\not\Rightarrow} d(R)_{S,p}$, so $T \overset{*}{\Rightarrow} d(R)_{S,p}$. Obviously, $S \not\Rightarrow d(R)_{S,p}$, so $p(d(T)_{S,p}) <= p(d(R)_{S,p})$. But $p(d(R)_{S,p}) < p(d(S)_{T,p})$, so $p(d(T)_{S,p}) < p(d(S)_{T,p})$ ($\Rightarrow\Leftarrow$).

      ii. Suppose $p(d(R)_{T,p}) <= p(d(R)_{S,p})$. Then $p(d(T)_{R,p}) < p(d(S)_{R,p})$. But, by assumption $p(d(S)_{R,p}) <= p(d(S)_{T,p})$, so $p(d(T)_{R,p}) < p(d(T)_{S,p})$, and according to the observation, $S \overset{*}{\Rightarrow} d(T)_{R,p}$, so $p(d(S)_{R,p}) <= p(d(T)_{R,p})$ ($\Rightarrow\Leftarrow$).

   (b) Suppose $p(d(S)_{R,p}) > p(d(S)_{T,p})$. Recall that $p(d(R)_{S,p}) < p(d(S)_{R,p})$, so $R \overset{*}{\Rightarrow} d(S)_{T,p}$. Obviously, $T \not\Rightarrow d(S)_{T,p}$, so $p(d(R)_{T,p}) <= p(d(S)_{T,p})$. But the assumption $(p(d(T)_{R,p}) < p(d(R)_{T,p}))$ implies $p(d(T)_{R,p}) < p(d(S)_{T,p})$. Then $S \overset{*}{\Rightarrow} d(T)_{R,p}$, so $p(d(S)_{R,p}) <= p(d(T)_{R,p})$. But this implies $p(d(S)_R) < p(d(S)_T)$ ($\Rightarrow\Leftarrow$).

   Hence, $p(d(R)_{T,p}) < p(d(T)_{R,p})$, so $R \overset{n}{\rightarrow} T$.

So, in all cases, $R \overset{n}{\rightarrow} T$. □

Lemma 5 (Total Order) The relation $\overset{n}{\rightarrow}$ is a total order.

Proof: Since $\overset{n}{\rightarrow}$ is transitive, unique, and total, $\overset{n}{\rightarrow}$ defines a total order. □

B Appendix – Correctness of the precedence function

In this appendix we establish that the function `precedence` (Section 3) generates the same relative ordering between two rules as the rule ordering algorithm (Algorithm 1). We must prove that the loop terminates, and at termination, `r1` is returned if and only if `r1`$\overset{n}{\rightarrow}$`r2`; `r2` is returned if and only if `r2`$\overset{n}{\rightarrow}$`r1`. Annotate the function as follows:

```
function precedence(rule r1, rule r2)
                              returns rule
  {
    loop
(A)    {
       s1 = next(successor(r1));
       s2 = next(successor(r2));

(B)    // first use the user-defined
       // precedence, if any

       if (s1 == r2)
(C)        return  r1;
```

```
        else if (s2 == r1)
(D)         return  r2;

        // now use the default precedence

        else if (p(s1) < p(s2))
(E)         return  r1;
        else if (p(s2) < p(s1))
(F)         return  r2;
        else //  p(s2) == p(s1)
            continue;
      }
  }
```

Lemma 6 (Loop Invariant) Assuming `s1` and `s2` are initially `NULL`, `s1` $==$ `s2` at `(A)` for each iteration.

Proof: This is clearly the case in the first iteration, since `s1` $==$ `NULL` $==$ `s2`.

The loop terminates whenever `s1`$! =$ `s2` since `p` is a total order. If `s1`$! =$ `s2` then `p(s1)`$! =$ `p(s2)`, and the loop exits at `(E)` with `p(s1)` $<$ `p(s2)`, or at `(F)` with `p(s1)` $>$ `p(s2)`. □

Lemma 7 (Termination) The loop does not execute indefinitely.

Proof: The loop will terminate since `r1` and `r2` are both in their own list of successors and there is a finite number of rules.

Suppose `r1`$\stackrel{*}{\Rightarrow}$`r2` or `r2`$\stackrel{*}{\Rightarrow}$`r1`. Then the loop will terminate at either `(C)` or `(D)` if not before.

Suppose `r1`$\stackrel{*}{\not\Rightarrow}$`r2` and `r2`$\stackrel{*}{\not\Rightarrow}$`r1`. Then the loop will terminate when `s1` $==$ `r1` or `s2` $==$ `r2`, if not before, since `r1` is not in `r2`'s successor list and `r2` is not in `r1`'s successor list. □

Observation 2 By the definition of `next(successor())`,

- `p(s1)` $>$ all previously visited successors of `r1`,
- `p(s1)` $<$ all unvisited successors of `r1`,
- `p(s2)` $>$ all previously visited successors of `r2`, and
- `p(s2)` $<$ all unvisited successors of `r2`.

Lemma 8 (Correctness of Function) `Precedence` returns `r1` if and only if `r1`$\stackrel{n}{\rightarrow}$`r2`, and `precedence` returns `r2` if and only if `r2`$\stackrel{n}{\rightarrow}$`r1`.

Proof:

1. Suppose `precedence` returns `r1`. Then the loop exited at either `(C)` or `(E)`.

   If the loop exited at `(C)` then `s1` $==$ `r2`. So `r2` is in `r1`'s successor list. So `r1`$\stackrel{*}{\Rightarrow}$`r2` and `r1`$\stackrel{n}{\rightarrow}$`r2`.

   If the loop exited at `(E)`, then `p(s1)` $<$ `p(s2)`. Let `s1a` be the value of `s1` and `s2a` be the value of `s2` in the iteration preceding loop termination. Now `p(s1a)` $==$ `p(s2a)` and `p(s1a)` $<$ `p(s1)` $<$ `p(s2)`. So, by observation 2, `s1` is a not in `r2`'s successor list. Note, however, that `s2` might be in `r1`'s successor list.

   Suppose there is a user precedence between `r1` and `r2`. Now, it cannot be the case that `r2`$\stackrel{*}{\Rightarrow}$`r1`, because then `s1` would be in `r2`'s successor list. So `r1`$\stackrel{*}{\Rightarrow}$`r2` and `r1`$\stackrel{n}{\rightarrow}$`r2`.

   Suppose there is a not a user precedence between `r1` and `r2`. By observation 2 and the loop invariant, `s1` $==$ $\mathtt{d(r1)}_{\mathtt{r2,p}}$ and `s2` $==$ $\mathtt{d(r2)}_{\mathtt{r1,p}}$ Since `p(s1)` $<$ `p(s2)`, `r1`$\stackrel{n}{\rightarrow}$`r2`.

2. Suppose `precedence` returns `r2`. Then the loop exited at `(D)` or `(F)`. The proof that `r2`$\stackrel{n}{\rightarrow}$`r1` follows in the same fashion as (1).

3. Suppose `r1`$\stackrel{n}{\rightarrow}$`r2`. The loop terminates at exactly 4 points. Suppose `precedence` does not return `r1`. Then `precedence` returns `r2`. But then, by (2), `r2`$\stackrel{n}{\rightarrow}$`r1`. But $\stackrel{n}{\rightarrow}$ is unique. ($\Rightarrow\Leftarrow$). So `precedence` must return `r1`.

4. Suppose `r2`$\stackrel{n}{\rightarrow}$`r1`. The proof that `precedence` returns `r2` follows in the same fashion as (3).

Therefore, the function `precedence` is correct. □

References

[1] Rakesh Agrawal, Alexander Borgida, and H. V. Jagadish. Efficient Management of Transitive Relationships in Large Data and Knowledge Bases. In Proc. SIGMOD 89 [23], pages 253–262.

[2] Rakesh Agrawal, Shaul Dar, and H. V. Jagadish. Direct Transitive Closure Algorithms: Design and Performance Evaluation. *ACM Transactions on Database Systems*, 15(3):427–458, September 1990.

[3] Rakesh Agrawal and H.V. Jagadish. Materialization and Incremental Update of Path Information. In Proc. DE 89 [22], pages 374–383.

[4] A. Buchmann, D. McCarthy, M. Hsu, and U. Dayal. Time-critical database scheduling: A framework for integrating real-time scheduling and concurrency control. In Proc. DE 89 [22], pages 470–480.

[5] W. W. Chu and L. M-T. Lan. Task Allocation and Precedence Relation for Distributed Real-Time Systems. *IEEE Transactions on Computers*, C-36(6):667–679, June 1987.

[6] D. Cohen. Compiling Complex Database Transition Triggers. In Proc. SIGMOD 89 [23], pages 225–234.

[7] C. de Maindreville and E. Simon. A Production Rule Based Approach to Deductive Databases. In *Proc. 4th IEEE International Conference on Data Engineering*, pages 234–241, Los Angeles, February 1988.

[8] L. M. L. Delcabmre and J. N. Etheredge. The Relational Production Language: A Production Language for Relational Databases. In *Proc. 2nd International Conference on Expert Database Systems*, pages 153–162, Tysons Corner, Virginia, April 1988.

[9] C. L. Forgy. *On the Efficient Implementation of Production Systems.* PhD thesis, CMU, February 1979.

[10] L.M. Haas, W. Chang, G.M. Lohman, J. McPherson, P.F. Wilms, G. Lapis, B. Lindsay, H. Pirahesh, M. Carey, and E. Shekita. Starburst Mid-Flight: As the Dust Clears. *IEEE Transactions on Knowledge and Data Engineering*, pages 143–160, March 1990.

[11] E. Hanson. An Initial Report on the Design of Ariel: A DBMS with an Integrated Production Rule System. *ACM SIGMOD Record*, 18(3):12–19, September 1989.

[12] Joseph M. Hellerstein and Meichun Hsu. Determinism in Partially Ordered Production Systems. Research Report RJ 8009, IBM Almaden Research Center, March 1991.

[13] IBM. *IBM Knowledge Engineering Environment/370 (KEE/370), User's Guide, Release 1*, December 1988.

[14] Yannis E. Ioannidis and Timos K. Sellis. Conflict Resolution of Rules Assigning Values to Virtual Attributes. In Proc. SIGMOD 89 [23], pages 205–214.

[15] Peter Jackson. *Introduction to Expert Systems.* Addison-Wesley Publishing Co., 1986.

[16] Donald Ervin Knuth. *Sorting and Searching*, volume 3 of *The Art of Computer Programming.* Addison-Wesley Publishing Co., 1973.

[17] E. L. Lawler. Optimal Sequencing of a Single Machine Subject to Precedence Constraints. *Management Science*, 19(5):544–546, January 1973.

[18] C. L. Liu and James W. Layland. Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment. *Journal of the ACM*, 20(1):46–61, January 1973.

[19] D. McCarthy and U. Dayal. The architecture of an active database management system. In Proc. SIGMOD 89 [23].

[20] J. McDermott and C. Forgy. Production System Conflict Resolution Strategies. In D.A. Waterman and Fredrick Hayes-Roth, editors, *Pattern Directed Inference Systems*, pages 177–199. Academic Press, 1978.

[21] M. Morgenstern. Active Databases as a Paradigm for Enhanced Computing Environments. In *Proc. 9th International Conference on Very Large Data Bases*, pages 34–42, Florence, Italy, October 1983.

[22] *Proc. 5th IEEE International Conference on Data Engineering*, Los Angeles, February 1989.

[23] *Proc. ACM-SIGMOD International Conference on Management of Data*, Portland, May-June 1989.

[24] Louiqa Raschid. Maintaining Consistency in a Stratified Production System Program. In *Proc. AAAI National Conference on Artificial Intelligence*, 1990.

[25] T. Sellis, C.-C. Lin, and L. Raschid. Implementing Large Production Systems in a DBMS Environment: Concepts and Algorithms. In *Proc. ACM-SIGMOD International Conference on Management of Data*, pages 404–412, Chicago, June 1988.

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

[27] A. Tzvieli. On the Coupling of a Production System Shell and a DBMS. In *Proc. 3rd International Conference on Data and Knowledge Bases - Improving Usability and Responsiveness*, pages 291–309, Jerusalem, June 1988.

[28] J. Widom and S. Ceri. Deriving Production Rules for Constraint Maintenance. In *Proc. 16th International Conference on Very Large Data Bases*, pages 566–577, Brisbane, August 1990.

[29] J. Widom, R. J. Cochrane, and B. G. Lindsay. Implementing Set-Oriented Production Rules as an Extension to Starburst. Research Report RJ 7979, IBM Almaden Research Center, February 1991.

[30] J. Widom and S.J. Finkelstein. Set-Oriented Production Rules in Relational Database Systems. In *Proc. ACM-SIGMOD International Conference on Management of Data*, pages 259–270, Atlantic City, May 1990.

[31] J. Xu and D. L. Parnas. Scheduling Processes with Release Times, Deadlines, Precedence, and Exclusion Relations. *IEEE Transactions on Software Engineering*, SE-16(3):360–369, March 1990.

[32] Yuli Zhou and Meichun Hsu. A Theory for Rule Triggering Systems. In Francois Bancilhon, Costantino Thanos, and Dennis Tsichritzis, editors, *Proc. International Conference on Extending Data Base Technology*, Advances in Database Technology - EDBT '90. Lecture Notes in Computer Science, Volume 416, Venice, March 1990. Springer-Verlag.