A Syntax and Semantics for Set-Oriented Production Rules in Relational Database Systems

(Extended Abstract)

Jennifer Widom
Sheldon J. Finkelstein

IBM Almaden Research Center
650 Harry Road
San Jose, CA 95120
widom@ibm.com, shel@ibm.com

We propose incorporating a production rules facility into a relational database system. Such a facility allows definition of database operations that are automatically executed whenever certain conditions are met. In keeping with the set-oriented approach of relational data manipulation languages, our production rules are also set-oriented—they are triggered by sets of changes to the database and may consequently perform sets of changes. The condition and action parts of our production rules may refer to the current state of the database as well as to the sets of changes triggering the rules. We define a syntax for production rule definition as an extension to SQL. A model of system behavior is used to give an exact semantics for production rule execution, taking into account externally-generated operations, self-triggering rules, and simultaneous triggering of multiple rules.

Due to space constraints, some details and discussion are omitted, and only a few examples are included. See [19] for a more extensive description.

1 Introduction

Recently, there has been considerable interest in integrating production rules systems and database management systems. Some work, such as [6,15,18], focuses on using database technology to efficiently support OPS-like production rules languages [1]. Other work—including ours—focuses on extending database systems to include a production rules facility [4,5,7,11,13,14,17]. Generally, production rules take the form *when X then Y*, where *X* is a triggering condition and *Y* is an action: whenever condition *X* is true, action *Y* is performed. Specific forms of such rules may be used in particular environments.

We consider production rules in the context of relational database systems. A number of potential uses motivate the addition of such a facility. Production rules are a natural mechanism for integrity constraint enforcement; more generally, they permit additional semantic structure for the database. In active database systems [13], production rules are used to monitor particular conditions (sometimes associated with timing constraints) and, when appropriate, to trigger corresponding actions. Production rules may be useful for authorization checking and for maintenance of derived data. Finally, production rules in database systems should provide a flexible framework for building efficient knowledge-based and expert systems.

We propose a production rules facility compatible with the SQL data manipulation language [10], although our framework could apply equally well to other relational database languages. Rule activation results from user-generated (or application-generated) database operations. In this paper, we focus on the syntax of rule definition and the semantics of rule execution. Our syntax is straightforward, based directly on SQL. The semantics of rule execution is somewhat more complicated. A number of issues must be addressed:

- What causes a rule to be triggered? Is it a database state, a transition from one state to another, either, both?
- If rules can be triggered by state transitions, what exactly constitutes a transition? An operation on a single tuple? A set-oriented database update? A transaction?
- When are rules executed? At any time? Only after certain operations? Only at transaction boundaries?
- What happens if several rules are triggered at the same time? Are all rules executed? If so, is there an order? Is only one rule executed? If so, how is it chosen?
- What happens if execution of a rule causes another rule to trigger? How does the new rule interact with other triggered rules? Can a rule trigger itself?
- If rules are not always executed as soon as they are triggered, what environment is used when a rule is finally executed?

- If several rules are triggered simultaneously, what happens if execution of one rule's action negates another rule's condition?

Many existing proposals leave some of these issues unresolved or unclear, or suggest restrictive solutions. We provide a precise semantics that allows an easy understanding of rule behavior while remaining expressive and flexible.

Other proposals for production rules in database systems (e.g., [4, 5, 7, 13, 17]) consider *instance-oriented* rules: rules that are applied once for each data item satisfying the condition part of the rule. In contrast, we propose *set-oriented* rules: rules that are triggered by sets of changes to the database and may perform sets of changes. This approach conforms to the set-oriented approach of relational database languages. In particular, set-oriented query processing in relational database systems permits extensive optimization and efficient execution; the same technology is applicable to our production rules. Set-oriented execution may also allow rules to be combined with database operations or with other rules [8]. In some cases, instance-oriented rules can be compiled for set-oriented execution [16], but our direct approach avoids the complications and limitations arising from such a scheme. Finally, our set-oriented rules allow specification of some conditions and actions not expressible using instance-oriented rules.

In Section 2, we define the notion of "operation blocks"; these correspond to sequences of SQL **update**, **delete**, and **insert** operations. Operation blocks appear to execute atomically and form the basis of the database transitions that trigger our production rules. In Section 3, we give a syntax for production rule definition. The semantics of rule execution in the presence of externally-generated database transitions is defined in Section 4. As a first step towards implementation, Section 4 also includes an algorithm for rule execution that reflects the desired semantics. A number of potential extensions are proposed in Section 5. Section 6 contains discussion of future work.

2 Operation Blocks

In relational database systems, users (or application programs) submit streams of data manipulation operations for execution. We consider a model of system behavior in which these operations are grouped into *operation blocks*. We assume that operation blocks always finish and are executed indivisibly—we consider a level of abstraction in which multiple users, concurrent processing, and failures are all transparent. (The correspondence between operations blocks and database transactions is discussed in Section 4.) During execution of an operation block, data items (tuples) may be updated, deleted, or inserted; for concreteness, we let each block correspond to a non-empty sequence of SQL **update**, **delete**, and **insert** operations.[1] We consider operation blocks (rather than individual operations) for generality and to permit a formalism that adapts easily to most relational database languages. Using a variant of the SQL data manipulation language [10] and assuming some familiarity with SQL, a syntax for operation blocks is given as follows:

op-block ::= *sql-op* ; *sql-op* ; ... ; *sql-op*

sql-op ::= *update-op* | *delete-op* | *insert-op*

update-op ::= **update** *table*
set *columns* = *expressions*
where *predicate*

delete-op ::= **delete from** *table*
where *predicate*

insert-op ::= **insert into** *table*
values $\langle v_1, v_2, \ldots, v_n \rangle$

| **insert into** *table*
(*select-op*)

select-op ::= **select** *columns*
from *tables*
where *predicate*

The predicate in **update**, **delete**, and **select** operations may be arbitrarily complex and may include embedded **select** operations.

We assume a typical relational database structure [3]: a set of named *tables* is defined, each having a fixed set of named and typed *columns*.[2] In a given *state* of the database, each table contains zero or more *tuples*, where a tuple assigns a single value to each column of the table. Duplicate tuples may appear in a table. In subsequent discussion, we often identify certain (multi)sets of tuples—some are tuples in the database, while others are tuples that have existed in a previous state of the database but have since been deleted. Often, we assume that these sets are actually composed of system *tuple handles*—distinct, non-reusable tuple identifiers.

To define the semantics of operation blocks, we begin by describing the behavior of the SQL operations:

[1] For simplicity, we do not include **select** operations here since they leave the database state unchanged. A possible extension is to consider **select** operations here, which would allow rules to be triggered by data retrieval; see Section 5.

[2] In some cases the database schema may change over time. For simplicity, we assume a fixed schema. Also, in this paper we do not consider views.

- **update**: The tuples in the specified table satisfying the given predicate are identified. For each identified tuple, the expressions are evaluated and the results are assigned to the specified columns.
- **delete**: The tuples in the specified table satisfying the given predicate are identified. Each identified tuple is deleted from the table.
- **insert** with values: A new tuple containing values $\langle v_1, \ldots, v_n \rangle$ is added to the specified table.
- **insert** with select operation: The embedded select operation is evaluated producing a group of tuples. Each tuple is inserted into the specified table.

Thus, given a database state and an SQL operation, the operation either updates, deletes, or inserts zero or more tuples in a single table. A new database state results.

Now consider execution of an operation block. Each operation in the block transforms the database state. However, since operation blocks are executed indivisibly, we are interested only in the single state transformation that results from the entire sequence of operations. The net effect of execution of an operation block can be described by:

- the tuples that are inserted by the block—tuples that exist in the final state but not in the initial state;
- the tuples that are deleted by the block—tuples that exist in the initial state but not in the final state;
- the tuples that are updated by the block—tuples that exist in the initial and final states but whose values may have been changed by **update** operations in the block.

Thus, we associate with every operation block a triple, $[U, D, I]$, where U is a set of tuple handles identifying those tuples updated by the block, D is a set of tuple handles identifying those tuples deleted by the block, and I is a set of tuple handles identifying those tuples inserted by the block. (In the full paper [19], we formally define these sets based on a semantics given for the individual operations.) We refer to execution of a single operation block as a database *transition*, and we call the associated triple $[U, D, I]$ the *effect* of the transition.

Notice that a transition effect does not include a tuple's history through execution of the operation block. If a tuple is updated by several operations and then deleted, we note only the deletion, since this is the net effect of the transition. Similarly, multiple updates of a tuple are observably equivalent to a single update (since set U in transition effects contains tuple handles only), an insertion followed by an update is observably equivalent to inserting the updated tuple, and an insertion followed by a deletion is not visible at all. In contrast, we do not consider deletion of a tuple followed by insertion of a new tuple as equivalent to updating the original tuple, since tuple handles are not reused.

3 Rule Definition

Production rules are triggered by execution of operation blocks. We do not want every rule to be triggered by every block, however—we want rules to be triggered by specified operations on specified tables. Hence, we define the notion of *transition predicates*—predicates of the form "**updated** *table*", "**deleted from** *table*", and "**inserted into** *table*".[3] Transition predicate **updated t**, where **t** is a table name, is true with respect to a transition with effect $[U, D, I]$ if and only if set U contains one or more handles for tuples in table **t**; similarly for predicates **deleted from t** and **inserted into t**. In the full paper, we consider an extension in which transition predicates may be combined using boolean operators **and**, **or**, and **not**. Here, for simplicity, we restrict ourselves to base predicates. Thus, the syntax for transition predicates is:

trans-pred ::= **updated** *table*
| **deleted from** *table*
| **inserted into** *table*

We may also want to qualify execution of a triggered rule; that is, we may want to specify a condition that must hold if a triggered rule is to execute its action. Hence, in addition to specifying a transition predicate, an arbitrary SQL predicate may be included. For simplicity, we assume that the action specified in a production rule is an operation block. (As an extension, we might permit more general actions here, such as data retrieval, transaction abort, or arbitrary program execution; see Section 5.) Thus, we propose the following syntax for production rules:

prod-rule ::= **when** *trans-pred*
where *predicate*
then *op-block*

The **when** clause controls rule triggering. The **where** clause is the *condition part* of the rule. As in SQL operations, the predicate may be arbitrarily complex and may include embedded **select** operations; it may also be omitted, in which case the

[3] A straightforward extension is to also allow transition predicates of the form "**updated** *table.column*".

meaning is equivalent to including **where true**. The **then** clause is the *action part* of the rule, an operation block as defined in Section 2. Note that all three components of a rule are set-oriented constructs, as illustrated by the examples given below.

In externally-generated operations (operations issued by users or application programs), all references to database tuples refer to the "current" state—the contents of the tables at the time the operation begins execution. Since production rules are triggered by a state transition in which tuples are updated, inserted, and deleted, in a rule one may also want to refer to the following groups of tuples:

- the inserted tuples: the values of those tuples in the current state of the database that were inserted by the transition that triggered the rule;
- the deleted tuples: the values of those tuples in the previous state of the database that were deleted by the transition that triggered the rule;
- the old values of the updated tuples: the values of those tuples in the previous state of the database that were updated by the transition that triggered the rule;
- the new values of the updated tuples: the values of those tuples in the current state of the database that were updated by the transition that triggered the rule.

To permit this, we define an extended SQL syntax for **select** operations appearing in the condition and action parts of production rules. For referring to the tuples inserted by a transition, we add the keyword **inserted**,[4] which may precede a table name in the **from** clause of embedded **select** operations:

select *columns*
from ..., **inserted** *table* **tvar** , ...
where *predicate*

References to table variable **tvar** in the **select** and **where** clauses of this operation refer to the tuples in the specified table that were inserted by the transition triggering the rule. A table qualified with **inserted** need not be assigned a variable name as long as there is no conflict with references to the current value of the same table. Similarly, table names in embedded **select** operations may be preceded by **deleted**, **old updated**, or **new updated**, for referring to the deleted tuples, the old values of the updated tuples, or the new values of the updated tuples, respectively.

We call the logical table obtained by preceding a table name with **inserted**, **deleted**, **old updated**, or **new updated** a *transition table*. The transition table construct is quite powerful since, in a single transition, a number of operations may be performed on a number of different tables. However, we restrict use of transition tables to those tables relevant to the rule in which the construct appears: A rule of the form **when inserted into t...** (where **t** is a table name) may use only **inserted t**; a rule of the form **when deleted from t...** may use only **deleted t**; a rule of the form **when updated t...** may use only **old updated t** and **new updated t**. These restrictions are syntactic, therefore they are easily enforced.

[4] Although keyword **inserted** is also used in transition predicates, there should be no confusion.

3.1 Example

We give a simple example illustrating rule definition. Additional examples are included in [19]. Consider a database schema with two tables, **emp** and **dept**. Table **emp** maintains employee data, having columns for **emp_no** (which is unique), **salary**, and **dept_no**. Table **dept** maintains department data, having columns for **dept_no** and **mgr_no** (both are unique). Thus, we have:

emp(emp_no, salary, dept_no)
dept(dept_no, mgr_no)

Although our example is relatively unrealistic, it serves to illustrate important aspects of production rule definition. The rule states: Whenever employees are inserted, check the new average employee salary. If it exceeds the salary of the manager of department #5, give all inserted employees a 10% salary cut.

```
when inserted into emp
where avg(select salary from emp) >
        (select salary from emp e, dept d
         where e.emp_no = d.mgr_no
         and d.dept_no = 5)
then update emp
      set salary = 0.9 * salary
      where emp_no in
        (select emp_no from inserted emp)
```

4 Rule Execution

Production rules are activated automatically as a result of database state transitions caused by externally-generated operation blocks. Suppose a stream of operation blocks is submitted for execution (and production rules are not considered). We denote system behavior using the following graphical notation:

$$S_0 \xrightarrow[T_1]{\mathcal{E}_1} S_1 \xrightarrow[T_2]{\mathcal{E}_2} S_2 \xrightarrow[T_3]{\mathcal{E}_3} \cdots$$

S_0, S_1, S_2 ... denote database states while the arcs denote state transitions. $\mathcal{T}_1$, $\mathcal{T}_2$, $\mathcal{T}_3$ ... are unique transition labels; $\mathcal{E}_1$, $\mathcal{E}_2$, $\mathcal{E}_3$ ... denote the effects of the transitions. Recall that a transition effect $\mathcal{E}$ is actually a triple $[U, D, I]$. For notational convenience, we let $U(\mathcal{E})$, $D(\mathcal{E})$, and $I(\mathcal{E})$ denote the U, D, and I components of $\mathcal{E}$, respectively.

We assume that externally-generated operation blocks correspond to database transactions.[5] That is, each state in the execution sequence above corresponds to a state in which a transaction begins execution; the subsequent state corresponds to the point at which the user or application program requests that the transaction be committed. This one-to-one correspondence between transitions and transactions holds only for externally-generated operations. When rules are executed, a single transaction is composed of one externally-generated transition followed by some number of rule-generated transitions (that is, rules are considered and executed just before committing each externally-generated transaction). A detailed semantics follows.

4.1 A Single Rule

Suppose a single production rule R is defined, and consider a transition $\mathcal{T}$ with effect $\mathcal{E}$:

$$S \xrightarrow[\mathcal{T}]{\mathcal{E}} S'$$

We say that rule R is *triggered by transition* $\mathcal{T}$ if R's transition predicate is true with respect to $\mathcal{E}$ (as defined in Section 3). Triggering alone is not enough to initiate execution of the action part of a rule—the specified condition must also hold. Suppose rule R is triggered by transition $\mathcal{T}$. Recall that the condition part of a rule may refer to the current state of the database (state S'). Additionally, R's condition may refer to the logical transition tables introduced in Section 3. These tables are derived from transition effect $\mathcal{E}$ and from pre- and post-transition states S and S'; details are given in the full paper.

If R's condition holds, then, before execution of another externally-generated operation block, R's action is executed. As with evaluation of R's condition, execution of R's action may depend on the tables in state S' as well as on transition tables. Execution of R's action causes a new transition, which we label $\mathcal{T}_R$:

$$S \xrightarrow[\mathcal{T}]{\mathcal{E}} S' \xrightarrow[\mathcal{T}_R]{\mathcal{E}_R} S''$$

[5] As an extension, we might permit more flexibility with respect to transitions and transactions; see Section 5.

Since the action part of a rule is an operation block producing an effect and a new state, rule-generated transition $\mathcal{T}_R$ is indistinguishable from externally-generated transitions. Thus, we allow a rule-generated transition to trigger other rules, or even trigger the same rule again. Since we are restricting attention to a single rule, consider an example of the latter case. Suppose the triggering condition of rule R is an **update** to table **t** and, as its action, rule R further updates **t**. Transition $\mathcal{T}_R$ then corresponds to an operation block consisting of an **update t** ... operation. Assume that set $U(\mathcal{E}_R)$ is non-empty, i.e., at least one tuple is selected in R's update. Then rule R is triggered again by transition $\mathcal{T}_R$. If R's condition again holds, then R's action is executed a second time and we have:

$$S \xrightarrow[\mathcal{T}]{\mathcal{E}} S' \xrightarrow[\mathcal{T}_R]{\mathcal{E}_R} S'' \xrightarrow[\mathcal{T}'_R]{\mathcal{E}'_R} S'''$$

If $U(\mathcal{E}'_R)$ is non-empty, R is triggered again by transition $\mathcal{T}'_R$, and its condition may still hold. In fact, if R is always triggered by the transition caused by executing its action, and its condition always holds, then execution of R repeats indefinitely.[6]

4.2 Multiple Rules

Now consider multiple rules, so that more than one rule may be triggered by a given transition. Assuming we are not interested in a semantics based on parallel execution, the triggered rules must be activated in some order, taking into consideration the transitions created as rules are executed (which may also trigger new rules). We begin by considering in detail the first two rule-generated transitions following an initial externally-generated transition. We then generalize by describing system behavior at an arbitrary point during rule processing.

Let $\mathcal{T}_1$ be an externally-generated transition that triggers some number of rules $R_1, R_2, \ldots, R_n$. These are used to define a set ρ of "pending" triggered rules. Rules triggered by subsequent transitions may be added to this set (as described below), so we include with each rule in ρ the label of the transition that triggered the rule. Thus, the initial value of the set is $\rho_1 = \{(R_1, \mathcal{T}_1), (R_2, \mathcal{T}_1), \ldots, (R_n, \mathcal{T}_1)\}$. Including the set of triggered rules in our notation, we write:

[6] The potential for such behavior is similar to the potential for infinite loops or non-terminating concurrent programs. We might want to provide a static rule analysis facility that issues a warning when the possibility of divergence is detected; see Section 6. Run-time detection using a timeout mechanism could also be incorporated.

$$S_0 \xrightarrow[\mathcal{T}_1]{\mathcal{E}_1} S_1/\rho_1$$

The next step is to choose a rule for consideration from the set of triggered rules. We defer discussion of how a rule is chosen—several possibilities are discussed in Section 4.4 below—and assume that we have some rule selection method. Suppose a rule R_i is chosen. Element $(R_i, \mathcal{T}_1)$ is removed from set ρ_1 and R_i's condition is evaluated. If R_i's condition does not hold, then R_i's action is not executed and a new rule is chosen. Note that we do not return R_i to the set of triggered rules; once we have determined that a triggered rule's condition does not hold, that rule is no longer considered.

Suppose R_i's condition does hold. Then R_i's action is executed and we have:

$$S_0 \xrightarrow[\mathcal{T}_1]{\mathcal{E}_1} S_1/\rho_1 \xrightarrow[\mathcal{T}_2]{\mathcal{E}_2} S_2/\rho_2$$

The new set ρ_2 of triggered rules contains the remaining rule-transition pairs from ρ_1. Set ρ_2 must also reflect new rules triggered by transition $\mathcal{T}_2$. We defer discussion of newly triggered rules, first discussing rules carried over from set ρ_1.

Let R_j be a rule carried over from set ρ_1, and suppose that element $(R_j, \mathcal{T}_1)$ is now selected from set ρ_2 for consideration. Rule R_j may include references to transition tables and to the "current" database state. Immediately following transition $\mathcal{T}_1$, the current state was clearly S_1 and the transition tables were derived from transition effect $\mathcal{E}_1$ and states S_0 and S_1. However, execution of rule R_i's action (transition $\mathcal{T}_2$) may have altered the database state and may have affected tuples in R_j's transition tables. Thus, we must carefully define the environment in which R_j's condition is to be evaluated and, if the condition is true, the environment in which R_j's action is to be executed.

It seems evident that if R_j's action is executed, execution should proceed with S_2 as the current state—it is not sensible to perform an operation in an outdated database state. Then, since we do not want to execute a rule in a state in which its condition does not hold, the current state for evaluation of R_j's condition should also be S_2. The implication of this choice is that it is possible for R_j's condition to hold at the time the rule is triggered (in state S_1), but not hold by the time the condition is evaluated (in state S_2). Conversely, a triggered rule may have a condition that does not hold initially, but does hold by the time it is evaluated.

Now consider R_j's transition tables. We distinguish three cases: R_j is a **when inserted into** rule, R_j is a **when deleted from** rule, and R_j is a **when updated** rule.

Case 1: R_j = **when inserted into t ...**

After $\mathcal{T}_1$, transition table **inserted t** refers to those tuples of table **t** in state S_1 identified by tuple handles in $I(\mathcal{E}_1)$. However, these tuples may have been subsequently updated or deleted by transition $\mathcal{T}_2$. In keeping with the decision to evaluate and execute rule R_j in current state S_2, we use a current version of transition table **inserted t**: we do not include inserted tuples subsequently deleted by transition $\mathcal{T}_2$, and the values for the remaining inserted tuples are obtained from state S_2. (Recalling Section 2, note that this parallels the definition of a transition effect: updating an inserted tuple is equivalent to inserting the updated tuple, and an insertion followed by a deletion is not considered at all. Here, with respect to rule R_j, we are effectively treating $\mathcal{T}_1$ and $\mathcal{T}_2$ as a single transition taking state S_0 to state S_2.) Suppose also that during transition $\mathcal{T}_2$, additional tuples are inserted into table **t**. There are two possibilities here: either rule R_j can be triggered a second time by transition $\mathcal{T}_2$, or the newly inserted tuples can be added to R_j's transition table. In the spirit of treating $\mathcal{T}_1$ and $\mathcal{T}_2$ as a single transition with respect to rule R_j, we choose the latter option. Thus, in rule R_j, **inserted t** now also includes those tuples of table **t** in state S_2 identified by tuple handles in $I(\mathcal{E}_2)$.

Case 2: R_j = **when deleted from t ...**

Tuples in transition table **deleted t** could not have been updated or deleted by transition $\mathcal{T}_2$, since they did not exist in state S_1. However, additional tuples could have been deleted from table **t** during transition $\mathcal{T}_2$. Again treating $\mathcal{T}_1$ and $\mathcal{T}_2$ as a single transition with respect to rule R_j, we want to include these tuples in **deleted t**, but only if they existed in state S_0; that is, we do not want to add deleted tuples that were inserted during (or following) the transition that triggered the rule. Thus, we add to R_j's **deleted t** those tuples of table **t** in state S_0 whose tuple handles are in $D(\mathcal{E}_2)$ but are not in $I(\mathcal{E}_1)$.

Case 3: R_j = **when updated t ...**

Tuples updated by transition $\mathcal{T}_1$ may have been deleted or further updated by transition $\mathcal{T}_2$. To reflect deletions, we do not include in transition tables **old updated t** or **new updated t** any tuples whose handles are in $D(\mathcal{E}_2)$. To reflect multiple updates, the values for the tuples in **new updated t** are obtained from current state S_2. Now consider the newly updated tuples—tuples in table **t** that were updated by transition $\mathcal{T}_2$ but were not updated by transition $\mathcal{T}_1$. We want to include these tuples in **old updated t** and **new updated t** only if they existed in state S_0. Thus, we add to **old updated t**

(respectively **new updated t**) those tuples of table **t** in state S_0 (respectively state S_2) whose tuple handles are in $U(\mathcal{E}_2)$ but are not in $U(\mathcal{E}_1)$ or $I(\mathcal{E}_1)$.

We have now fully described how any rule triggered by transition $\mathcal{T}_1$ may be considered for execution following transition $\mathcal{T}_2$. Set ρ_2 of triggered rules must also include those rules newly triggered by transition $\mathcal{T}_2$. As discussed above, rules that were triggered by transition $\mathcal{T}_1$ but have not yet been considered cannot be triggered a second time by transition $\mathcal{T}_2$; rather, additional changes are accumulated in the transition tables of the appropriate pending rules. Hence, after transition $\mathcal{T}_2$, set ρ_2 contains all rule-transition pairs remaining from set ρ_1, along with all pairs $(R_k, \mathcal{T}_2)$ such that rule R_k's transition predicate is satisfied by transition effect $\mathcal{E}_2$, and $(R_k, \mathcal{T}_1)$ is not already in the set.

The generalization of this semantics is straightforward. Consider an arbitrary state S_n resulting from execution of a user-generated operation block followed by some number of rules:

$$S_0 \xrightarrow[\mathcal{T}_1]{\mathcal{E}_1} S_1/\rho_1 \xrightarrow[\mathcal{T}_2]{\mathcal{E}_2} \cdots \xrightarrow[\mathcal{T}_n]{\mathcal{E}_n} S_n/\rho_n$$

Set ρ_n of triggered rules contains elements of the form $(R, \mathcal{T}_x)$, $1 \leq x \leq n$, where transition $\mathcal{T}_x$ has triggered rule R and this pair has not yet been selected for consideration. An element $(R, \mathcal{T}_x)$ is chosen and removed from the set. R's condition is checked, with S_n as the current database state. If R contains references to transition tables, these tables are evaluated as follows (for any table **t**):

- **inserted t** contains all tuples of table **t** in state S_n whose handle appears in some $I(\mathcal{E}_y)$, $x \leq y \leq n$, but whose handle does not appear in any $D(\mathcal{E}_y)$, $x \leq y \leq n$;
- **deleted t** contains all tuples of table **t** in state S_{x-1} whose handle appears in some $D(\mathcal{E}_y)$, $x \leq y \leq n$, but whose handle does not appear in any $I(\mathcal{E}_y)$, $x \leq y \leq n$;
- **old updated t** contains all tuples of table **t** in state S_{x-1} whose handle appears in some $U(\mathcal{E}_y)$, $x \leq y \leq n$, but whose handle does not appear in any $I(\mathcal{E}_y)$ or $D(\mathcal{E}_y)$, $x \leq y \leq n$;
- **new updated t** contains all tuples of table **t** in state S_n whose handle appears in some $U(\mathcal{E}_y)$, $x \leq y \leq n$, but whose handle does not appear in any $I(\mathcal{E}_y)$ or $D(\mathcal{E}_y)$, $x \leq y \leq n$;

Again, note the close correspondence between these definitions and the definition of transition effects (Section 2). Rule R effectively treats its triggering transition and all subsequent transitions until the time it is considered (i.e., transitions $\mathcal{T}_x, \ldots, \mathcal{T}_n$) as if they were a single triggering transition.[7] After R's condition is checked, if the condition is found to be false, then a new rule-transition pair is chosen from set ρ_n. (If ρ_n is empty then there are no rules left to consider and the next user-generated operation block may be executed.) If R's condition holds, then its action is executed (with respect to current state S_n and the transition tables as described above), creating the next transition. Rules triggered by the new transition are added to the set of triggered rules if they are not already in the set, and a new rule is selected for consideration.

4.3 Rule Execution Algorithm

We give an algorithm that reflects our semantics of rule execution. We do not assume access to all previous states and transition effects. Rather, we maintain for each transition (starting with the most recent externally-generated transition) enough information to construct transition tables for any rule triggered by that transition; this information is modified incrementally as operation blocks are executed and new transitions are created. Many aspects of the algorithm are simplified considerably from an implementation, however. For example, the entire database state does not need to be saved before each transition, and, instead of automatically maintaining all transition table information for each transition, we might consider which rules have actually been triggered.

The algorithm is given in Fig. 1. It loops indefinitely as long as there are externally-generated operation blocks to be executed. Each iteration of the loop corresponds to a single transaction containing an externally-generated transition followed by consideration and possible execution of some number of rules. Variable *trans-list* is used to maintain an ordered list of transitions (represented by their unique labels), starting with the most recent externally-generated transition. Other variables are self-explanatory. We assume the existence of the following functions:

- *current-state*() returns the current state of the database;
- *execute-external-block*() executes the next externally-generated operation block and returns a pair containing a unique label for the created transition and the transition effect;

[7] One exception to this is that a **when inserted into** rule remains triggered even if all inserted tuples are deleted before the rule is considered; similarly for a **when updated** rule. For consistency, we could detect such situations and "untrigger" the relevant rules.

```
repeat
  old-state ← current-state();
  (T, E) ← execute-external-block();
  ρ ← ∅;
  for each R ∈ triggered(E) do
    ρ ← ρ ∪ {(R, T)}
  end for;
  init-trans-tables(T, E, old-state);
  trans-list ← ⟨T⟩;
  while ρ ≠ ∅ do
    repeat
      (R, T) ← select-rule(ρ);
      ρ ← ρ − {(R, T)};
      cond-holds ← check-condition(R, T)
    until cond-holds or ρ = ∅;
    if cond-holds then
      old-state ← current-state();
      (T, E) ← execute-rule-block(R, T);
      for each R ∈ triggered(E) do
        if R does not appear in ρ then
          ρ ← ρ ∪ {(R, T)}
        end if
      end for;
      modify-trans-tables(trans-list, E, old-state);
      init-trans-tables(T, E, old-state);
      trans-list ← append(trans-list, T)
    end if
  end while
forever
```

Figure 1: Rule Execution Algorithm

- *triggered*($\mathcal{E}$) returns the set of rules whose transition predicate is satisfied by effect $\mathcal{E}$;
- *select-rule*(ρ) returns one member of set ρ of rule-transition pairs;
- *check-condition*(R, T) returns *true* if rule R's condition holds with respect to the current database state and the current version of T's transition tables, and returns *false* otherwise;
- *execute-rule-block*(R, T) executes rule R's action with respect to the current database state and the current version of T's transition tables, and returns a pair containing a unique label for the created transition and the transition effect.

Two procedures are called; in the full paper, algorithms are given for both:

- *init-trans-tables*($T, \mathcal{E}$, *old-state*) initializes transition table information for the transition labeled T. A mapping is established from T to three sets: a set *inserted* containing tuple handles for those tuples inserted by T, a set *deleted* containing values for those tuples deleted by T, and a set *updated* containing handle-value pairs for those tuples updated by T. Handles are used to obtain new values while old values are kept explicitly.
- *modify-trans-tables*(*trans-list*, $\mathcal{E}$, *old-state*) modifies transition table information for each transition in *trans-list*, based on a new transition with effect $\mathcal{E}$ and pre-transition state *old-state*. For each T in *trans-list*, sets *inserted*, *deleted*, and *updated* are modified as described in Section 4.2.

4.4 Rule Selection

We have left unspecified the method for choosing a rule for consideration when the set of pending triggered rules has more than one element. (This is similar to *conflict resolution* in OPS, which has received considerable attention [1].) A number of strategies are possible. Rules could be chosen arbitrarily, but such purely non-deterministic behavior is probably undesirable—in many cases it is useful or even necessary to have some degree of control over rule selection. The rules could be totally ordered, in which case the triggered rule highest in the ordering is chosen. We want the flexibility of allowing rules to be defined independently, however, in which case a total ordering may not be possible. A compromise is to partially order the rules, in which case a rule is chosen such that no other triggered rule is strictly higher in the ordering. In some cases we may also want to take into consideration the age of the triggering transition, i.e., preferring those rules triggered least recently or those triggered most recently. For a thorough comparison and evaluation of rule selection strategies we must consider a number of large-scale examples. Further discussion of rule selection is included in [19].

4.5 Example

A simple example is given to illustrate the semantics of rule execution. We continue with the database schema defined in Section 3.1. Our example illustrates the cascaded delete method of enforcing referential integrity:[8] Whenever managers are deleted, all employees in the departments managed by the deleted employees are also deleted, along with the departments themselves. (We assume a hierarchical structure of employees and departments. We also

[8] *Referential integrity* is any constraint of the form "every child must have a parent". The *cascaded delete* method of enforcement requires that whenever a parent is deleted, so are all of its children.

assume that employee numbers are not immediately reused.)

```
when deleted from emp
then begin
  delete from emp
  where dept_no in
    (select dept_no from dept
     where mgr_no in
       (select emp_no from deleted dept);
  delete from dept
  where mgr_no in
    (select emp_no from deleted emp)
end
```

The self-triggering property of this rule under our semantics correctly reflects the recursive nature of a cascaded delete.

Additional examples (involving multiple rules) are given in [19].

5 Extensions

For clarity and simplicity, in many cases we have omitted features that we may choose to include in an implementation. In Section 3, two extensions to our notion of transition predicates were suggested: first, incorporating predicates of the form **updated t.c** (where **c** names a column of table **t**), which permits more discriminatory triggering of **when updated** rules, and second, combining base transition predicates using boolean operators. Here, we briefly discuss other potential extensions.

We have considered **select** as an embedded operation only. Intuitively, it seems that most production rules would be triggered by changes to the database and would perform changes, such as rules used to enforce integrity constraints. However, we may want the action part of a rule to include data retrieval; for example, we might want to define a rule that automatically delivers a summary of employee data whenever salaries are updated. In some cases, we may also want to define rules that are triggered by data retrieval; this is a necessary feature, for example, if rules are to be used for authorization checking.[9] Both features can be added by modifying the definition of an operation block to include *select-op* as a top-level *sql-op* (recall Section 2). We then add a transition predicate of the form "**selected** *table*" (or, perhaps, "**selected** *table.column*"), and add to transition effects an S component containing handles for those tuples that have been selected during the transition. There are a few issues left to be addressed, such as whether tuples from embedded **select** operations are included in $S(\mathcal{E})$, and how transition tables based on **select** operations are affected by subsequent transitions. In general, however, it should not be difficult to incorporate data retrieval into our framework.

We have specified that, as a single transaction, the system executes an externally-generated operation block (corresponding to an externally-defined transaction) followed by all resulting rule-generated operation blocks. Although this is a reasonable choice in many situations, we might want additional flexibility in the time at which rules are triggered and in the correspondence between rules and transactions. (See [2, 13] for further discussion of these issues.) For example, we might want the ability to specify that a rule's action should be executed in a separate transaction. Also, in some cases, it might be advantageous to execute several externally-generated transactions before considering triggered rules, or, conversely, we might prefer to consider rules earlier than the commit point of an externally-generated transaction.[10] One way to achieve this is to allow user-defined "rule triggering points": When a rule triggering point is reached, the externally-generated transition is considered complete, rules are processed, and a new transition begins. In this case, the correspondence between transitions and transactions must be carefully defined.

The action parts of our production rules are limited to SQL data manipulation operations. This permits a simple and clean formalism. However, just as database applications generally include more than simple data manipulation operations (they may include sections of code in a host programming language, commands relating to transactions, etc.) it would be useful to allow the same flexibility in the action part of production rules. As a first step, a simple and certainly useful extension is to allow transaction abort commands to be issued by rules. (Adding such a facility to the semantics described in this paper is straightforward.) Note that if we allow more flexibility in the action part of production rules, we would also want to extend our semantics to accommodate errors or exception conditions that may arise during rule execution.

Lastly, as database systems are extended to support features such as real-time processing and references to arbitrary versions, we want our rules to support these features as well.

6 Future Work

We are implementing a set-oriented production rules facility following the syntax and semantics proposed

[9] Authorization checking also requires a facility for considering rules before (rather than after) database operations.

[10] The latter case is necessary to simulate constraint enforcement in existing systems [10].

in this paper. The facility is being incorporated into the Starburst extensible database system [9, 12] at the IBM Almaden Research Center. Concurrently, we are investigating automatic analysis of set-oriented production rules. Viewed as a programming language, our production rules facility is very expressive but very unstructured. Rather than limiting expressiveness, we want to provide tools that support the database production rules programmer. In particular, the programmer might benefit from knowing that a set of rules may create an infinite loop, or from knowing that ordering between certain rules may affect the final database state. We envision a facility that uses analysis techniques when production rules are defined to issue warnings of potential loops and conflicts. Finally, we suspect that a large class of production rules fall into the category of constraint-maintaining rules. In many cases, it may be possible to take a description of a constraint (at some level) and automatically refine it into a set of production rules that maintain the constraint. (Such a facility could also be viewed as a high-level structuring mechanism for a certain class of rules.) We are exploring the feasibility of this approach.

Acknowledgements

Hamid Pirahesh and Manolis Tsangaris pursued initial work in this area [8]. Bruce Lindsay suggested modifications to the semantics that should facilitate the implementation.

References

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

[2] M.J. Carey, R. Jauhari, and M. Livny. *On Transaction Boundaries in Active Databases: A Performance Perspective.* Technical Report 796, Computer Sciences Dept., Univ. of Wisconsin, November 1988.

[3] E.F. Codd. A relational model for large shared data banks. *CACM*, 13(6):377–387, June 1970.

[4] D. Cohen. Compiling complex database transition triggers. In *Proc. ACM SIGMOD Conference*, pages 225–234, Portland, Oregon, May 1989.

[5] C. de Maindreville and E. Simon. A production rule based approach to deductive databases. In *Proc. Fourth Conference on Data Engineering*, pages 234–241, Los Angeles, California, February 1988.

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

[7] K.P. Eswaran. *Specifications, Implementations and Interactions of a Trigger Subsystem in an Integrated Database System.* IBM Research Report RJ1820, IBM San Jose Research Laboratory, August 1976.

[8] S.J. Finkelstein, H. Pirahesh, and M. Tsangaris. Rule support for Starburst. 1988. Unpublished notes.

[9] L.M. Haas, J.C. Freytag, G.M. Lohman, and H. Pirahesh. Extensible query processing in Starburst. In *Proc. ACM SIGMOD Conference*, Portland, Oregon, May 1989.

[10] *IBM Systems Application Architecture, Common Programming Interface: Data-base Reference.* IBM Form Number SC26-4348-1, October 1988.

[11] A.M. Kotz, K.R. Dittrich, and J.A. Mulle. Supporting semantic rules by a generalized event/trigger mechanism. In *Advances in Database Technology—EDBT '88, Lecture Notes in Computer Science 303*, pages 76–91, Springer-Verlag, Berlin, March 1988.

[12] B. Lindsay, J. McPherson, and H. Pirahesh. A data management extension architecture. In *Proc. ACM SIGMOD Conference*, pages 220–226, San Francisco, California, May 1987.

[13] D.R. McCarthy and U. Dayal. The architecture of an active database management system. In *Proc. ACM SIGMOD Conference*, pages 215–224, Portland, Oregon, May 1989.

[14] L. Raschid and S.Y.W. Su. A transaction oriented mechanism to control processing in a knowledge base management system. In *Expert Database Systems—Proceedings from the Second International Conference*, pages 353–373, Benjamin/Cummings, Redwood City, California, 1989.

[15] T. Sellis, C.-C. Lin, and L. Raschid. Implementing large production systems in a DBMS environment: concepts and algorithms. In *Proc. ACM SIGMOD Conference*, pages 404–412, Chicago, Illinois, June 1988.

[16] E. Simon and C. de Maindreville. Deciding whether a production rule is relational computable. In *Proc. International Conference on Database Theory*, Bruges, Belgium, September 1988.

[17] M. Stonebraker, E. Hanson, and S. Potamianos. A rule manager for relational database systems. In *The POSTGRES Papers*, pages 47–68, Memorandum No. UCB/ERL M86/85 (revised), Electronics Research Laboratory, University of California, Berkeley, June 1987.

[18] A. Tzvieli. On the coupling of a production system shell and a DBMS. In *Proc. Third Conference on Data and Knowledge Bases*, pages 291–309, Jerusalem, Israel, June 1988.

[19] J. Widom and S.J. Finkelstein. *A Syntax and Semantics for Set-Oriented Production Rules in Relational Database Systems.* IBM Research Report, IBM Almaden Research Center, June 1989.