The Nonmonotonic Semantics of Active Rules in Deductive Databases

Carlo Zaniolo
University of California,
Los Angeles, CA., 90024,
zaniolo@cs.ucla.edu

Abstract

We provide a logic-based semantics for active database rules under the immediately-after, tuple-level activation policy of SQL3. This result allows us to characterize and analyze the behavior of complex rule systems using the declarative framework of stable models, and to lay the theoretical foundation for a very desirable integration of active databases and deductive databases.

1 Introduction

Deductive database research has long recognized, and struggled to resolve, two related problem areas pertaining to the formal semantics of logic-oriented, rule-based systems. The first problem is how to move beyond stratified negation, while preserving computational tractability; the second problem is how to provide simple declarative semantics for updates, and, then, active rules. In fact, there has been much progress on the first front but little on the second front. Indeed, the introduction of concepts such as well-founded models [6] and stable models [4] have established a formal and comprehensive framework for the formal semantics of programs with nonstratified negation. Moreover, various subclasses of these programs that are computationally well-behaved were identified and implemented. For instance, current database prototypes support the computation of well-founded models for various classes of non-stratified programs, which include modularly stratified programs [11, 10], XY-stratification [12], and the closely-related notion of explicitly locally stratified programs [3, 9]. Furthermore, the notion of choice models [7] takes logic-based semantics beyond the deterministic realm of well-founded models; in fact it identifies a class of programs that always have total stable models of polynomial data complexity, and captures the 'don't care' manifestation of nondeterminism from stable models.

On the other hand, advances on the second front have been less significant and have so far failed to have a major impact on actual prototypes. Although different systems attempted diverse approaches and original solutions to updates, in the end, they all settled for expedient combinations of declarative semantics and operational update constructs. (This is exemplified by the Glue/Nail! system where the operational Glue layer was wrapped around the original Datalog layer, featuring compatible data types but no unified semantics [5].)

The framework of transaction logic [2] offers an approach that is appealing at the theoretical level, but its orientation toward top-down semantics, and the computational complexities, which come with its great power and flexibility, make its adoption in the basic framework of bottom-up deductive systems somewhat problematic.

The main contribution of this paper is to show that a simple and general solution of the second problem follows naturally from the theoretical and practical advances achieved and implemented on the first front. Therefore, we proceed as follows. In the next section, we recall the concept of XY-stratified programs [12], which through their explicit structure of temporal arguments ensure an efficient compilation and computation for their total well-founded models. Then, we develop a simple framework to model updates and active rules, using a revised view of intensional versus extensional information. In the current theory of deductive databases, the database supplies the extensional information an the deductive rules provide the intensional information. Here, the *database history* becomes our extensional information, and both deductive and active rules define intensional information.

Then, to closely model SQL3 active rules, we include the concept of nondeterminism in our XY-stratified programs, leading to the class of programs called *XY-Stratified Choice Programs*, which have a semantics defined by multiple (total) stable models. This allow us to create a faithful model for SQL3 rules under the immediately-after, for-each-tuple activation semantics [14].

2 Non-Stratified Programs

The main practical limitation of semantics based on concepts such as well-founded models is that there is no general way to decide whether a program obeys such a semantics, short of executing the program. This is in sharp contrast with the concept of stratified negation, where a predicate dependency graph free of cycles provides a simple criterion for the programmer to follow in writing the program, and for the compiler to check and use in validating and optimizing the execution of the program. We will next discuss a particular mixture of Datalog_{1S} and stratification that has great expressive power but preserves much of the simplicity of stratified programs.

For instance, the ancestors of `marc`, and the number of generations that separates them from `marc`, can be computed using the following program, which also includes the differential fixpoint improvement:

Example 1 *Computing the ancestors of* `marc` *and the generation gap using the differential fixpoint improvement*

$$
\begin{array}{ll}
r_1 : \texttt{delta\_anc(0,marc)}. & \\
r_2 : \texttt{delta\_anc(J+1,Y)} \leftarrow & \texttt{delta\_anc(J,X),parent(Y,X)}, \\
 & \neg\texttt{all\_anc(J,Y)}. \\
r_3 : \texttt{all\_anc(J+1,X)} \leftarrow & \texttt{all\_anc(J,X)}. \\
r_4 : \texttt{all\_anc(J,X)} \leftarrow & \texttt{delta\_anc(J,X)}.
\end{array}
$$

This program is locally stratified by the first argument in `anc` that serves as temporal argument. The bottom stratum, i.e., the stratum at level 0, consists of atoms of nonrecursive predicates such as `parent` and of atoms that satisfy the goal `all_anc(0,_)` or `delta_anc(0,_)`. Likewise, the k^{th} stratum consists of atoms of the form `all_anc(k,_)`, `delta_anc(k,_)`. Thus, this program is locally stratified, since the heads of recursive rules belong to strata that are one above those of their goals.

So far, we have studied the case where all recursive atoms with the same temporal argument belong to the same stratum. This structure can be generalized by partitioning atoms with the same temporal argument into multiple substrata. For instance, in Example 1, a strict stratification would place $\mathtt{delta\_anc}(\mathtt{k}, \_)$ in a stratum lower than $\mathtt{all\_anc}(\mathtt{k}, \_)$. In fact, if the goal $\neg\mathtt{delta\_anc}(\mathtt{J}+\mathtt{1}, \_)$ is added to the third rule (this will not change the meaning of the program), having the former atoms in a lower stratum becomes necessary to preserve local stratification.

From these examples, therefore, we can now describe the syntactic structure of our programs as follows:

Definition 1 *XY-programs. Let P be a set of rules defining mutually recursive predicates. Then we say that P is an XY-program if it satisfies the following conditions:*

1. *Every recursive predicate of P has a distinguished temporal argument.*
2. *Every recursive rule r is either an X-rule or a Y-rule, where*
   - *r is an X-rule when the temporal argument in every recursive predicate in r is the same variable (called the temporal variable for r),*
   - *r is a Y-rule when (i) the head of r has as temporal argument $J+1$ (where J is called the temporal variable of r), (ii) some goal of r has as temporal argument J, and (iii) the remaining recursive goals have either J or $J+1$ as their temporal arguments.*

For instance, the program in Example 1 is an XY-program where r_4 is an X-rule with temporal variable `J`, while r_2 and r_3 are Y-rules, each with temporal variable `J`.

Therefore, exit rules establish initial values for the temporal argument; then the X-rules are used for reasoning within the same state (i.e., the same value of temporal argument) while the Y-rules are used for reasoning from one state to the successor state.

There is a simple test to decide whether an XY-program P is locally stratified. The test begins by labeling the recursive predicates in P to yield the *bistate program* P_{bis}, computed as described next.

Procedure 1 Computation of P_{bis}.

For each $r \in P$ do:

1. *Rename all the recursive predicates in r that have the same temporal argument as the head of r with the distinguished prefix* `new_` *.*
2. *Rename all other occurrences of recursive predicates in r with the distinguished prefix* `old_`*.*
3. *Drop the temporal arguments from the recursive predicates.*

For instance, the bistate version for the program in Example 1 is as follows:

Example 2 *The bistate version of the program in Example 1*

```
new_delta_anc(marc).
new_delta_anc(Y) ←    old_delta_anc(X), parent(Y, X),
                      ¬old_all_anc(Y).
new_all_anc(X) ←      old_all_anc(X).
new_all_anc(X) ←      new_delta_anc(X).
```

Definition 2 *Let P be an XY-program. P is said to be XY-stratified when P_{bis} is a stratified program.*

The program of Example 2 is stratified with the following strata: S_0 = {parent, old_all_anc, old_delta_anc}, S_1 = {new_delta_anc}, and S_2 = {new_all_anc}. Thus, the program in Example 1 is locally stratified.

Theorem 1 *Let P be an XY-stratified program. Then P is locally stratified.*

Let Σ be a stratification of P_{bis} in $n+1$ strata numbered from 0 to n, where we can assume, without loss of generality, that if p is a recursive predicate, then old_p along with every nonrecursive predicate belongs to stratum 0. Then, a local stratification for P can be constructed on the basis of the temporal argument of the recursive predicates.

In our temporal arguments, we used the postfix function symbol $+1$ as the temporal successor functor. Then, we used $x+n$ to denote the result of applying $+1$ to x n times; $x+0$ denotes x. If $y = x+n$, and there exists no term z for which $x = z+1$, then we will say that n is the *temporal rank* of y. For instance, a and $a+3$ share the same temporal rank, zero, while $g(a+3)+2$ and $a+2$ share temporal rank 2. Two atoms are temporally congruent if they share the same predicate name and their temporal arguments share the same temporal rank. Let us now denote by $\tau_P(y)$ the set of atoms in H_P temporally congruent with a given atom y.

Let q be recursive predicate in an XY-stratified program, P, and $a_q \in B_P$ be an atom with predicate name q and temporal argument t_q. Also, say that new_q belongs to the k^{th} stratum in the stratification of P_{bis}. Then, the local stratification for P is constructed by assigning a_q to the stratum $j \times n+k$, where j denotes the temporal rank of t_q; all the atoms that are temporally congruent with a_q are also assigned to the same stratum. All atoms of nonrecursive predicates are assigned to stratum 0. Then, by construction, the head of every rule $r \in ground(P)$ belongs to a stratum that is higher than the strata containing positive goals of r and strictly higher than the strata containing negated goals of r. Thus, the program of Example 1 is locally stratified with strata

$$
\begin{array}{rcl}
S_0 & = & \texttt{parent}(\_,\_) \\
S_1 & = & \tau_P(\texttt{delta\_anc}(0,\_)) \\
S_2 & = & \tau_P(\texttt{all\_anc}(0,\_)) \\
S_3 & = & \tau_P(\texttt{delta\_anc}(1,\_)) \\
S_4 & = & \tau_P(\texttt{all\_anc}(1,\_)) \\
 & \ldots & \\
S_{j\times 2+1} & = & \tau_P(\texttt{delta\_anc}(j,\_)) \\
S_{j\times 2+2} & = & \tau_P(\texttt{all\_anc}(j,\_))
\end{array}
$$

For an XY-stratified program P, the iterated fixpoint computation [14] becomes quite simple; basically it reduces to a repeated computation over the stratified program P_{bis}. However, since the temporal arguments have been removed from this program, we need to (1) store the temporal argument as an external fact `counter(T)`, and (2) add a new goal $\mathtt{counter(I_r)}$ to each exit rule r in P_{bis}, where $\mathtt{I}_r$ is the temporal argument in the original rule r. The program so constructed will be called the *synchronized version* of P_{bis}. For instance, to obtain the synchronized version of the program in Example 2, we need to change the first rule to

$$\mathtt{new\_delta\_anc(marc)} \leftarrow \quad \mathtt{counter(0)}.$$

since the temporal argument in the original exit rule was the constant 0.

The iterated fixpoint computation provides a general approach to the computation of the well-founded model for a locally stratified program. For an XY-stratified program P, this computation specializes into the following algorithm [14]:

Procedure 2 *Computing the well-founded model of an XY-stratified program*

Initialize: *Set $T = 0$ and insert the fact* `counter(T)`.

Forever repeat the following two steps:

1. *Apply the iterated fixpoint computation to the synchronized program P_{bis}, and for each recursive predicate* `q`*, compute* `new_q`*. Return the* `new_q` *atoms so computed, after adding a temporal argument* `T` *to these atoms; the value of* `T` *is taken from* `counter(T)`.
2. *For each recursive predicate* `q`*, replace* `old_q` *with* `new_q`*, computed in the previous step. Then, update the fact* `counter(T)` *to* $\mathtt{counter(T+1)}$.

Thus, for Example 1, the goal `counter(0)` ensures that the exit rule only fires once immediately after the initialization step[1].

For practical applications, we need simple sufficient conditions that allow us to stop the computation. In particular, we want to stop the computation as soon as we determine that all the recursive relations computed after the current step are empty. Therefore, it is easy to show that, for XY-stratified programs where each rule contains some recursive positive goal, the computation can be stopped at step `J` where the following two conditions hold: (i) the exit rules cannot produce any new atoms for temporal argument values greater than `J`, and (ii) for each recursive rule, there is a positive goal, say, `q`, for which no `new_q` atoms were obtained at the last step. For Example 1, the temporal argument in the exit rule is a constant and condition (i) is satisfied at every point after that. However, the third rule fails condition (ii) forever, causing an infinite computation. Thus, the programmer should add the goal `delta_anc(J,_)` to this rule. Once no new arcs are found, then because of the negated goal in the second rule, there is no `new_delta_anc(J,_)` atom and the computation stops.

[1]However, if we have exit rules where the temporal argument is a constant greater than zero, or it is not a constant, then the exit rules might produce results at later steps, too.

The computation of Procedure 2 can be made very efficient by some simple improvements. The first improvement consists in observing that the replacement of `old_q` with `new_q` described in the last step of the procedure can become a zero-cost operation if properly implemented (e.g., by switching the reference pointers to the two relations). A second improvement concerns copy rules such as rule r_3 in Example 1. Observe that the body and the head of this rule are identical, except for the prefixes `new` or `old`, in its bistate version (Example 2). Thus, we can let the same underlying relation supporta both the 'old' version and 'new' version of `all_anc`; then, to compute `new_all_anc`, we execute the copy rule first, by simply resetting the pointer from `new_all_anc` to `old_all_anc`—a zero-cost operation. Then, we execute the other rules defining `new_all_anc`—i.e., the last rule, in Example 2.

In the next example, we use XY-stratified programs to compute temporal projections [14]. Say, for instance, that we have a temporal relation as follows:

$$\texttt{emp\_dep\_sal}(\texttt{Eno}, \texttt{Dept}, \texttt{Sal}, \texttt{From}, \texttt{To}).$$

Now, say that

```
emp_dep_sal(1001, shoe, 35000, 19920101, 19940101).
emp_dep_sal(1001, shoe, 36500, 19940101, 19960101).
```

represent two tuples from this relation. The first fact denotes that employee with `Eno = 1001` has kept the same salary ($35,000) and department (`shoe`) from 1992/01/01 (year/month/day) till 1994/01/01[2]. According to the second fact, this employee, still in the `shoe` department, then received a salary of $36,500 from 1994/01/01 till 1996/01/01. If we now project out the salary and department information, these two intervals must be merged together. In this example, we have intervals overlapping over their endpoints. In more general situations, where a temporal projection eliminates some key attributes, we might have intervals overlapping over several time granules.

We now define the auxiliary predicates `distinct` and `select_larger`. The first verifies that two intervals are not the same interval. The second selects the larger of a pair of values.

```
distinct(Frm1, To1, Frm2, To2) ←   To1 ≠ To2.
distinct(Frm1, To1, Frm2, To2) ←   Frm1 ≠ Frm2.

select_larger(X, Y, X) ←      X ≥ Y.
select_larger(X, Y, Y) ←      Y > X.
```

Then, we have the program of Example 3, which iterates over two basic computation steps. The first step is defined by the `overlap` rule. This determines pairs of distinct overlapping intervals, where the first interval precedes (i.e., contains the start) of the second interval. The second step consists of deriving a new interval that begins at the start of the first interval, and ends at the later of the two endpoints. Finally, there is a copy rule that copies those intervals that do not overlap other intervals.

[2] In a Datalog system that supports temporal data types those should be used instead of this crude representation.

Example 3 *Merging overlapping periods into maximal periods after a temporal projection*

$$
\begin{array}{ll}
\texttt{e\_hist(0, Eno, Frm, To)} \leftarrow & \texttt{emp\_dep\_sal(0, Eno, D, S, Frm, To)}. \\[6pt]
\texttt{overlap(J + 1, Eno, Frm1, To1, Frm2, To2)} \leftarrow & \\
 & \texttt{e\_hist(J, Eno, Frm1, To1)}, \\
 & \texttt{e\_hist(J, Eno, Frm2, To2)}, \\
 & \texttt{Frm1} \leq \texttt{Frm2}, \texttt{Frm2} \leq \texttt{To1}, \\
 & \texttt{distinct(Frm1, To1, Frm2, To2)}. \\[6pt]
\texttt{e\_hist(J, Eno, Frm1, To)} \leftarrow & \texttt{overlap(J, Eno, Frm1, To1, Frm2, To2)}, \\
 & \texttt{select\_larger(To1, To2, To)}. \\
\texttt{e\_hist(J + 1, Eno, Frm, To)} \leftarrow & \texttt{e\_hist(J, Eno, Frm, To)}, \\
 & \texttt{overlap(J + 1, \_, \_, \_, \_, \_)}, \\
 & \neg\texttt{overlap(J + 1, Eno, Frm, To, \_, \_)}, \\
 & \neg\texttt{overlap(J + 1, Eno, \_, \_, Frm, To)}. \\[12pt]
\texttt{final\_e\_hist(Eno, Frm, To)} \leftarrow & \texttt{e\_hist(J, Eno, Frm, To)}, \\
 & \neg\texttt{e\_hist(J + 1, \_, \_, \_)}.
\end{array}
$$

The program P of Example 3 is XY-stratified since the nodes of P_{bis} can be sorted into the following strata $\sigma_0 = \{$`distinct`, `select_larger`, `old_overlap`, `old_e_hist`$\}$, $\sigma_1 = \{$`new_overlap`$\}$, $\sigma_2 = \{$`new_e_hist`$\}$.

Thus, in the corresponding local stratification, the atoms `distinct` and the atoms `select_larger` go to the bottom stratum S_0. Then the atoms in σ_1 are in strata $S_{j\times 2+1}$, while those in σ_2 are now in strata $S_{j\times 2+2}$ (j denotes the temporal argument of these atoms).

The second `e_hist` rule in Example 3 is a qualified copy rule; that is, the head is copied from the body provided that certain conditions are satisfied. This can be implemented by letting `new_e_hist` and `old_e_hist` share the same table, and use deletion-in-place for those tuples that do not satisfy the two negated goals. The goal `overlap(J, _, _, _, _)` ensures that the computation stops as soon as no more overlapping intervals are found.

As demonstrated by these examples, XY-stratified programs allow an efficient logic-based expression of procedural algorithms—in fact the alternating fixpoint used in the computation of well-founded models can also be expressed using these programs [3]. Also observe that, for the examples used here, the bistate program is nonrecursive, as in the original definition of XY-stratification [12]. However, the definition of explicit local stratification in [3] allows for recursive bistate programs. In general, by making the differential fixpoint computation of the recursive predicate explicit, as it was done for the `anc` example, it is possible to rewrite a program having a recursive bistate version into an equivalent one, where the bistate program is nonrecursive. Thus, we will here assume that all our programs have this property, and take advantage of the simpler treatment of XY-stratification that follows from this assumption.

3 Updates and History Relation

In general, logic-based systems have not dealt well with database updates. For instance, Prolog resorts to its operational semantics to give meaning to `assert` and `retract` operations. Similar problems are faced by deductive database systems, which, however, concentrate on changes in the base relations, rather than in both facts and rules as in Prolog. Active database rules contain updates in both their bodies and their heads. Much current research work pursues the objective of providing a unified treatment for active rules, dealing with updates, and deductive rules, dealing with queries. The two are now viewed as separate areas of database technology, although they often use similar techniques and concepts. Here, we propose a unifying solution for both problems, using the concept of Datalog_{1S} [1] and XY-stratification.

The approach here proposed is driven by the threefold requirement of (1) providing a logical model for updates, (2) supporting the deductive rules and queries of deductive databases, and (3) supporting the active rules of active databases.

The first requirement can be satisfied by using history relation as extensional data. For instance, say that we have an university database, with the following relations:

```
student(Name, Major, Year);
took(Name, Course, Grade);
alumni(Name, Sex, Degree, ClassOf).
```

Then, the `hist` relation might contain the following tuples describing Jim Black's history:

Example 4 *The history of changes for* `Jim Black`

```
hist(2301, +student('Jim Black', ee, freshman)).

hist(4007, −student('Jim Black', ee, freshman)).
hist(4007, +student('Jim Black', ee, sophomore)).

hist(4805, −student('Jim Black', ee, sophomore)).
hist(4805, +student('Jim Black', cs, sophomore)).

hist(6300, −student('Jim Black', cs, sophomore)).
hist(6300, +student('Jim Black', cs, junior)).
```

Thus, `'Jim Black'` joined as a freshman in `ee`. The first column, `2301`, is a change counter that is global for the system—that is, it is incremented for each change request. In fact, several changes can be made in the same SQL update statement: for instance, it might be that all the `ee` freshmen have been updated to sophomores in the same request. Thus, one year later, as Jim moves from freshman to sophomore, the change counter has been set to `4007`. Therefore, there has been a total of $4,007 - 2,301$ database changes during the course of that year. Thus, we represent here deletions by the "−"prefix and inserts by the "+" prefix. An update can be represented by a delete/insert pair having the same value in the temporal argument. More sophisticated representations of updates and other database events (e.g., representing the update directly, and timestamping the tuples) could also be handled in this modeling approach.

The remaining history of Jim Black's records states that he became a sophomore, and then he changed his major from `ee` to `cs`. Finally Jim became a junior—and that represents the current situation.

For a history database to be correct, it must satisfy a *continuity axiom*, which basically states that there is no hole in the history log. Thus we might write:

$$\texttt{bad\_history} \leftarrow \quad \texttt{hist}(\texttt{J}+1, \_), \neg\texttt{hist}(\texttt{J}, \_).$$

Support for deductive queries against the current database is achieved through snapshot predicates, derived from the history relations using frame axioms. For the `student` relation, for instance, we have the following rules:

Example 5 *Snapshot predicates for* `student` *via frame axioms*

$$\begin{array}{l}\texttt{student\_snap}(\texttt{J}+1, \texttt{Name}, \texttt{Major}, \texttt{Level}) \leftarrow \\ \qquad\qquad \texttt{student\_snap}(\texttt{J}, \texttt{Name}, \texttt{Major}, \texttt{Level}), \\ \qquad\qquad \neg\texttt{hist}(\texttt{J}+1, -\texttt{student}(\texttt{Name}, \texttt{Major}, \texttt{Level})). \\ \texttt{student\_snap}(\texttt{J}, \texttt{Name}, \texttt{Major}, \texttt{Level}) \leftarrow \\ \qquad\qquad \texttt{hist}(\texttt{J}, +\texttt{student}(\texttt{Name}, \texttt{Major}, \texttt{Level})).\end{array}$$

These rules express what are commonly known as *frame axioms*. Basically, the content of a database relations remain same, except for the deleted tuples (that are subtracted), and the inserted tuples that are added. Observe that `student_snap` rules depends on `hist`, while the latter does not depend on the former. However, as shown in the next section, with active rules, the next state in `hist` depends on the current events and snapshots. Therefore, a recursive dependency structure emerges, where XY-stratification becomes essential to achieve a well-defined semantics.

Notions derived from XY-sratification are also useful in implementing the positive rules of this section. For instance, observe the first rule in Example 5, which computes the effects of deletions. This is a copy rule that, when only the current content of relations is of interest, can be implemented by removing the tuples satisfying the negated goal (i.e., the deleted tuples) and leaving the rest of the relation unchanged. Similar frame axiom rules will be defined for each relation in the schema.

Deductive queries are then answered against the current content of the database (i.e., the final value of the change counter).

Example 6 *The current content of the relation* `student`

$$\begin{array}{ll}\texttt{current\_state}(\texttt{J}) \leftarrow & \texttt{hist}(\texttt{J}, \_), \neg\texttt{hist}(\texttt{J}+1, \_). \\ \texttt{student}(\texttt{Name}, \texttt{Major}, \texttt{Year}) \leftarrow & \texttt{student\_snap}(\texttt{J}, \texttt{Name}, \texttt{Major}, \texttt{Year}), \\ & \texttt{current\_state}(\texttt{J}).\end{array}$$

Because of the continuity axiom, the condition $\texttt{hist}(\texttt{J}), \neg\texttt{hist}(\texttt{J}+1)$ uniquely defines the current state `J`.

Similar rules will be written for the remaining relations in the database. The predicates so derived can then used in deriving the usual SQL queries, views and deductive rules of deductive databases. Rules, such as those in Example 5, will be called *current-state rules*.

4 Active Rules with Deterministic Semantics

In this section, we limit our discussion to *Event-Condition-Action* rules, under the immediate-after activation semantics, where we assume that all applicable rules are fired in parallel once their enabling conditions are satisfied.

Active rules can be modeled naturally in this framework, since conditions can be expressed against the snapshot relations, while events and actions can be expressed against history relation. For instance, say that we have the following active rules:

ar_1 : *If a student is added to the alumni relation, then delete his name from the student relation, provided that this is a senior-level student.*

ar_2 : *If a person takes a course, and the name of this person is not in the student relation, then add that name to the student relation, using the (null) value* `tba` *for Major and Level.*

Under an immediately-after activation semantics [14], these active rules can be modeled as follows:

Example 7 *Active rules on histories and snapshots*

$$\begin{aligned} ar_1 : \mathtt{hist}(\mathtt{J}+\mathtt{1}, -\mathtt{student}(\mathtt{Name}, \mathtt{Major}, \mathtt{senior})) \leftarrow & \\ & \mathtt{hist}(\mathtt{J}, +\mathtt{alumni}(\mathtt{Name}, \_, \_, \_)), \\ & \mathtt{student\_snap}(\mathtt{J}, \mathtt{Name}, \mathtt{Major}, \mathtt{senior}). \end{aligned}$$

$$\begin{aligned} ar_2 : \mathtt{hist}(\mathtt{J}+\mathtt{1}, +\mathtt{student}(\mathtt{Name}, \mathtt{tba}, \mathtt{tba})) \leftarrow & \\ & \mathtt{hist}(\mathtt{J}, +\mathtt{took}(\mathtt{Name}, \_, \_)), \\ & \neg\mathtt{student\_snap}(\mathtt{J}, \mathtt{Name}, \_, \_). \end{aligned}$$

We will now define an *active program*, $\mathcal{A}$, to be the logic program consisting of the following clauses:

(1) the history relation,

(2) the rules defining the snapshot predicates, and

(3) the active rules.

In active programs, therefore, the history relation can be viewed as extensional information, while the active rules can be viewed as explicit intensional information; the snapshot rules define implicit intensional information that relate the content of relations to history. On top of this active logic program, we have the deductive logic program which is defined by the current-state rules and the usual deductive rules and queries. In the the rest of the paper, we will focus on the properties of active programs that are the much less understood than those of deductive ones.

Because of its XY-stratified structure an active program has a unique stable model M, which defines the meaning of the program.

Now assume that a new tuple is added to the history relation. Active rules can be triggered and their actions must then be entered as events in the history relation; these events can trigger more rules, and so on, until no more rules can fire. However, only the external changes requested by a user, and those triggered by active rules, can be in the history relation. Therefore, an active program $\mathcal{A}$ must satisfy the following two axioms:

1. *Completeness Axiom.* The history relation in $\mathcal{A}$ must be identical to the history relation in the stable model of $\mathcal{A}$.

2. *External Causation Axiom.* Let $\mathcal{A}_{ext}$ be the logic program obtained from $\mathcal{A}$ by eliminating from the history relation all changes but the external changes requested by users. Then, the stable model of $\mathcal{A}_{ext}$ and the stable model of the original $\mathcal{A}$ must be identical.

These two requirements provide an obvious adaptation to our current framework of the general requirements of logic-based semantics, where a canonical model M is characterized by the conditions that (i) no additional atoms are implied by M and the rules, and (ii) each atom in M is supported by an implication chain starting from the initial set of given facts.

Thus, while predicates defined by deductive rules behave as virtual views, the history relation behaves as a concrete view. Whenever a new change is requested by users, all changes implied by the active rules are also added to the history relation. The complete set of new entries added to the history relation define the response of the system to the changes requested by the user.

5 Nondeterministic Reasoning

Say that, with relation `student(Name, Major, Year)`, our university database contains the relation `professor(Name, Major)`. In fact, say that our toy database contains only the following facts:

```
student('Jim Black', ee, senior).        professor(ohm, ee).
                                         professor(bell, ee).
```

Now, the rule is that the major of a student must match his/her advisor's major area of specialization. Then eligible advisors can be computed as follows:

```
elig_adv(S, P) ←   student(S, Majr, Year),  professor(P, Majr).
```

This yields

```
elig_adv('Jim Black', ohm).
elig_adv('Jim Black', bell).
```

But, since a student can only have one advisor, the goal `choice((S), (P))` must be added to force the selection of a unique advisor, out of the eligible advisors, for a student.

Example 8 *Computation of unique advisors by choice rules*

```
actual_adv(S, P) ←   student(S, Majr, Yr),  professor(P, Majr),
                     choice((S), (P)).
```

The goal `choice((S), (P))` can also be viewed as enforcing the functional dependency (FD) $\mathtt{S} \rightarrow \mathtt{P}$; thus, in `actual_adv`, the second column (professor name) is functionally dependent on the first one (student name).

The result of executing this rule is *nondeterministic*. It can either give a singleton relation containing the tuple $(\mathtt{'Jim\ Black'}, \mathtt{ohm})$ or that containing the tuple $(\mathtt{'Jim\ Black'}, \mathtt{bell})$.

Program rules containing one or more choice goals are called choice rules; a program with choice rules is called a *choice program*. In the choice goal `choice((S), (P))`, `S` and `P` are, respectively, called the first and the second argument. In general the first argument of a choice goal is a list of zero or more variables, while the second argument must consists of one or more variables.

The semantics of a choice program P can be defined by transforming P into a program with negation, $foe(P)$, called the *stable version* of a choice program P. $foe(P)$ exhibits a multiplicity of stable models, each obeying the FDs defined by the choice goals. Each stable model for $foe(P)$ corresponds to an alternative set of answers for P and is called a *choice model* for P. $foe(P)$ is defined as follows:

Definition 3 *The* first order equivalent *of a choice program P will be denoted $foe(P)$; $foe(P)$ is obtained by the following transformation. Consider a choice rule r in P:*

$$r : A \leftarrow B(Z),\ choice((X_1), (Y_1)),\ \ldots,\ choice((X_k), (Y_k)).$$

where,

(i) $B(Z)$ denotes the conjunction of all the goals of r that are not choice goals, and

(ii) X_i, Y_i, Z, $1 \leq i \leq k$, denote vectors of variables occurring in the body of r such that $X_i \cap Y_i = \emptyset$ and $X_i, Y_i \subseteq Z$.

Then the original program P is transformed as follows:

1. *Replace r with a rule r' obtained by substituting the choice goals with the atom $chosen_r(W)$:*

   $$r' : A \leftarrow B(Z),\ chosen_r(W).$$

   where $W \subseteq Z$ is the list of all variables appearing in choice goals, i.e., $W = \bigcup_{1 \leq j \leq k} X_j \cup Y_j$.

2. *Add the new rule*

   $$chosen_r(W) \leftarrow B(Z),\ \neg diffChoice_r(W).$$

3. *For each choice atom $choice((X_i), (Y_i))$ $(1 \leq i \leq k)$, add the new rule*

   $$diffChoice_r(W) \leftarrow chosen_r(W'),\ Y_i \neq Y_i'.$$

   where (i) the list of variables W' is derived from W by replacing each $A \in Y_i$ with a new variable $A' \in Y_i'$ (i.e., by priming those variables), and (ii) $Y_i \neq Y_i'$ is true if $A \neq A'$, for some variable $A \in Y_i$ and its primed counterpart $A' \in Y_i'$.

The foe of Example 8 is given in Example 9; this can be read as a statement that a professor will be assigned to a student if a different professor has not been assigned to the same student.

Example 9 *The foe of the rule in Example 8*

$$
\begin{array}{ll}
\mathtt{actual\_adv(S,P)} \leftarrow & \mathtt{student(S,Majr,Yr), professor(P,Majr),} \\
 & \mathtt{chosen(S,P).} \\
\mathtt{chosen(S,P)} \leftarrow & \mathtt{student(S,Majr,Yr), professor(P,Majr),} \\
 & \neg\mathtt{diffChoice(S,P).} \\
\mathtt{diffChoice(S,P)} \leftarrow & \mathtt{chosen(S,P'), P \neq P'.}
\end{array}
$$

In general, the program $foe(P)$ generated by the transformation discussed above has the following properties:

- $foe(P)$ has one or more total stable models.
- The `chosen` atoms in each stable model of $foe(P)$ obey the FDs defined by the choice goals.

The stable models of $foe(P)$ are called *choice models* for P.

Stratified Datalog programs with choice are in DB-PTIME: actually they can be implemented efficiently by producing `chosen` atoms one at a time and memorizing them in a table. The `diffchoice` atoms need not be computed and stored; rather, the goal ¬`diffchoice` can simply be checked dynamically against the table `chosen`.

The use of choice is critical in many applications [14]. The most significant applications involve the use of choice in recursive predicates. For instance, the following program computes the spanning tree, starting from the source node `a`, for a graph where an arc from node `b` to `d` is represented by the database fact `g(b,d)`.

Example 10 *Computing a spanning tree*

$$
\begin{array}{ll}
\mathtt{st(root,a).} & \\
\mathtt{st(X,Y)} \leftarrow & \mathtt{st(\_,X),\ \ g(X,Y),\ \ Y \neq a,\ \ choice((Y),(X)).}
\end{array}
$$

In this example, the goal $\mathtt{Y} \neq \mathtt{a}$ ensures that, in `st`, the end-node for the arc produced by the exit rule has an in-degree of one; likewise, the goal `choice((Y),(X))` ensures that the end-nodes for the arcs generated by the recursive rule have an in-degree of one.

Stratified Datalog programs with choice are also DB-PTIME complete [14], without having to assume that the universe is totally ordered. In fact, the following program generates a precedence relation that totally orders `d(Y)`:

$$
\begin{array}{ll}
\mathtt{ordered\_d(root,root).} & \\
\mathtt{ordered\_d(X,Y)} \leftarrow & \mathtt{ordered\_d(\_,X),\ \ d(Y),} \\
 & \mathtt{choice((X),(Y)),\ \ choice((Y),(X)).}
\end{array}
$$

The choice goals in this example ensure that once an arc `(X,Y)` is generated, this is the only arc leaving the source node `X` and the only arc entering the sink node `Y`.

We can now combine the notions of stratification and XY-stratification with that of choice programs as follows:

Definition 4 *The* deterministic version *of a choice program P is constructed from P by eliminating all choice goals. The deterministic version of P is denoted $det(P)$.*

Definition 5 *Let P be a choice program with negated goals. Then, if its deterministic version $det(P)$ is stratified, P is said to be a stratified choice program.*

Definition 6 *Let P be a program where the rules contain choice goals and negated recursive goals. Then, P is said to be an XY-stratified choice program when*

- *$det(P)$ is XY-stratified*
- *Each choice goal in a recursive rule r contains the temporal variable of r in its first argument.*

The second condition ensures that the FD constraints implied by the choice goals are local to a specific stratum rather than spanning over multiple states.

As shown in the Appendix, the following properties hold, for a XY-stratified choice program P:

1. P always has one or more total stable models
2. Each such a model obeys the functional dependencies specified by the choice goals of P
3. Procedure 2 for computing the well-founded model of a XY-stratified program must be modified as follows:
   - When constructing P_{bis} the temporal variables are dropped from the choice goals.
   - In Step 1 of Procedure 2, the iterated fixpoint procedure on the synchronized program P_{bis} must compute a choice model, rather than the least model, for the strata containing choice goals.

   With these modifications, Procedure 2 is sound and (nondeterministically) complete for the computation of the stable models for P.

6 Nondeterministic Activation

If e is an event, and the body of $n > 1$ rules are enabled by this event, then, the SQL3 semantics (ignoring user-defined priorities between rules that will be discussed later) prescribes that any of the n rules can be triggered. If rule r is triggered, then the events specified in the head of this rule are added to those waiting to be serviced. Any of the new events waiting to be serviced can then be selected to trigger the next rule r'; the only condition is that an event instance e can not trigger the same rule instance twice. While the general framework and the axioms of continuity, completeness, and external causation remain the same, we need to use two new temporal tables as follows: the `evt` table stores events that occurred since the last external event, while the `memo` table keeps track of pairs event/rule-number used since the last external event, to avoid duplicate firings. The table `evt` copies the last external event from history as follows:

ir_1 : $\texttt{evt(J,Ev)} \leftarrow\quad \texttt{hist(J,Ev)}.$

Active rules considered for firing are described by the predicate `enabled`; only rules where the combination event/rule-number is not contained in `memo` can fire (as `memo` memorizes the pairs that have already fired). Therefore, the previous rules ar_1 and ar_2 are modeled as follows (we preserve in the head, both the candidate enabling event `Ev` and the unique rule number—ar_1 and ar_2 in the example below):

$$\begin{aligned} ar_1 : \texttt{enabled}(\texttt{J}+1, -\texttt{student}(\texttt{Name}, \texttt{Major}, \texttt{senior}), \texttt{ar}_1, \texttt{Ev}) \leftarrow \; & \texttt{evt}(\texttt{J}, \texttt{Ev}), \texttt{Ev} = +\texttt{alumni}(\texttt{Name}, \_, \_, \_), \\ & \texttt{student\_snap}(\texttt{J}, \texttt{Name}, \texttt{Major}, \texttt{senior}), \\ & \neg\texttt{memo}(\texttt{J}, \texttt{ar}_1, \texttt{Ev}). \end{aligned}$$

$$\begin{aligned} ar_2 : \texttt{enabled}(\texttt{J}+1, +\texttt{student}(\texttt{Name}, \texttt{tba}, \texttt{tba}), \texttt{ar}_2, \texttt{Ev}) \leftarrow \; & \texttt{evt}(\texttt{J}, \texttt{Ev}), \texttt{Ev} = +\texttt{took}(\texttt{Name}, \_, \_), \\ & \neg\texttt{student\_snap}(\texttt{J}, \texttt{Name}, \_, \_), \\ & \neg\texttt{memo}(\texttt{J}, \texttt{ar}_2, \texttt{Ev}). \end{aligned}$$

Only one enabled rule instance is chosen for each `J`; this can be accomplished with a choice construct as follows:

$$\begin{aligned} ir_2 : \texttt{fired}(\texttt{J}, \texttt{Actions}, \texttt{RNo}, \texttt{Ev}) \leftarrow \; & \texttt{enabled}(\texttt{J}, \texttt{Actions}, \texttt{RNo}, \texttt{Ev}), \\ & \texttt{choice}((\texttt{J}), (\texttt{RNo}, \texttt{Ev})). \end{aligned}$$

Thus, if multiple event instances occur at time `J`, each enabling several rules to fire, only one of these rules will be chosen in a nondeterministic fashion and will fire at time `J`. However, the firing of this rule could result in several actions to be taken in parallel, as per the condition part of the rule. For instance, if we have the event $\texttt{Ev} = +\texttt{alumni}('\texttt{Joe Doe}', \ldots)$, where $'\texttt{Joe Doe}'$ is a senior in 10 different majors, then 10 deletion actions will first be enabled in the head of the rule ar_1, and then executed if the pair $(\texttt{ar}_1, \texttt{Ev})$ is chosen by ir_2.

Then, we have to do some upkeeping; thus, the `Actions` are added to history, while the pair (`RNo`, `Ev`) is added to `memo`.

$$ir_3 : \texttt{hist}(\texttt{J}, \texttt{Actions}) \leftarrow \; \texttt{fired}(\texttt{J}, \texttt{Actions}, \_, \_).$$

$$\begin{aligned} ir_4 : \texttt{memo}(\texttt{J}+1, \texttt{RNo}, \texttt{Ev}) \leftarrow \; & \texttt{memo}(\texttt{J}, \texttt{RNo}, \texttt{Ev}), \\ & \texttt{fired}(\texttt{J}+1, \_, \_, \_), \neg\texttt{hist}(\texttt{J}, \texttt{Ev}). \\ ir_5 : \texttt{memo}(\texttt{J}, \texttt{RNo}, \texttt{Ev}) \leftarrow \; & \texttt{fired}(\texttt{J}, \_, \texttt{RNo}, \texttt{Ev}), \neg\texttt{hist}(\texttt{J}, \texttt{Ev}). \end{aligned}$$

Rule ir_4, above, is a copy rule that preserves in `memo` the pairs that have already fired rules, while updating their temporal argument to the current value. However, whenever the new event fired is identical to an old one, the new event msut be allowed to fire unconditionally. This is accomplished by simply removing all the event instances from `memo`.

A copy rule is also needed for the `evt` table:

$$ir_6 : \texttt{evt}(\texttt{J}+1, \texttt{EvT}) \leftarrow \; \texttt{evt}(\texttt{J}, \texttt{EvT}), \texttt{fired}(\texttt{J}+1, \_, \_, \_).$$

In summary, rule ir_1 copies an external event requested by a user to the table `evt`, where it can enable active rules such as ar_1 and ar_2 (for which the rule/event pair has not been used before). Then, the choice rule ir_2 selects and fires an arbitrary rule instance among the enabled ones; the actions in the head of the fired rule are added to the `memo` table by rules ir_4 and ir_5. When no active rule makes the `enabled` predicates true, then the `fired` predicate is no longer true and rule ir_6 is no longer activated, and no new event is added to `hist`. Only when another external event (e.g., a user request) occurs later, this is added to the tables `hist` and `evt`. However, the `fired` goals in the copy rules are now false and thus, the auxiliary table `memo` are restarted empty; and so is table `evt` but for the new entry added by rule ir_1.

These rules yield an XY-stratified choice program; therefore, the complex nondeterministic behavior of SQL3 rules can be modeled in the deductive DB framework using non-monotonic constructs already supported in systems such as $\mathcal{LDL}$++. While additional artifacts (i.e., the predicates `memo evt` and their supporting rules) were required, the simple axiomatization established by the axioms of *continuity, completeness and external causation* remain unchanged.

Further improvements to simplify these rules, and to eliminate function symbols are also possible but will not be discussed here for lack of space. Also SQL3 declaration for explicit priorities between rules can be included in a natural fashion: we store the priorities between rules in a closure relation $\texttt{precedes}(\texttt{RN1}, \texttt{RN2})$. Then the `fired` rule becomes:

$$
\begin{aligned}
ir'_1 : \texttt{fired}(\texttt{J}, \texttt{Action}, \texttt{RNo}, \texttt{Ev}) \leftarrow \quad & \texttt{enabled}(\texttt{J}, \texttt{Action}, \texttt{RNo}, \texttt{Ev}), \\
& \neg\texttt{higher\_priority}(\texttt{J}, \texttt{RNo}), \\
& \texttt{choice}((\texttt{J}), (\texttt{RNo}, \texttt{Ev})).
\end{aligned}
$$

$$
\begin{aligned}
ir'_2 : \texttt{higher\_priority}(\texttt{J}, \texttt{RNo}) \leftarrow \quad & \texttt{enabled}(\texttt{J}, \_, \texttt{RN1}, \_), \\
& \texttt{precedes}(\texttt{RN1}, \texttt{RNo}).
\end{aligned}
$$

7 Conclusion

The main result of this paper is that the semantics of SQL3 updates and active rules can be fully expressed with the more advanced nonmonotonic constructs implemented in deductive database prototypes. While, additional artifacts (such as auxiliary relations) had to be used to achieve our goal, the additional effort and complications they require is modest, when compared to that required for mastering competing approaches to logic-based semantics for active rules, or even, the operational semantics of active rules in SQL3 proposed standards. Moreover, substantial benefits can follow from our additional effort, including:

- The design of DB languages where active and deductive rules can be freely combined to yield much better support for complex queries, reasoning and advanced knowledge-based applications, and
- Powerful logic-based tools to solve critical active-rule problems, such as termination, confluence, and design of concrete views.

Future work will pursue these objectives, and also address important aspects of database changes not discussed here. One is the logic-based modeling of transactions. Several solutions were proposed for this problem; remarkably, the

Statelog-based approach [8] has many techniques in common with the Datalog$_{1s}$ approach used here. Another interesting problem is the modeling of different activation semantics for rules, including immediate-before, and deferred-activation semantics, but also the durable-change semantics [13] (originally defined only for deterministic rule activation), which has the important advantage of avoiding infinite activation loops.

Acknowledgment

The author would like to thank Nanni Mirko for his critical comments on XY-stratified choice programs.

References

[1] M. Baudinet, J. Chomicki, and P. Wolper. Temporal Deductive Databases, in Tansel et al (eds), pp. 294–320, *Temporal Databases: Theory, Design, and Implementation*, Benjamin/Cummings, 1994.

[2] A. Bonner and M. Kifer. An Overview of Transaction Logic. *Theoretical Computer Science* 133, 1994.

[3] D. B. Kemp, K. Ramamohanarao, P. J. Stuckey, "ESL-programs and the efficient evaluation of nonstratified programs by transformation to ESL"," In *Procs., International Conference on Deductive and Object-Oriented Databases, DOOD'95*, pp. 91–100, Springer-Verlag, 1995.

[4] M. Gelfond and V. Lifschitz. The stable model semantics for logic programming. In R.A. Kowalski and K. Bowen, editors, *Logic programming: Proceedings of the fifth Int'l Conf. and Symposium*, pp. 1070–1080, 1988.

[5] G. Phipps, M. A. Derr, and K. A. Ross. Glue-Nail!: A Deductive Database System. In *Proceedings International ACM SIGMOD Conference on Management of Data*, pp. 308–317, 1991.

[6] A. Van Gelder, K. A. Ross, and J. S. Schlipf. The Well-Founded Semantics for General Logic Programs. *Journal of ACM* 38:620–650, 1991.

[7] F. Giannotti, D. Pedreschi, D. Saccà, and C. Zaniolo. Non-Determinism in Deductive Databases. In C. Delobel, M. Kifer, and Y. Masunaga (eds.), *Proceedings Deductive and Object-Oriented Databases, Second International Conference, DOOD'91*, pp. 129–146, Springer-Verlag, 1991.

[8] B. Lud/ascher, W. May and G. Lausen, Nested Transactions in a Logical Language for Active Rules, *Procs. of Logic in Databases (LID'96) Workshop*, D. Pedreschi and C. Zaniolo (Eds.), Springer, 1996.

[9] Jayen Vaghani, Kotagiri Ramamohanarao, David B. Kemp, Zoltan Somogyi, Peter J. Stuckey, Tim S. Leask, and James Harland. The Aditi deductive database system. *VLDB Journal*, 3:245–288, 1994.

[10] R. Ramakrishnan, D. Srivastava, S. Sudanshan, and P. Seshadri. Implementation of the CORAL Deductive Database System. In *Proceedings of the International ACM SIGMOD Conference on Management of Data*, pp. 167–176, 1993.

[11] K. A. Ross. Modular Stratification and Magic Sets for Datalog Programs with Negation. *Journal of ACM* 41(6):1216–1266, 1994.

[12] C. Zaniolo, N. Arni, and K. Ong, Negation and aggregates in recursive rules: the $\mathcal{LDL}$++ approach In *Procs., International Conference on Deductive and Object-Oriented Databases, DOOD'93*, pp. 204-221 1993.

[13] Carlo Zaniolo, Active Database Rules with Transaction-Conscious Stable-Model Semantics In *Procs., International Conference on Deductive and Object-Oriented Databases, DOOD'95*, pp. 55-72 Springer 1995.

[14] C. Zaniolo, S. Ceri, C. Faloutsos, R. Snodgrass, V.S. Subrahmanian, and R. Zicari, *Advanced Database Systems*, Morgan Kaufmann, 1997.

Appendix

Let $\Sigma_{det(P)}$ be a stratification (i.e., an ordered partition of the predicate names) of P. We can now extend this stratification to the predicate names in $foe(P)$ to obtain a partition of $foe(P)$ by assigning each $\texttt{chosen}_\texttt{r}$, and $\texttt{diff\_choice}_\texttt{r}$ predicates to the same stratum as that to which the head of **r** belongs to.

Now for a stratified program Q let Q^j denote the rules in Q with the head predicate name in the j^{th} stratum. Then, the following algorithm provides a simple generalization to the stratified fixpoint used to compute the (unique) stable model for a stratified program:

Iterated Choice Fixpoint for a stratified program Q:
Base: $M^0 = \emptyset$
Induction: For $j > 0$, let M^j = a choice model for $M^{j-1} \cup Q^j$
Result: $M^\omega = \bigcup_{0 \leq j < \omega} M^j$

For an XY-stratified choice program P, we extend the local stratification of $det(P)$ to $foe(P)$ by making sure that, for a recursive rule **r** with temporal variable k, the atoms $\texttt{chosen}_\texttt{r}(\texttt{k}, \ldots)$ and $\texttt{diff\_choice}_(\texttt{k}, ...)$ are assigned to the same stratum as $\texttt{head}_\texttt{r}(\texttt{k}, \ldots)$, where $\texttt{head}_\texttt{r}$ is the predicate name of the head of r.

Now, with Q a locally stratified program, we denote by $ground^j(Q)$ the clauses of $ground(Q)$ with head in the j-th stratum. Then, the iterated choice fixpoint can then modified by letting the induction step be as follows:

For $j > 0$, let M^j = a choice model for $M^{j-1} \cup ground^j(Q)$.

The soundness and completeness of the iterated choice fixpoint to compute the stable models of XY-stratified programs then follow directly from the definitions:

Theorem 2 *Let P be a stratified choice program. Then, the set of results of the Iterated Choice Fixpoint is equal to the set of stable models for P.*

Finally, with the modifications in the choice rules for P_{bis} described in Section 5, Procedure 2 provides an efficient implementation of the iterated choice fixpoint.