Rules In Database Systems—RIDS'93

A Unified Semantics for Active and Deductive Databases

Carlo Zaniolo
Computer Science Department
University of California
Los Angeles, California U.S.A.
zaniolo@cs.ucla.edu

Abstract

These two rule-oriented paradigms of databases have been the focus of extensive research and are now coming of age in the commercial DBMS world. However, the systems developed so far support well only one of the two paradigms—thus limiting the effectiveness of such systems in many applications that require complete integration of both kinds of rules. In this paper, we discuss the technical problems that make such an integration difficult, and trace their roots to a lack of a unified underlying semantics. Then, we review recent advances in the semantics of non-monotonic logic and show that they can be used to unify the foundations of active databases and deductive databases. Finally, we outline the design a new rule language for databases that integrates a deductive system with a trigger-based DBMS.

1 Introduction

Rules provide the main paradigm for expressing computation in active databases and deductive databases. The unification of the two paradigms represents a research problem of obvious theoretical interest and many important applications could benefit from it; yet, there has been little formal work on the marrying these powerful paradigms. While cultural and historical biases share the blame for this chasm, the root of the problem is actually technical and can be traced to certain semantic inadequacies in the conceptual foundations of both approaches.

Several active database languages and systems have been proposed so far: a very incomplete list include [?, ?, ?, ?, ?]. However, there is is no unifying semantic theory for active databases: most of the work done so far has concentrated on explaining operational semantics of particular systems. On the contrary, not one but three equivalent formal semantics exist for Horn clauses that form the core of deductive database languages [?, ?]. Unfortunately, this elegant semantics is brittle and can not be generalized easily to deal with non-monotonic constructs, such as negation and updates. Similar non-monotonic reasoning problems have emerged in the areas of knowledge representation and of logic programming and remain the focus of intense research. While deductive databases encountered early successes in this problem area (e.g., with the introduction of the concept of stratified negation), recent progress has been very slow. No totally satisfactory semantics currently exist for programs which use non-monotonic constructs such as negation and aggregates in recursion—and the problem of updates in recursion is understood even less. Also, the problem of finding a logic for reasoning about actions and situations represents one of the classical challenge of AI [?].

Given this rather ominous background, the solution presented in this paper is surprisingly simple and general. We introduce the notion of XY-stratified programs that allow non-monotonic constructs in recursive rules. Then, we show that a formal semantics for updates and triggers in databases can be given using XY-stratified programs. The blueprints for the design of a unified rule language for active databases and deductive databases follow from such a solution.

2 Logic-Based Semantics

This area has benefited significantly from research in deductive databases. The adoption of the fixpoint-based bottom-up approach to define the declarative and constructive semantics of logic programs led almost immediately to the concept of stratified negation and stratified set aggregates [?]. This concept removes several of the limitations and problems of Prolog's negation-by-failure, and it is conductive to efficient implementation, as demonstrated by systems such as *Glue-Nail*, $\mathcal{LDL}$ and *CORAL* [?, ?, ?]. However, experience gained with real-life applications [?] revealed that stratification is too restrictive and there remain many important applications where negation and set aggregates are needed: such applications, range from processing a Bill of Materials to finding the shortest path in a graph [?].

Therefore, during the last five years, a substantial research effort has been devoted to solving the non-stratification issue. This endeavor has produced significant progress on the theoretical front, with the introduction of concepts such as locally stratified programs, well-founded models [?], and the stable models[?], but it has not yet begotten a solution that is both general and practical. Indeed a practical solution must satisfy three difficult requirements, inasmuch as it must

- have a formal logic-based semantics,
- have a simple and intuitive constructive semantics,
- be amenable to efficient implementation.

Thus, in addition to requiring formal semantics and efficient implementation, any practical proposal must also stress the importance of having a simple concrete semantics: i.e., one that can be easily comprehended by the application programmer, without a need to understand abstract formalisms. For instance, a notion such as stratification can be mastered by a programmer, who can make full use of it without having to understand its formal perfect-model semantics. Furthermore, it is simple for a compiler to verify that stratification is satisfied, and then support stratified programs by an efficient bottom-up computation. However, an understanding of the logical formalism is required to understand notions such as well-founded models or stable models. Furthermore, no simple syntactic check exists for deciding whether a program has a well-founded model or a stable model; when such models exist their computation can be very expensive.

The notion of XY-stratified programs was recently proposed to overcome these difficulties [?]. This is a special subclass of locally stratified programs that is easy for a compiler to recognize and implement using the fixpoint-based computation of deductive DBs. It was shown in [?] that classical computational problems such as Floyd's shortest path algorithms, can be expressed naturally by programs in this class using non-stratified negation and aggregates. In this paper, we will build on such a semantics to provide a formal model of updates in deductive databases and triggers in active databases.

The problem of providing a formal semantics to updates in the context of logic also represents a difficult challenge. Several of the approaches proposed deal with the more general problem of modeling revisions of knowledge bases—i.e., including additions, deletions and modifications of rules [?, ?, ?, ?]. As a result, these theories are more complex and less conducive to efficiency than it is desirable in a practical semantics, according to the criteria discussed above. Therefore, we will restrict our attention to the problem of modifying the extensional database only [?]. Most of the work done in this more specific context is based on Dynamic Logic [?]. In particular, the approach given in [?, ?] uses dynamic logic to formalize a concrete semantics where the updates take place according to rule instances—a *tuple-at-a-time* semantics often leading to non-determinism. However, relational databases support a *set-at-a-time* update semantics, where all the applicable updates are fired at once, in parallel. A set-at-a-time semantics based on dynamic logic was adopted and efficiently implemented in $\mathcal{LDL}$ [?, ?]. While the $\mathcal{LDL}$ design has progressed further than other systems toward a complete integration of database-oriented updates into a logic-based language, several problems remain, such as multiple update rules sharing the same heads, and failing goals after update goals [?]. Furthermore, since dynamic logic is quite different from standard logic, the two do not mix well, and, as a result, updates are not allowed in recursive $\mathcal{LDL}$ rules; a special construct, called `forever` had to be introduced to express `do-while` iterations over updates [?]. A further illustration of the difficulties encountered by declarative logic-based languages in dealing with updates is provided by the design of the *Glue-Nail* system [?]. In this second-generation deductive database system, updates were banned from the core declarative language and relegated to the procedural shell that is tightly wrapped around the core [?].

Viewed against the tormented landscape of previous work, the model of updates and active databases proposed in this paper is surprisingly simple. Basically, we define rules with updates in their heads by simply re-writing them into equivalent update-free logic programs that are XY-stratified. The common semantics of active and deductive predicates, so obtained, is the basis for our unification of the active and deductive aspects of databases. This semantics is conducive to the design of a powerful language capable of expressing reasoning, triggers and detection of events as required by the next generation of intelligent database applications.

2.1 XY-Stratification

We begin with a simple example [1] that computes the nodes `X` of a graph `g` reachable from a given node `a`, and the minimum distance of `X` from this node:

Example 1 *Reachable nodes:*

$$
\begin{aligned}
\mathtt{r_0}: \mathtt{delta(nil,X,1)} \leftarrow \;& \mathtt{g(a,X)}.\\
\mathtt{r_1}: \mathtt{delta(s(I),Y,D1)} \leftarrow \;& \mathtt{delta(I,X,D),g(X,Y),}\\
& \neg\mathtt{all(I,Y,\_),D1 = D + 1}.\\
\mathtt{r_2}: \mathtt{all(s(I),Y,D)} \leftarrow \;& \mathtt{all(I,Y,D),delta(s(I),\_,\_)}.\\
\mathtt{r_3}: \mathtt{all(I,Y,D)} \leftarrow \;& \mathtt{delta(I,Y,D)}.
\end{aligned}
$$

This program presents several unusual traits. A most obvious one is the presence of terms such as `nil, I, s(I)` in the first argument of the recursive predicates. These arguments will be called *stage arguments*, and their usage is for counting as in the recursive definition of integers: `nil` stands for zero and `s(I)` stands for `I+1`.

The intuitive meaning of the program of Example 11 is quite obvious: it implements a seminaive computation of a transitive closure [?], through the use of two predicates: `delta` contains the new values, and `all` is the union of all values computed so far. In rule r_1, `all` is used for checking that no previous (therefore shorter) path exists to this node.

The formal semantics of the program also supports its intuitive semantics. Because of its particular syntactic form, the program is locally stratified, where each stratum corresponds to a different value of the stage argument. The first stratum contains atoms of the form:

`delta(nil,...), all(nil,...)`

the next stratum consists of all atoms of the form:

`delta(s(nil),...), all(s(nil),...)`

[1]We assume our reader familiar with the basic concepts pertaining to Datalog and logic rules including the concept of locally stratified programs and the *iterated fixpoint procedure* that computes the perfect model of these programs [?, ?]. Given a program P, a set of rules of P defining a maximal set of mutually recursive predicates will be called a *recursive clique* of P. The non-recursive rules in this set are called *exit* rules (r_0 is the only exit rule in Example 1).

and so on. The particular syntactic form of the recursive rules, w.r.t. the stage arguments, makes it simple for a compiler to recognize the presence of a *locally stratified program* (although local stratification is normally undecidable in the presence of function symbols).

Furthermore, this type of program can be implemented efficiently using a modified fixpoint computation. In general, the perfect model of such a program can only be obtained using a transfinite computation called iterated fixpoint [**?**]. This proceeds as follows: the least fixpoint is first computed for the first (bottom) stratum; then, once the least fixpoint is computed for the n-th stratum, the least fixpoint is computed for the $n+1$-th stratum. The transfinite computation of perfect models simplifies dramatically for the program at hand. Assume, for now, that the stratification is determined by the values of stage arguments. The fixpoint at the bottom stratum is reached after firing rule r_0 followed by r_3. The fixpoint for the stratum `s(nil)` is reached by firing rule r_1, followed by firing r_2 and r_3. Then, the higher strata are inductively generated by firing these three rules in the same order. Therefore, the general transfinite procedure to compute perfect models here reduces to the customary fixpoint iteration. Further improvements can be easily made to ensure that this computation executes quite efficiently. For instance, while rule r_2 seems to suggest that a complete copying of the old relation is needed at each step, no such operation is needed in reality. In fact, the only instances of rules that can produce new atoms are those instantiated with stage values from the current stratum: values from the old strata are not used and can be discarded. Thus, if we keep the values of the stage variable in a separate memory cell, all is needed to perform the copy operation is to increase the value of the integer in this cell by one.

Given a recursive clique, Q, the first arguments of recursive predicates of a rule r (of Q) will be called the *stage arguments* of r (of Q) [2] Then a recursive rule will be called an

- **X-rule** when all the stage arguments of r are equal to a simple variable, say J; furthermore, J does not appear anywhere else in r;
- **Y-rule** when (i) some positive goal of r has as stage argument a simple variable, say J, (ii) the head of r has stage argument $s(J)$, (iii) all the remaining stage arguments are either J or $s(J)$ and (iv) J does not appear anywhere else in r.

In Example 11, r_3 is an X-rule, while r_1 and r_2 are Y-rules. A recursive clique Q such that all its recursive rules are either X-rules or Y-rules, will be said to be a recursive *XY-clique.* We now introduce simple syntactic criteria for classes of non-stratified programs that are amenable to efficient implementation.
Priming: $\mathtt{p}'(...)$ will be called the primed version of an atom $\mathtt{p}(...)$. Given an XY-clique, Q, its primed version Q', is constructed by priming certain occurrences of recursive predicates in recursive rules as follows:

[2]This is only a matter of convention. Alternatively, we could let the last arguments of recursive predicates be our stage arguments.

- X-rules: all occurrences of recursive predicates are primed,
- Y-rules: the head predicate is primed, and so is every goal with stage argument equal to that of the head.

The primed version of our example is as follows:

$$\begin{aligned}
&\mathtt{r_0 : delta(nil, X, 1)} \leftarrow && \mathtt{g(a, X).}\\
&\mathtt{r_1 : delta'(s(I), Y, D1)} \leftarrow && \mathtt{delta(I, X, D), g(X, Y),}\\
& && \mathtt{\neg all(I, Y, _), D1 = D + 1.}\\
&\mathtt{r_2 : all'(s(I), Y, D)} \leftarrow && \mathtt{all(I, Y, D), delta'(s(I), _, _).}\\
&\mathtt{r_3 : all'(I, Y, D)} \leftarrow && \mathtt{delta'(I, Y, D).}
\end{aligned}$$

Definition 1 *An XY-clique Q will be said to be* XY-stratified *when*

- *The primed version of Q is non-recursive*
- *All exit rules have as stage argument the same constant.*

□

A program will be said to be *XY-stratified* when every recursive rule that contains some negated recursive goal(s) in its body belongs to an XY-stratified clique.

The dependency graph for a primed clique provides a very simple syntactic test on whether a program is XY-stratified. Furthermore, these programs are amenable to efficient implementation as proven in the following discussion.

The primed version Q' of an XY-stratified clique defines a non-recursive program: i.e., the dependency graph of Q' contains no cycle. Therefore, there exists a topological sorting of the nodes of Q' which obeys stratification, and such that all unprimed predicate names precede the primed ones. For Example 1, for instance, the following

$$\mathtt{all} < \mathtt{delta} < \mathtt{delta'} < \mathtt{all'}$$

is such a topological sorting. The tail of such a sorting, restricted to the primed predicate names will be called a *primed sorting* of Q. For the previous example, we obtain the following primed sorting:

$$\mathtt{delta'} < \mathtt{all'}$$

Now, we can partition the atoms in Herbrand Base B_Q of the *original* Q into classes according to their predicate names and their stage arguments as follows:

- there is a distinct class, say σ_0, containing all instances of non-recursive predicates in Q—i.e., those without a stage argument.

- all atoms with the same recursive predicate name and the same number of function symbols `s` in the stage argument belong to same equivalence class $\sigma_{n,p}$ with n denoting the number of `s` function symbols in the stage argument of p. Thus, $all(s(nil), ...)$, and $all(s(a), ...$, belong to the same class (stratum) in our example.

The partition Σ of B_Q so constructed can be totally ordered, by letting σ_0 be the bottom stratum in Σ, and then letting $\sigma_{n,p} \prec \sigma_{m,q}$ if

- $n < m$, or
- if $n = m$ but p' precedes q' in the primed sorting of the clique.

The total ordered Σ so constructed will be called the *stage layering* of B_Q. For Example 1, let each of $C_1, C_2, ...$ range over the set $\{nil, a, 1\}$ and each $V_1, V_2, ...$ range over Q's Herbrand Universe; then Σ has the following form:

$$\{\mathtt{g}(\mathtt{V}_1, \mathtt{V}_2)\} \prec$$
$$\{delta(C_1, V_3, V_4)\} \prec \{all(C_2, V_5, V_6\} \prec$$
$$\{delta(s(C_3), V_7, V_8)\} \prec \{all(s(C_4), V_9, V_{10})\} \prec ...$$

A locally stratified program p will be called *strictly* stratified when for every instance r of of a rule in P, the head of r belongs to a layer strictly higher than the layers for the goals of r. Then, we have the following theorem:

Theorem 1 *Each XY-stratified clique Q is strictly stratified according to a stage layering of B_Q.*

Proof: If g is an unprimed goal g for r then, the stage argument of the head of r, $h(r)$ contains one more `s` than g. If g is primed, then $h(r)$ is also primed and its name follows the name of g in the primed ordering. □

Since the stratification is strict, in computing the iterated fixpoint, *the saturation for each stratum is reached in one step.* Therefore, the compiler can resequence the rules according to the primed sorting of their head names. Then, once all atoms with stage value J have been computed, a *single pass through the rules of Q ordered according to the primed sorting* computes all the atoms with stage value $s(J)$. Therefore, if Q is an XY-stratified clique, then (i) Q is locally stratified, and (ii) its perfect model can be constructed by a fixpoint iteration to the first ordinal: the recursive rules in the clique are fired according to the topological layering. We thus obtain the following algorithm, which assumes that the stage arguments are projected out from the rules (the notation w.s.a. will denote a rule without its stage argument) [?]:

Perfect Model Computation for XY-stratified Cliques

Step 1. The stage variable is assigned the stage constant from the exit rules

Step 2. Fire the X-rules (w.s.a), once

Step 3. Fire the recursive rules (w.s.a.) sequentially.

Therefore, the stage argument is updated only once per cycle. XY-stratified programs can express every program expressible under inflationary fixpoint [**?**].

3 Syntactic Encapsulation

New syntactic constructs introduced to replace frequently used expressions of First Order Logic can yield dramatic benefits in terms of readability and efficient implementation. For instance, the simple idea of encapsulating disjunctive clauses, with no more than one non-negated literal, by the basic rule notation yields the improvements in readability and amenability to efficient implementation that are at root of the popularity of the logic programming paradigm. The same method has be used with remarkable success in other situations. For instance, the choice construct was defined in [**?**] to capture the notion of don't care non-determinism and encapsulate certain kinds of negative programs amenable to a polynomial-time computation. Two other such constructs are `if-then(-else)` and `min/max` predicates, both used in the XY-stratified program below. This expresses a linear version of Floyd algorithm for computing the shortest paths in a graph:

Example 2 *Floyd Algorithm.*

```
delta(nil, X, X, 0).
delta(s(J), X, Z, min(< C >)) ←  delta(J, X, Y, C1),
                                 g(Y, Z, C2), C = C1 + C2.
all(s(J), X, Z, C) ←  all(J, X, Z, C).
all(J, X, Z, C) ←     delta(J, X, Z, C),
                      if(all(J, X, Z, C3) then C3 > C).
```

The `if-then` construct used in the last rule of Example 2, is a construct of $\mathcal{LDL}$ and $\mathcal{LDL}$++ whose formal semantics is defined by its re-writing into a negative program: our rule is equivalent to

```
all(J, X, Z, C) ←  delta(J, X, Z, C),
                   all(J, X, Z, C3),  C3 > C.
all(J, X, Z, C) ←  delta(J, X, Z, C),
                   ¬all(J, X, Z, C3).
```

Thus, programs containing if-then-else constructs are X-Y stratified iff their expansion using negation is. Likewise, the notion of least elements in a set can be defined by the property that no lesser element exists. Take for instance the following rule from Example 2:

```
delta(s(J), X, Z, min(< C >)) ←  delta(J, X, Y, C1),
                                 g(Y, Z, C2), C = C1 + C2,
```

Here $<$ C $>$ denotes the grouping of C values with respect to the other variables in the head; and min denotes the least of such values. The meaning of the previous rule can in fact be defined through the definition of set aggregates [?] or equivalently by its expansion using negation [?].

$$
\begin{aligned}
\texttt{delta(s(J),X,Z,C)} \leftarrow\ & \texttt{delta(J,X,Y,C1)},\ \texttt{g(Y,Z,C2)}, \texttt{C} = \texttt{C1} + \texttt{C2}, \\
& \neg\texttt{lesser(s(J),X,Z,C)}. \\
\texttt{lesser(s(J),X,Z,C)} \leftarrow\ & \texttt{delta(s(J),X,Z,C}'),\ \texttt{g(Y,Z,C2)}, \texttt{C}' = \texttt{C1} + \texttt{C2}, \\
& \texttt{C}' < \texttt{C}.
\end{aligned}
$$

The approach of establishing a one-to-one correspondence between certain syntactic constructs leading to efficient operational semantics and classes of non-monotonic programs with a formal semantics has far-reaching implications. For instance, in supporting a program such as that of Example 1, it is clear that rules r_2 and r_3 can be implemented efficiently via an imperative *append_to (insert)* operator. Conversely, database updates can often be viewed as a syntactic encapsulation for XY-stratified programs: we will expand on this approach in the next sections.

4 Semantics of Updates

Let us now consider a database language that, in addition to query requests, supports other commands, such as requests to add or delete some extensional facts. As shown in [?], the definition of the semantics of programs with updates need not make use of imperative constructs. Rather, defining the semantics of such a language tantamounts to defining the external behavior of programs written in this language. Neglecting for the moment integrity constraints, we see that the external response to an update command should basically be an acknowledgement of some sort (e.g., a carriage return). Thus, all it is left to do is to define the meaning of queries. However, there is a key difference with respect to standard framework of query-only logic-based semantics [?, ?]: here we must specify the answer to queries *after the database has been modified by a given sequence of updates.* Thus, in our formal model we have (i) a program P containing a set of rules and a schema describing the extensional database, (ii) a set of extensional facts D defining the initial database state (iii) a sequence of update requests R, and (iv) a query Q; then we must define the *meaning function* $M(P, D, R, Q)$. For instance, consider the following example

Example 3 *We assume that our program P contains the declaration of two database relations* std *and* grad *(describing the majors of students and the courses and grades they took) and the following rule*

$$\texttt{csst(X,C)} \leftarrow\ \texttt{std(X,cs)},\texttt{grad(X,C,\_)}.$$

The initial database D contains the following facts:

```
std(ann,ee).      grad(ann,cs143,3).
std(tom,cs).
```

R, the set of update requests, is:

```
req(1,add,std(marc,ee)).
req(2,del,std(ann,ee)).
req(2,add,std(ann,cs)).
```

The query Q is: `?csst(X, Y).`

We have represented our sequence of update requests as a relation `req`; the first argument in `req` places the particular request in the proper time sequence. Successive requests are given successive integers by the system. However, several requests can be given the same sequence number, to ensure that they are processed in parallel. For instance, the last two entries in R correspond to a user-level request to modify the major of Ann from EE to CS.

Inasmuch as we only need to define the external behaviour of the system, the definition of our meaning function reduces to the specification of the response (answer) to any query such as `?csst(X, Y)`, given a certain initial database and an arbitrary sequence of updates. Since a query can inquire about the content of any relation after a sequence of such updates, we will have to model the notion of states the database goes through; however, we must avoid destructive assignments in order to remain declarative and obtain a logic-based semantics. Toward this goal, we use a distinguished predicate `quevt` that, basically, operates as a queue of events. For now, the `quevt` predicate can be thought of as performing a copy of the `req` predicate as follows:

Example 4 *A first attempt at* `quevt`

```
quevt(N,ActionTyp,Atom,N) ←   req(N,ActionTyp,Atom).
```

The meaning of a program P with external updates is thus defined by generating an equivalent program P'. For each extensional predicate `q/n` (`q` is the name of the predicate and `n` is its arity) we now define a new intensional predicate `q/n+1` (we assume without loss of generality that there is no `q/n+1` in the original P.) These new predicates are defined recursively, by XY-stratified programs:

Example 5 *From extensional predicates to XY-stratified programs.*

```
std(0,X1,X2) ←         std(X1,X2).
std(J+1,X1,X2) ←       quevt(J+1,_,_,_), std(J,X1,X2),
                       ¬quevt(J+1,del,std(X1,X2),_).
std(J+1,X1,X2) ←       std(J,_,_), quevt(J+1,add,std(X1,X2),_).

grad(0,X1,X2,X3) ←     grad(X1,X2,X3).
grad(J+1,X1,X2,X3) ←   quevt(J+1,_,_,_), grad(J,X1,X,X3),
                       ¬quevt(J+1,del,grad(X1,X2,X3),_).
grad(J+1,X1,X2,X3) ←   grad(J,_,_), quevt(J+1,add,grad(X1,X2,X3),_).
```

Furthermore, the old rules of P are replaced with new ones, obtained from the old ones by adding a stage argument to every predicate in the rules:

Example 6 *Rewriting the original rules*

$$\mathtt{csst(J,X,C)} \leftarrow \quad \mathtt{std(J,X,cs),grad(J,X,C,\_)}.$$

The query goal `?sst(S, C)` is then modified in an obvious way. To find the proper answer to the query after the first request `req(1, add, std(marc, ee))`, we pose the query: `?csst(1, S, C)`. But, the correct answer to the same query after the next two requests have been serviced is produced by `?csst(2, S, C)`.

Thus, we replaced the old predicates with new ones containing an additional stage argument. For notational convenience we shall represent the stage as a superscript; thus instead of writing `std(J, X, cs)` we write $\mathtt{std^J(X,cs)}$. Thus a new program P' is constructed from the original one P by replacing the old rules of P with new ones where the predicates are stage-superscripted. Moreover, for each extensional predicate `q` of P, P' contains the following set of XY-stratified rules:

Example 7 *Updates modeled by rules*

$$\begin{aligned}
\mathtt{r_1 : q^0(\mathbf{X})} \leftarrow \quad & \mathtt{q(\mathbf{X})}. \\
\mathtt{r_2 : q^{J+1}(\mathbf{X})} \leftarrow \quad & \mathtt{quevt^{J+1}(_,_,_),\ q^J(\mathbf{X}),} \\
& \neg\mathtt{quevt^{J+1}(del,q(\mathbf{X}),_)}. \\
\mathtt{r_3 : q^{J+1}(\mathbf{X})} \leftarrow \quad & \mathtt{q^J(\mathbf{X}),\ quevt^{J+1}(add,q(\mathbf{X}),_)}.
\end{aligned}$$

These three rules will be called, respectively as follows: r_1 the *base rule*, r_2 the *copy-delete rule*, and r_3 the *add rule*. Then, the deletion-copy rule copies the old relation into a new one, modulo any deletion that is currently pending on the event queue `quevt`. The insert rule services the add requests currently pending in `quevt`. The base rule defines a derived predicate with stage value of zero, for each extensional predicate.[3]

The resulting program P' is XY-stratified and defines the meaning of the original program P. The correct answer to query $?\mathtt{q}(\mathbf{X})$ once all the $\mathtt{req^J}$ entries have been serviced is simply the answer to $?\mathtt{q^J}(\mathbf{X})$. For instance, with P, D and R defined in Example 4, the perfect model of our modified program P' contains the following derived facts:

Example 8 *The perfect model for P' (derived facts only)*

$\mathtt{std^0(tom,cs)}$	$\mathtt{grad^0(ann,cs143,3)}$
$\mathtt{std^0(ann,ee)}$	
$\mathtt{std^1(tom,cs)}$	$\mathtt{grad^1(ann,cs143,3)}$
$\mathtt{std^1(ann,ee)}$	

[3] We assume that initially our database relations are not empty. Otherwise, an additional exit rule, $\mathtt{p^0(\mathbf{nil})} \leftarrow \neg\mathtt{p}(\mathbf{X})$, can be added.

$\mathtt{std}^1(\mathtt{marc},\mathtt{ee})$		
$\mathtt{std}^2(\mathtt{tom},\mathtt{cs})$	$\mathtt{grad}^2(\mathtt{ann},\mathtt{cs143},3)$	$\mathtt{csst}^2(\mathtt{ann},\mathtt{cs143})$
$\mathtt{std}^2(\mathtt{marc},\mathtt{ee})$		
$\mathtt{std}^2(\mathtt{ann},\mathtt{cs})$		

A query, such as `?csst(S, C)`, is then changed into ?$\mathtt{csst}^2(\mathtt{S},\mathtt{C})$ and answered against such a perfect model.

This simple rendering of the semantics of updates captures one's intuitive understanding of these operations. It also is suggestive of efficient operational semantics. In fact, delete-copy rules can be implemented with the update-in-place policy, outlined for XY-programs, whereby records are simply added to, or deleted from, the current copy of the relation. The declarative semantics of these rules is, however, fully retained, as demonstrated by the fact that queries corresponding to update subsequences are also supported: it is also possible to pose queries such as $\mathtt{csst}^0(\mathtt{S},\mathtt{G})$ or $\mathtt{csst}^1(\mathtt{S},\mathtt{G})$.

Integrity constraints can also be treated in this framework. If the enforcement policy consists in rejecting any request that violates the constraint (e.g., rejecting a request for insertion of a new tuple violating a key constraint), then the proper checking conditions can be attached to the rule defining `quevt`. Policies where violations are corrected by additional actions (e.g., elimination of dangling foreign key references) can be supported using the condition-action rules or the event-action rules discussed next.

5 Condition-Action Rules

Say that we want to enforce a rule such as: If a student has taken both cs10 and cs20, then he or she is considered having CS as major. For that, we could write:

$$r_4 : \mathtt{add}(\mathtt{std}(\mathtt{S},\mathtt{cs})) \leftarrow \quad \mathtt{grad}(\mathtt{S},\mathtt{cs10},\_),\ \ \mathtt{grad}(\mathtt{S},\mathtt{cs20},\_).$$

Another possible rule could enforce a deletion dependency whereby one will want to delete the classes taken by students that are not longer enrolled. This can be accomplished as follows:

$$r_5 : \mathtt{del}(\mathtt{grad}(\mathtt{S},\mathtt{C},\mathtt{G})) \leftarrow \quad \mathtt{grad}(\mathtt{S},\mathtt{C},\mathtt{G}),\ \ \neg\mathtt{std}(\mathtt{S},\_).$$

The urgency for a formal semantics to supplement intuition is obvious even in this simple example. In fact, assume that a request is placed to add a cs20 record and a cs10 record in `grad` for a new student `adam`, as follows:

$\mathtt{req}(3,\mathtt{add},\mathtt{grad}(\mathtt{adam},\mathtt{cs10},4)).$
$\mathtt{req}(3,\mathtt{add},\mathtt{grad}(\mathtt{adam},\mathtt{cs20},3)).$

Then, according to intuition alone, each of the following alternatives appears plausible: (i) the addition of two initial `grad` records followed by that of a new cs student in std; (ii) the insertion of the two initial records immediately followed by deletion of all the courses this student has taken; (iii) the addition of the initial grad records followed by the parallel insertion of a record for the new student and the deletion of course records for this student; (iv) no action; or (v) an infinite loop. After the introduction of a formal semantics, only one of these alternatives will be considered correct

The semantics we propose for active rules, views del and add as built-in derived predicates. Thus these two rules are simply re-written as any other rule:

$$\begin{array}{ll} r'_4 : \mathtt{add^J(std(S,cs))} \leftarrow & \mathtt{grad^J(S,cs10,\_)},\ \mathtt{grad^J(S,cs20,\_)}. \\ r'_5 : \mathtt{del^J(grad(S,C,G))} \leftarrow & \mathtt{grad^J(S,C,G)},\ \neg\mathtt{std^J(S,\_)}. \end{array}$$

Furthermore, there is no change in the intensional update rules for the extensional predicates. However, the rules defining quevt must be extended to account for the add and del predicates as follows:

Example 9 *An improved definition for* quevt

$$\begin{array}{ll} \mathtt{quevt^{J+1}(add,W,N)} \leftarrow & \mathtt{quevt^J(\_,\_,N)},\ \mathtt{add^J(W)}. \\ \mathtt{quevt^{J+1}(del,W,N)} \leftarrow & \mathtt{quevt^J(\_,\_,N)},\ \mathtt{del^J(W)}. \\ \mathtt{quevt^{J+1}(X,W,N+1)} \leftarrow & \mathtt{quevt^J(\_,\_,N)},\ \neg\mathtt{add^J(\_)}, \neg\mathtt{del^J(\_)}, \\ & \mathtt{req^{N+1}(X,W)}. \end{array}$$

Thus, active rules will add to the quevt table. Once these rules have stopped firing, then new external requests from req can further expand the table.

After a sequence of n requests, the correct answer to query $?\mathtt{q}(\mathbf{X})$, is obtained by answering the following three goals

$$?\mathtt{quevt^J(\_,\_,n)}, \neg\mathtt{quevt^{J+1}(\_,\_,n)}, \mathtt{q^J}(\mathbf{X})$$

The first two goals find the highest value J reached by the stage variable after servicing all requests with sequence number n; then the values for the correct answer are derived from $\mathtt{p^J}(\mathbf{X})$.

For our previous example for instance, the request to add the cs10 and cs20 courses for adam (with respective grades 4 and 3) will generate the following "new" state:

$$\begin{array}{lll} \mathtt{std^3(tom,cs)} & \mathtt{grad^3(ann,cs143,3)} & \mathtt{csst^3(ann,cs143)} \\ \mathtt{std^3(marc,ee)} & \mathtt{grad^3(adam,cs10,4)} & \\ \mathtt{std^3(ann,cs)} & \mathtt{grad^3(adam,cs20,3)} & \end{array}$$

Then, both rules r'_4 and r'_5 are activated in this situation resulting in the appearance of $\mathtt{std^4(adam,cs)}$ and the disappearance of adam's cs10a and cs20 records in the final state (basically option (iii) among the alternatives above).

Condition-action rules are very powerful, and can be used in several applications, including constraint maintenance or truth-maintenance support [?]. On the other hand, condition-action rules can be expensive to support since they require the recomputation of the body of each rule every time a new update occurs in the base relations defining such a rule. Thus, every database predicate appearing in the body of an active rule must be monitored for changes; for derived predicates, possibly recursive ones, the derivation tree (dependency graph) must be traced down to the database predicates involved. While differential methods, such as the semi-naive fixpoint, and truth-maintenance techniques, can be exploited in this context, it is also clear that condition-action rules tend to be complex and expensive to support. For these reasons, more recent proposals favor an alternative approach where the events that can trigger the firing of the rules are stated explicitly in the bodies of the rules. This is discussed next.

6 Event-Action Rules

In systems such as Postgres [?], the events upon which a rule fires are stated explicitly. These rules can be easily modeled in our framework. For instance, the previous active rules involving students and courses could be expressed as follows:

Example 10 *Event-driven rules*

$$\begin{aligned}
&\mathtt{add(std(S,cs))} \leftarrow && \mathtt{add(grad(S,cs10,_)),\ grad(S,cs20,_).}\\
&\mathtt{add(std(S,cs))} \leftarrow && \mathtt{grad(S,cs10,_),\ add(grad(S,cs20,_)).}
\end{aligned}$$

$$\mathtt{del(grad(S,\_,\_))} \leftarrow \ \mathtt{del(std(S,\_)).}$$

These event-driven rules are easily supported in our framework. We basically interpret event-action rules as stating that, when `add` or `del` events are queued in the `quevt` relation, then, the `add` or `del` predicates are enabled (requested). Thus, the meaning of the previous rules is defined by the following re-writing:

Example 11 *Expansion of event-driven rules*

$$\begin{aligned}
&\mathtt{add^J(std(S,cs))} \leftarrow && \mathtt{quevt^J(add,grad(S,cs10,_),_),grad(S,cs20,_).}\\
&\mathtt{add^J(std(S,cs))} \leftarrow && \mathtt{grad(S,cs10,_)),quevt^J(add,grad(S,cs20,_),_).}\\
&\mathtt{del^J(grad(S,_,_))} \leftarrow && \mathtt{quevt^J(del,std(S,_),_).}
\end{aligned}$$

By the definition of `quevt`, these `add` and `del` requests queued at stage `J` will be executed at stage $J+1$.

Observe that since our event driven rules have been written to detect single events they will not be triggered by the contemporary request of inserting the two records cs10 and cs20 for adam. The final result therefore corresponds to alternative (i) out of those listed above.

7 Event-Based Programming

New events can be defined in addition to the basic `add`, `del` ones and used in various roles, including constraint management and application programming. As a (somewhat contrived) example, for instance, say that we want to raise the grades of Ann until her grades are all greater or equal to 4. This will be accomplished by the definition of a new `raise` event,

Example 12 *Raising the grades of students*

$$
\begin{array}{ll}
\mathtt{del(grad(S,C,G))} \leftarrow & \mathtt{evt(raise(S)), grad(S,C,G), G < 4.} \\
\mathtt{add(grad(S,C,G'))} \leftarrow & \mathtt{evt(raise(S)), grad(S,C,G), G < 4,} \\
 & \mathtt{G' = G + 1.} \\
\mathtt{evt(raise(S))} \leftarrow & \mathtt{evt(raise(S)), grad(S,C,G), G < 4.}
\end{array}
$$

followed by the request: `evt(raise(ann))`. These rules, with $\mathtt{S} = \mathtt{ann}$ are now executed in parallel enabling the corresponding set of events. These are then enqueued by the following `quevt` rules:

Example 13 *The final definition of* `quevt`

$$
\begin{array}{ll}
\mathtt{quevt^{J+1}(add, W, N)} \leftarrow & \mathtt{quevt^{J}(\_,\_,N),\ \ add^{J}(W).} \\
\mathtt{quevt^{J+1}(del, W, N)} \leftarrow & \mathtt{quevt^{J}(\_,\_,N),\ \ del^{J}(W).} \\
\mathtt{quevt^{J+1}(ev, W, N)} \leftarrow & \mathtt{quevt^{J}(\_,\_,N),\ \ ev^{J}(W).} \\
\mathtt{quevt^{J+1}(X, W, N+1)} \leftarrow & \mathtt{quevt^{J}(\_,\_,N),} \\
 & \mathtt{\neg add^{J}(\_),\ \ \neg del^{J}(\_),\ \ \neg evt^{J}(\_),} \\
 & \mathtt{req^{N+1}(X, W).}
\end{array}
$$

Then, event-action rules can be re-written as follows:

Example 14 *Raising the grades of students*

$$
\begin{array}{ll}
\mathtt{del^{J}(grad(S,C,G))} \leftarrow & \mathtt{quevt^{J}(evt, raise(S), \_), grad^{J}(S,C,G), G < 4.} \\
\mathtt{add^{J}(grad(S,C,G'))} \leftarrow & \mathtt{quevt^{J}(evt, raise(S), \_), grad^{J}(S,C,G), G < 4,} \\
 & \mathtt{G' = G + 1.} \\
\mathtt{evt^{J}(raise(S))} \leftarrow & \mathtt{quevt^{J}(evt, raise(S), \_), grad^{J}(S,C,G), G < 4.}
\end{array}
$$

The detection of an event condition evt(raise(S)) results in the checking of additional conditions and in the setting of a new event, including the resetting of the old event evt(raise(S)), as illustrated by the last rule above. All applicable rules are fired in parallel; it is thus possible to perform recursive programming, whereby the same action is repeated while the body conditions remain true. The action performed at each step can be a basic update, or some other action, including the invocation of a query or the printing of some results.

8 Conclusion

The semantic framework here proposed yields the design of a rule-based language capable of addressing both the active and deductive aspects of programming. For deductive rules, one can keep the basic framework of Horn Clauses, with non-monotonic extensions, including negation and aggregates under the XY-stratification assumption. Active rules can be specified with the same syntax, provided that `add, del, evt` are built-in predicates. A uniform perfect-model semantics is ensured by the re-writing methods just discussed, which do not rely on meta-level or higher order constructs.

Building on this semantic bedrock, the language designer can consider further improvements and structuring of the language to improve the efficiency and clarity of programs. For instance, practical considerations might suggest that condition-action rules should be disallowed; in this case, the language will only support two kinds of rules. The first kind of rules are deductive ones without any event predicate. The other kind consists of event-action rules: these are defined as having an event predicate in their head, and one or more positive event goals in their bodies. Syntactic sugaring conventions might also be used to improve the expressivity of the language. A simple improvement would allow rules with multiple heads as a short-hand for several rules with similar bodies. The previous example, for instance, could be abbreviated as follows:

Example 15 *A multi-head rule.*

$$\begin{array}{ll} \mathtt{del(grad(ann,C,G))}, & \\ \mathtt{add(grad(ann,C,G1))}, & \\ \mathtt{evt(raise(S))} \leftarrow & \mathtt{evt(raise(S))}, \\ & \mathtt{grad(S,C,G)}, \mathtt{G} < 4, \mathtt{G1} = \mathtt{G} + 1. \end{array}$$

Also observe that that within the power and uniformity provided by such a language, there will be specialized usages. For instance, a system administrator will be predominantly concerned with monitoring events such such as `add` and `del`. However, application programmers will be mostly interested in defining new event types that will be invoked by certain classes of users. Each such application is defined by an event that is invoked by a user request or triggered by other applications. Each application can call itself recursively, or can take various actions, including calling other applications.

A new rule-based language incorporating these principles is currently being designed at UCLA.

References

[1] S. Abiteboul and V. Vianu. Datalog extensions for database queries and updates. *Journal of Comp. and System Sc.*, 43(1):62–124, August 1991.

[2] Apt, K., and J.M. Pugin, "Maintenance of stratified databases viewed as a belief revision system", *ACM PODS*, 1987.

[3] Bry, F., Intensional updates: abduction via deduction, in: *Proc. 7th Int. Conf. on Logic Programming*, Jerusalem, 561-575, 1990.

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

[5] Chimenti, D. et al., "The $\mathcal{LDL}$ System Prototype," *IEEE Journal on Data and Knowledge Engineering*, vol. 2, no. 1, pp. 76-90, March 1990.

[6] Fagin, R.., Kuper, G., D.Ullman and M.Y.Vardi, "Updating logical databases", *Advances in Comp.Res.*, vol.3, 1-18, JAI Press Inc., 1986.

[7] Harel, D., "Dynamic logic", in *Handbook of Philosophical Logic*, (Gabbay and Guenther, eds.), D.Reidel Publishers, 1983.

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

[9] M. Gelfond and V. Lifschitz. The stable model semantics of logic programming. *Proceedings of the Fifth Intern. Conference on Logic Programming*, pages 1070–1080, 1988.

[10] S. Ganguly, S. Greco, and C. Zaniolo. *Minimum and Maximum Predicates in Logic Programming. Proceedings of the Tenth ACM Symposium on Principles of Database Systems*, pp. 154–113, 1991.

[11] F. Giannotti, D. Pedreschi, D. Saccà, and C. Zaniolo. Nondeterminism in deductive databases. *Proc. 2nd Int. Conf. on Deductive and Object-Oriented Databases*, 1991.

[12] A. Van Gelder, K.A. Ross, and J.S. Schlipf. The well-founded semantics for general logic programs. *Journal of ACM*, 38(3):620–650, 1991.

[13] Katzuno, H. and A.O. Mendelzon, Propositional knowledgebase revision and minimal change, *Artificial Intelligence*, 52, 263-294, 1991.

[14] P.G. Kolaitis and C.H. Papadimitriou, Why not negation by fixpoint?, *JCSS*, 43(1), 125-144, 1991.

[15] Krishnamurthy, R., Naqvi, S. and C. Zaniolo, "Database Updates and Transactions in $\mathcal{LDL}$", *Procs. of 1989 North American Conference on Logic Programming*, MIT Press, 1989.

[16] Gelfond, M. and Lifschitz, V., "Representing Actions in Extended Logic Programming," *Proc. Joint Int. Conf-Symp on Logic Programming*, 1992, MIT Press, 1992.

[17] Lifschitz, V. "Formal Theories of Action" in: F.M. Brown, ed. *The Frame Problem in Artificial Intelligence*, Proc. 1987 Workshop, Morgan Kaufman, Los Altos, CA, 1987.

[18] Lloyd, J.W., *Foundations of Logic Programming,*, Springer Verlag, 1977.

[19] Manchanda, S. and D.S. Warren, "Towards a logical theory of database view updates", *Int.Worksh. on Foundations of Deductive databases and Logic Programming*, J.Minker ed., Aug. 1988.

[20] D. McCarty and U. Dayal. The architecture of an active database management system. In *ACM SIGMOD International Conf. on Management of Data*, pages 215–224, 1989.

[21] M. Morgenstern. Active databases as a paradigm for enhanced computing environments. In *Ninth International Conf. on Very Large Data Bases, Florence*, pages 34–42, 1983.

[22] L. Palopoli and R. Torlone. Specifying the dynamics of complex object databases. In *4th Int. Workshop on Foundations of Models and Languages for Data and Objects – Modeling Database Dynamics*, pp. 143–160. Springer-Verlag, 1992.

[23] Phipps, G., M.A., Derr and K. A. Ross, "Glue-Nail: a Deductive Database System," *Proc. 1991 ACM–SIGMOD Conference on Management of Data*, pp. 308-317 (1991).

[24] Przymusinski, T.C. "Every logic program has a natural stratification and an iterated fixed point model", in *PODS 1989.*

[25] S. A. Naqvi, S. Tsur *"A Logical Language for Data and Knowledge Bases"*, W. H. Freeman, 1989.

[26] Ramakrishan, R., Srivastava, D. and Sudarshan, S., "*CORAL*: A Deductive Database Programming Language," Proc. VLDB'92 Int. Conf, pp. 238-250, 1992.

[27] Reiter, R., "On Formalizing Database Updates: Preliminary Report," in, *Advances in Database Technology–EDBT'92*, (Pirotte, Delobel, Gottlob, eds.), Springer Verlag, 1992

[28] M.L. Stonebraker, A. Jhingran, J. Goh, and S. Potamianos. On rules, procedure, caching and views in data base systems. In *ACM SIGMOD International Conf. on Management of Data*, pages 281–290, 1990.

[29] van Emden M.H. and R.A. Kowalski, "The Semantics of Predicate Logic as a Programming Language," *J.ACM 23*, 4 (Oct. 76), 67-75.

[30] Warren, D.S., Database Updates in Pure Prolog, *Proc. Int. Conf. on Fifth Generation Computer Systems*, 244-253, 1985.

[31] J. Widom and S. Finkelstein. Set-Oriented production rules in relational database systems. In *ACM SIGMOD International Conf. on Management of Data*, pages 259–270, 1990.

[32] M. Winslett, "A model-theoretic approach to updating logical databases", *ACM PODS*, 1986.

[33] Zaniolo, C., *Intelligent Databases: Old Challenges and New Opportunities*, Journal of Intelligent Information Systems, 1, 271-292 (1992).

[34] Zaniolo, C., N. Arni, K. Ong, "Negation and Aggregates in Recursive Rules: the $\mathcal{LDL}$++ Approach", *Proc. 3rd Int. Conference on Deductive and O-O DBs, DOOD-93*, Phoenix, AZ, Dec 6-8, 1993.