A Simple Model for Active Rules and their Behavior in Deductive Databases

Carlo Zaniolo and **Reza Sadri**
Computer Science Department
University of California
Los Angeles, CA 90024
zaniolo@cs.ucla.edu reza@cs.ucla.edu

Abstract

Recent advances in non-monotonic semantics of deductive databases provide a simple framework for modeling the even-condition-action rules of active databases. This approach unifies the semantics of active and deductive databases and yields several benefits. In particular it can be used to model the semantics of different active databases and to perform termination analysis for active rules.

1 Introduction

Rules provide the main construct for expressing computation in active databases and deductive databases. The unification of these two database disciplines represents a research problem of obvious theoretical interest and many important applications could benefit from it; yet, there has been little formal work on 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 [1, 3, 8, 9, 14]. However, there remains a dire need for clarifying and formalizing the semantics of these systems [18, 19]. The situation for deductive databases appears to be just the opposite, since not one but three equivalent formal semantics have been developed for Horn clauses [16, 7]. Unfortunately, this elegant semantics are brittle and not easily generalized to 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: for instance, reasoning about changes and actions remains one of the classical challenges of AI [6].

Given this rather troubled 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.

1.1 XY-Stratification

We begin with a simple example based on what is arguably the oldest procedure in the world: the Sieve of Eratosthenes. The idea is to generate inductively each successor integer, and check if it divisible by a prime number previously generated. If such a divisor is found, then the new integer is classified as "unprime" otherwise it is classified as "prime". We can write the following recursive program with negation ($\mathtt{div(J+1,K)}$ is true when integer J+1 is divisible by integer K):

$$
\begin{array}{ll}
\mathtt{prime(2).} & \\
\mathtt{unprime(J+1)} \leftarrow & \mathtt{int(J),\ prime(K),\ div(J+1,K).} \\
\mathtt{prime(J+1)} \leftarrow & \mathtt{int(J),\ \neg unprime(J+1).} \\
\mathtt{int(I)} \leftarrow & \mathtt{prime(I).} \\
\mathtt{int(I)} \leftarrow & \mathtt{unprime(I).}
\end{array}
$$

The computation begins with the first rule (a non-recursive rule or an exit rule in our parlance), which derives that 2 is a prime number. The second rule, a recursive rule, is examined next. This fails since 3 not divisible by 2; thus the the third rule succeeds and asserts `prime(3)`. The remaining two rules produce `int(3)`. Having so completed a first pass through the rules, we can now see that the second rule fires, since $3 + 1$ is divisible by the argument of `prime(3)`, yielding `unprime(4)`; from that, rule r_6 computes `int(4)` and the inductive computation continues. This program has unique well-founded model due to the fact that $\mathtt{div(J+1,K)}$ can only be satisfied when $J+1 \geq K$. (If we only consider instances of the rules that satisfy this constraint we obtain a locally stratified program—thus the original program is modularly stratified).

Unfortunately, to detect the well-formedness of this program, a compiler must "understand" the properties of predicates, such as `div`, which are used in the recursive rules—a task that is in general undecidable, if we let `div` be defined by arbitrary program rules. For these reasons, there has been much recent interest in finding classes of recursive programs where the existence of a well-founded model or other canonical semantics can be decided at compile-time on the basis of the syntactic properties of the program itself.

The concept of XY-stratification [20] provides a simple and general solution to this problem. The basic idea is to rely on a particular argument (say the first argument) in recursive predicates as the main enforcer of the stratification. Then only two kinds of rules are allowed:

- **X-rule**: when all recursive predicates in the rule share the same simple variable, say J, as their first arguments:
- **Y-rule** when one or more of the recursive predicates in the body of the rule share a simple variable, say J, as their first argument, while the remaining recursive predicates have $J + 1$ as their first argument.

For instance, the following is a XY-stratified version of the Sieve:

$$
\begin{array}{lll}
\mathtt{r_1 : prime(2,2).} & & \\
\mathtt{r_2 : unprime(J+1)} \leftarrow & & \mathtt{int(J), prime(J,K),\ div(J+1,K)} \\
\mathtt{r_3 : prime(J+1,J+1)} \leftarrow & & \mathtt{int(J), \neg unprime(J+1).} \\
\mathtt{r_4 : prime(J+1,K)} \leftarrow & & \mathtt{prime(J,K).} \\
\mathtt{r_5 : int(I)} & \leftarrow & \mathtt{prime(I,\_).} \\
\mathtt{r_6 : int(I)} & \leftarrow & \mathtt{unprime(I).}
\end{array}
$$

Here r_5, r_6 are X-rules while r_3, r_4 are Y-rules. The rule r_1 is exit rule for this recursive clique[1] Observe that the one-argument `prime` has been changed into a two-argument predicate, and that r_4 is a copy rule that accumulates all the prime numbers found so far.

All the recursive rules in recursive clique must be X-rules or Y-rules. Then the recursive predicates in such a rule is adorned as follows. All recursive predicates in X-rules are adorned with the label "new". In the Y-rules instead, only recursive predicates with first argument $J+1$ are adorned with the label "new" and the first arguments: all other recursive predicates are instead adorned with "old".

Definition 1 *Let C be a recursive clique with negation. Then C is said to be XY-stratified if:*

- *all the recursive rules of C are either X-rules or Y-rules*
- *the adorned version of C is non-recursive*
- *all exit rules must have as first argument the same constant.*

Let us assume that the the symbol $+$ in the first argument is an infix function symbol; thus the number "2" in the example is only a short-hand for $0+1+1$. Then we have the following theorem[20]:

Theorem 1 *Each XY-stratified clique Q is locally stratified.*

Given the fact that the adorned version of the recursive clique is non-recursive it follows that the local stratification is *strict* in the sense that the head of each rule instance belongs to a stratum strictly higher than those of the goals. Therefore, one application of the immediate consequence operator yields the fixpoint at each stratum; then, the iterated fixpoint [11] that computes the perfect model of a locally stratified program is no longer transfinite.

Implementing XY-stratified programs is simple. The compiler must build the adorned dependency graph for each recursive clique defined using negation. The compiler must then check that the adorned dependency graph is acyclic, and that all the exit rules of the clique share the same constant as their first argument. Having so verified XY-stratification, the compiler proceeds with the generation of a computation plan. Two copies of each predicates are kept: the "new" copy and the "old" copy. The computation of the new copies from the old ones follows the bottom-up order specified by the adorned dependency graph. For the example at hand, for instance, the sequence of computation is determined by the following dependencies:

$\mathtt{int}_{\mathtt{old}}, \mathtt{prime}_{\mathtt{old}}, \rightarrow \mathtt{unprime}_{\mathtt{new}}$
$\mathtt{int}_{\mathtt{old}}, \mathtt{unprime}_{\mathtt{new}} \rightarrow \mathtt{prime}_{\mathtt{new}}$
$\mathtt{prime}_{\mathtt{old}} \rightarrow \mathtt{prime}_{\mathtt{new}}$
$\mathtt{prime}_{\mathtt{new}} \rightarrow \mathtt{int}_{\mathtt{new}}$
$\mathtt{unprime}_{\mathtt{new}} \rightarrow \mathtt{int}_{\mathtt{new}}$

Once every "new" copy has been computed, then the new and old copies are switched in their role and the computation resumes. In fact, observe that during this computation

[1]By *recursive clique* we mean the set of rules defining a maximal set of mutually recursive predicates. Non-recursive rules defining recursive clique predicates are called *exit* rules.

we need not keep the value of J explicitly stored in the relations: this value can be stored in a separate cell and increased by one when the new and old copy are switched. Several refinements make this computation efficient [20]. For instance, rule r_4 performs an identity copy; thus, if rule r_4 is computed before r_3, all it is needed is to let the pointers to the old copy and new copy point to the same relation, and rule r_4 is computed with zero cost.

The general algorithm for computing XY-stratified cliques is as follows (the notation w.f.a. will denote a rule without their first arguments) [20]:

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.f.a), once

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

While, XY-stratification is syntactically strict, many computations of interest as expressed naturally using it. For instance the following well-known program is XY-stratified that is compiled and computed correctly as just described:

```
even(0).
even(s(J)) ←  ¬even(J).
```

2 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 [13], 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 the 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 new line followed by a prompt). 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 [7, 16]: 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 1 *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*

```
csst(X, C) ←  std(X, cs), 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 2 *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 3 *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 4 *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 $\mathtt{q}$ of P, P' contains the following set of XY-stratified rules:

Example 5 *Updates modeled by rules*

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

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.[2]

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

$\mathtt{std^0(tom,cs)}$	$\mathtt{grad^0(ann,cs143,3)}$	
$\mathtt{std^0(ann,ee)}$		
$\mathtt{std^1(tom,cs)}$	$\mathtt{grad^1(ann,cs143,3)}$	
$\mathtt{std^1(ann,ee)}$		
$\mathtt{std^1(marc,ee)}$		
$\mathtt{std^2(tom,cs)}$	$\mathtt{grad^2(ann,cs143,3)}$	$\mathtt{csst^2(ann,cs143)}$
$\mathtt{std^2(marc,ee)}$		
$\mathtt{std^2(ann,cs)}$		

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

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.

3 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 need 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:

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

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 two `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:

$$\begin{array}{ll}r'_4 : \mathtt{add}^{\mathtt{J+1}}(\mathtt{std}(\mathtt{S},\mathtt{cs})) \leftarrow & \mathtt{grad}^{\mathtt{J}}(\mathtt{S},\mathtt{cs10},\_),\ \ \mathtt{grad}^{\mathtt{J}}(\mathtt{S},\mathtt{cs20},\_).\\ r'_5 : \mathtt{del}^{\mathtt{J+1}}(\mathtt{grad}(\mathtt{S},\mathtt{C},\mathtt{G})) \leftarrow & \mathtt{grad}^{\mathtt{J}}(\mathtt{S},\mathtt{C},\mathtt{G}),\ \ \neg\mathtt{std}^{\mathtt{J}}(\mathtt{S},\_).\end{array}$$

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 7 *An improved definition for* quevt

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

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 ?q($\mathbf{X}$), is obtained by answering the following three goals

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

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 $\mathtt{q}^{\mathtt{J}}(\mathbf{X})$.

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:

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

Then, both rules r'_4 and r'_5 are activated in this situation resulting in the appearance of $\mathtt{std}^4(\mathtt{adam},\mathtt{cs})$ and the disappearance of adam's cs10 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 many applications; however they 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.

4 Event-Action Rules

In systems such as Postgres [14], the events upon which a rule fires are stated explicitly. For instance, the previous active rules involving students and courses could be expressed as follows:

Example 8 *Event-driven rules*

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

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 9 *Expansion of event-driven rules*

$$\begin{array}{ll} \mathtt{add^{J+1}(std(S,cs))} \leftarrow & \mathtt{quevt^{J}(add,grad(S,cs10,\_),\_),\ \ grad^{J}(S,cs20,\_).} \\ \mathtt{add^{J+1}(std(S,cs))} \leftarrow & \mathtt{grad^{J}(S,cs10,\_)),quevt^{J}(add,grad(S,cs20,\_),\_).} \\ \mathtt{del^{J+1}(grad(S,\_,\_))} \leftarrow & \mathtt{quevt^{J}(del,std(S,\_),\_).} \end{array}$$

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

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 (ii) out of those listed above.

5 Termination Analysis

Let us now consider the following example borrowed from[18]. We have two relations wire, and wire-type as follows:

```
wire(wire-id, from, to, type, voltage, power)
wtype(type, cross-section, max-voltage, max-power)
```

Following[18] we use condition-event rules to enforce the following integrity constraint: no wire can be laid between two nodes if the resulting voltage or power exceeds that specified for such wire type. A tuple that violate the specs, must automatically be replaced with another wire which passes specification: if no such wire exists then the offending wire is simply removed, and no substitute is added in its place. Concretely, we use the following rules to specify that wires that violate their specs are removed:

$$\begin{array}{ll} \mathtt{r_0 : okwire(Type,Volts,Watts)} \leftarrow & \mathtt{wtype(Type,MaxV,MaxW),Volts \leq MaxV,Watts \leq MaxW.} \\ \mathtt{r_1 : del(wire(Type,X,Y,V,W))} \leftarrow & \mathtt{wire(Type,X,Y,V,M),\neg okwire(Type,V,W).} \end{array}$$

When a different wire type is available that can take the required load, we add this in lieu of the other (when several such replacements are available, we choose that with the lowest voltage rating):

$$\begin{array}{ll} \mathtt{r_2 : add(wire(Id,X,Y,Typ1,V,W))} \leftarrow & \mathtt{wire(Id,X,Y,Typ,V,W),\neg okwire(Typ,V,W),} \\ & \mathtt{wtype(Typ1,\_,V1,W1),okwire(Typ1,V,W),} \\ & \mathtt{\neg lesser(V,W,V1).} \\ \mathtt{r_3 : lesser(V,W,V1)} \leftarrow & \mathtt{wtype(Typ,\_,V2,W2),V \leq V2,W \leq W2,V2 \leq V1.} \end{array}$$

Say for instance that in a certain state, $k-1$, we have the following tuples:

$\texttt{wire}^{k-1}\texttt{(1, n}_0\texttt{, n}_1\texttt{, type}_1\texttt{, 100, 100)}$
$\texttt{wtype}^{k-1}\texttt{(type}_1\texttt{, 2, 150, 250)}$
$\texttt{wtype}^{k-1}\texttt{(type}_2\texttt{, 2, 250, 150)}$

And a request is made to introduce a new tuple as follows:

$$\texttt{req(1, add, wire(1, n}_1\texttt{, n}_2\texttt{, type}_1\texttt{, 200, 200))}$$

The execution of this insert, produces a databases which contains the old tuples above (with superscript k) and the following new tuple:

$$\texttt{wire}^{k}\texttt{(1, a}_0\texttt{, a}_1\texttt{, type}_1\texttt{, 200, 200)).}$$

Since this wire exceeds the voltage rating for $\texttt{type}_1$, it is not an okwire, and therefore causes the following instantiation of rule r_1 to fire:

$$\begin{aligned}\texttt{del}^{\texttt{k+1}}(\texttt{wire}(1,\texttt{n}_1,\texttt{n}_2,\texttt{type}_1,200,200) \leftarrow\ & \texttt{wire}^{\texttt{k}}(1,\texttt{n}_1,\texttt{n}_2,\texttt{type}_1,200,200),\\ & \neg\texttt{okwire}^{\texttt{k}}(\texttt{type}_1,200,200).\end{aligned}$$

Given the current database, however, rule r_2 does not fire since the search for a wire type that can stand both a voltage and power of 200 fail. Thus, the chain of events started by the insertion of the wire comes to a quick end. However, one must wonder, whether there can be situations where the insertion of a tuple triggering the firing of rule r_2, causes an insertion of a new tuple that fires a new instance of rule r_2,..., leading to an infinite chain of events. In a nutshell, we need to perform the termination analysis on our rule set.

Termination of active rule set is undecidable in general, but simple situations like the one above can be studied with relative ease. Indeed, for rule r_2 to fire in state k the following two goals must be both satisfied:

$$\leftarrow\quad \texttt{wire}^{\texttt{k+1}}(\texttt{Id},\texttt{X},\texttt{Y},\texttt{ATyp},\texttt{V},\texttt{W}),\neg\texttt{okwire}^{\texttt{k+1}}(\texttt{ATyp},\texttt{V},\texttt{W})$$

Now, using SLD resolution, we conclude (See Example 5) that the first goal is true only if: $\texttt{wire}^{\texttt{k}}(\texttt{Id},\texttt{X},\texttt{Y},\texttt{ATyp},\texttt{V},\texttt{W})$ holds (i.e., an old wire not being ok) or $\texttt{quevt}^{\texttt{k+1}}(\texttt{add}(\texttt{wire}(\texttt{Id},\texttt{X},\texttt{Y},\texttt{ATyp},\texttt{V},\texttt{W})))$ holds (i.e., the new wire not being ok). Since we are only interested in the second case, we search for a rule whose head unifies with the goal $\texttt{quevt}^{\texttt{k+1}}(\texttt{add}(\texttt{wire}(\texttt{Id},\texttt{X},\texttt{Y},\texttt{ATyp},\texttt{V},\texttt{W})))$ and find (see the first rule in Example 7) that we need to prove $\texttt{add}^{\texttt{k+1}}(\texttt{wire}(\texttt{Id},\texttt{X},\texttt{Y},\texttt{ATyp},\texttt{V},\texttt{W})))$. This last goal unifies only with the head of the following rule:

$$\begin{aligned}\texttt{add}^{\texttt{k+1}}(\texttt{wire}(\texttt{Id},\texttt{X},\texttt{Y},\texttt{ATyp},\texttt{V},\texttt{W})) \leftarrow\ & \texttt{wire}^{\texttt{k}}(\texttt{Id},\texttt{X},\texttt{Y},\texttt{Typ},\texttt{V},\texttt{W}),\neg\texttt{okwire}^{\texttt{k}}(\texttt{Typ},\texttt{V},\texttt{W}),\\ & \texttt{wtype}^{\texttt{k}}(\texttt{ATyp},\_,\texttt{V1},\texttt{W1}),\texttt{okwire}^{\texttt{k}}(\texttt{ATyp},\texttt{V},\texttt{W}),\\ & \neg\texttt{lesser}^{\texttt{k}}(\texttt{V},\texttt{W},\texttt{V1}).\end{aligned}$$

Therefore, we have expanded the original pair of goals into a set of several goals as follows:

$$\leftarrow ..., \texttt{okwire}^{\texttt{k}}(\texttt{ATyp},\texttt{V},\texttt{W}), ..., \neg\texttt{okwire}^{\texttt{k+1}}(\texttt{ATyp},\texttt{V},\texttt{W})$$

Now, the goal okwire only depends on wtype that remained the same in state $k+1$ as it was in state k; thus the two goals contradict each-other. We conclude that the effect of a wire insertion will not propagate beyond one step—a conclusion that holds independent of the database state.

This simple example also demonstrates the complexity of the problem at hand. For instance, the conclusion might not hold if we assume that the `wtype` is also updated along with the `wire`. Also this conclusion might not hold if two separate rules are used one for finding a replacement that satisfies the power rating and another for a replacement that satisfies the voltage rating. This situation is in fact discussed in [18] and resolved with techniques similar to those previously discussed (but using the more cumbersome framework of relational algebra).

A related problem is that of determinacy. Here we have implicitly assumed that the wire that minimally satisfies the power and voltage constraints is unique. If that is not the case, there is a question of which (if not all) replacements will actually be used. A non-deterministic outcome might be acceptable here.

While this paper proposes no general solutions to these problems, we claim that the logic based framework we have described is simpler and more conducive to solutions than the framework of relational algebra used by previous authors[18]. In particular, if we do not use function symbols (except in the first argument) we obtain a class of programs known as Datalog$_{1S}$ programs that have been studied extensively in context of temporal databases[10]. In particular it is known that computation of these programs terminates or is eventually periodic.

6 Conclusion

In this paper we have provided a unified solution to two important problems of deductive databases: one is that of negation and aggregates in recursive predicates, the other is that of semantics for database updates. The solution is based on the concept of XY-stratification, which leads to programs similar to the Datalog$_{1S}$ programs that have been used in the context of temporal databases[10]. While our approach is similar to other approaches in modeling state changes through frame axioms[13, 12], it goes well beyond these inasmuch as it builds a bridge between deductive databases, active databases and temporal databases. The proposed approach is flexible and can handle both condition-action and event-condition-action rules. We are currently focusing our research in this area which appears to be in dire need for better semantic foundations.

References

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

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

[3] 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.

[4] 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.

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

[6] 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.

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

[8] 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.

[9] 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.

[10] J. Chomicki, "Polynomial-time Computable Queries in Temporal Deductive Database Systems," *PODS* 1990.

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

[12] G. Lausen and B. Ludascher, "Updates by Reasoning About States" University of Hamburg Report, 1, 1994.

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

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

[15] V.S. Subrahamanian and C. Zaniolo, "Database Updates and AI Planning Domains," submitted for publication.

[16] 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.

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

[18] Baralis E., Ceri S., Widom J., Bettr Termination Analysis for Active Databases, 1993.

[19] 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.

[20] 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.