A Logical Framework for Active Rules

Institut für Informatik — Report 78*

Bertram Ludäscher *Ulrich Hamann*
Georg Lausen
Institut für Informatik, Universität Freiburg
Am Flughafen 17, 79110 Freiburg, Germany
{ludaesch,hamann,lausen}@informatik.uni-freiburg.de

Abstract

We present a state-oriented extension to Datalog called *Statelog* which comprises two kinds of rules within a simple, yet flexible unified framework, i.e. (i) passive *query rules* used to express queries and static integrity constraints, and (ii) active *transition rules* for defining state-changing operations. We show that Statelog is powerful enough to capture many typical active rule applications (including complex updates, integrity enforcement and incremental view maintenance strategies) while retaining a declarative, set-oriented, and deterministic semantics thus reconciling active and deductive databases. Using the Statelog framework, formal results on termination, expressiveness and complexity of active rules are established which carry over to related state-oriented languages like XY-stratified Datalog. We prove that it is undecidable for general Statelog programs whether a program terminates on all databases. We then develop syntactic criteria which guarantee program termination and investigate the impact of these restrictions on expressive power. The class of Δ-*monotone programs* extends previous results on Statelog and is of practical importance since it includes self-triggering active rules and expresses all Fixpoint transformations. In addition, our syntactic conditions yield an alternative static analysis technique based on the logical notion of monotony whereas previous approaches use execution and triggering graphs.

Keywords: theory, active databases, deductive databases

Proc. 7th Intl. Conference on Management of Data (COMAD), Pune, India, December 1995, Tata-McGraw Hill

1 Introduction

It is folk knowledge that active rules are more powerful and expressive than (passive) deductive rules. While Datalog-like query languages provide concise and elegant representation of extensional and derived data, they lack means to specify dynamic aspects like the definition of updates and reactive behaviour. Active rules, on the other hand, support automatic triggering of data manipulating operations as a response to external or internal events. Applications for active rules range from traditional trigger mechanisms as provided by commercial systems like Ingres, Sybase and Oracle, to specification and implementation of advanced DBMS functionality (checking and enforcing integrity constraints, incremental view maintenance, (view) updates, access control etc.) or even application-specific monitoring tasks. Not surprisingly, active rules have attracted and stimulated an immense body of research which led to an ever increasing number of active DBMSs (a very incomplete list is [4, 5, 7, 8, 13, 12, 16, 17, 28, 27]).

Although it is intuitively clear, that additional expressive power of active vs. deductive rules stems from the ability to (i) define actions which modify the state of the database and (ii) monitor changes in the database, relatively little is known on the precise impact of (i) and (ii) on the expressive power of rules. Moreover, a serious problem often referred to in the context of active rules is the difficult predictability of rule effects and the lack of declarative and intuitive semantics. Indeed, unless simpler semantic frameworks are available *"... rule designers will have to be four-star wizards in order to ensure that the rule system does what they actually want"* [31]. We address both issues, by using *Statelog* — a "minimal" extension of stratified Datalog by states — as a logical language core for ECA (Event-Condition-Action) rules. Statelog has been proposed in [22] as a declarative update language. In the present paper, the Statelog framework is adapted to active rules which allows to investigate important properties like termination, expressiveness and complexity in the context of active rules.

First, we show by means of illustrative examples that Statelog is powerful enough to capture many typical ECA rule applications (such as integrity enforcement and incremental view maintenance or (view) updates) while retaining a declarative, set-oriented and deterministic semantics.

The major theoretical contribution are formal results on expressiveness, complexity and termination of active rules which also give new insights into the closely related *XY-stratified Datalog* programs [33]. We prove that it is undecidable for general Statelog programs whether a program terminates on all databases (i.e. there can be no compile-time check capturing all terminating programs). We then develop syntactic criteria which guarantee program termination and investigate the impact of these restrictions on expressive power. The class of Δ-*monotone programs* is of particular interest since it includes self-triggering recursive rules. Our syntactic conditions yield an alternative static analysis technique based on the logical notion of monotony whereas previous approaches use execution and triggering graphs [5].

The paper is structured as follows. In section 2 we introduce the basic ideas of our approach and illustrate them by means of examples from active databases. Section 3 presents the logical framework of Statelog and formal results on termination and expressiveness of active Statelog rules. In section 4, a common architecture for active and deductive databases is proposed based on Δ-monotone rules. After comparing our approach to related work in section 5, we give a conclusion and outlook in the last section.

2 Statelog by Example

A distinctive feature of active rule languages is the ability to refer to different database states and delta relations. In contrast to query processing where a "one-state logic" is sufficient, data manipulating operations require access to more than one state. Therefore, we extend Datalog with the ability to refer to different database states. The resulting language *Statelog* [22] corresponds to a (very restricted) temporal logic and is close in spirit to the temporal query languages *Templog* and *Datalog$_{1S}$* [6] and to *XY-stratified Datalog* [33] (cf. section 5).

The basic execution model of Statelog is as follows. Given the current state S_0 of a database, non-state-changing *query rules* are used to query the contents of the database — they are identical to the usual (stratified) Datalog rules for defining intensional relations (IDB). Triggered by the occurrence of one or more external events, *transition rules* become activated and perform the actions which are defined in the rule head provided the condition given by the body of the rules is satisfied. This results in a sequence $S_1, S_2, \ldots$ of intermediate (virtual) states. Computation terminates if a fixpoint is reached, i.e. if two subsequent states are equal. In a transition rule, every Datalog literal is extended with a *state term* of the form $[S+k]$ (where S is a state variable and k is a natural number) defining the state under consideration. In order to illustrate the above notions, consider the following example.

Example 2.1 (Enforcing Referential Integrity)
Given relations emp(EmpName, Salary, DeptName) and dept(DeptName, MgrName) with the obvious meaning, we want to express that whenever managers are deleted from **emp**, then all employees in the departments managed by these are also deleted, along with the departments themselves. Using the framework of [32] the following active rule can be used:

```
when deleted from emp
then delete from emp
     where DeptName in
          (select DeptName from dept
          where MgrName in
               (select EmpName from deleted emp));
     delete from dept
     where MgrName in
          (select EmpName from deleted emp)
```

The logical counterpart is represented in Statelog as follows:[1]

$$\begin{array}{l} [S+1]\ (\mathsf{del{:}emp(E1,Sal1,D)},\ \mathsf{del{:}dept(D,E)}) \leftarrow \\ \quad [S]\ (\mathsf{del{:}emp(E,\_,\_)},\ \mathsf{dept(D,E)}),\ [S+1]\ \mathsf{emp(E1,Sal1,D)}. \end{array} \tag{2.1}$$

Assume an initial database state S_0 as depicted in figure 1. Assume further that in S_0 there is a request to delete the manager md1 of department d1, i.e. $[S_0]$ del:emp(md1,10000,d1)

[1]For notational convenience, we "factor out" common state terms, i.e. $[S](A,B) \Leftrightarrow [S]A, [S]B$. Similarly, a rule $(A,B) \leftarrow C$ with multiple heads stands for two rules $A \leftarrow C$ and $B \leftarrow C$.

holds. This triggers rule (2.1) for the first time which results in the transition to the intermediate state S_1. A *frame rule*

$$[S+1]\ \mathsf{emp(E,Sal,D)} \leftarrow [S]\ (\mathsf{emp(E,Sal,D)},\ \neg\ \mathsf{del{:}emp(E,Sal,D)})\ . \tag{2.2}$$

propagates emp tuples which are not deleted to subsequent states (a corresponding rule is used for dept). In this case, employee md1 is removed in S_1 while the activated rule (2.1) raises new requests for the deletion of the employees of d1 as well as for the department d1 itself. The former request includes the deletion of md2 who is also the manager of department d2. Consequently, this triggers rule (2.1) *recursively*, issuing requests to delete department d2 together with its employees in S_2. Finally, the effect of these events is realized in S_3, which is the the new final state since $S_i = S_3$ for all $i > 3$. Figure 1 shows the initial and final database state and intermediate values of relations del:emp and del:dept.[2]

emp

EName	*Salary*	*Dept*
ed1	7000	d1
md1	10000	d1
md2	1000	d1
ed2	6000	d2
md2	8000	d2
md3	9000	d3

dept

DName	*Mgr*
d1	md1
d2	md2
d3	md3

$\leadsto$

emp

EName	*Salary*	*Dept*
md3	9000	d3

dept

DName	*Mgr*
d3	md3

$[S_0]$ ... $[S_3]$

	del:emp	**del:dept**
$[S_0]$	(md1,10000,d1)	$\emptyset$
$[S_1]$	(ed1,7000,d1),(md2,1000,d1)	(d1,md1)
$[S_2]$	(ed2,6000,d2),(md2,8000,d2)	(d2,md2)
$[S_3]$	$\emptyset$	$\emptyset$

Figure 1: Initial state $[S_0]$, final state $[S_3]$ and Δ-relations for example 2.1

Remarks We stress the point, that there is no built-in semantics to literals like del:R or ins:R which are just a notational convention. Instead, the semantics solely depends on the declarative reading of rules (cf. section 3.2). E.g. it is only by virtue of rule (2.2) that del:emp defines a relation containing the (to be) deleted tuples of emp. Moreover, tuples which are present in $[S]$ are not automatically transferred to $[S+1]$ but a frame rule (or another rule which defines an insertion) has to be used. The general form of a frame rule is $[S+1]R(\bar{X})\leftarrow [S](R(\bar{X}), Body)$ where $\bar{X}$ is a vector of variables and *Body* is a conjunction of Datalog literals restricting the propagation of tuples.

Frame rules are generally used for all extensional or materialized relations and can be implemented very efficiently (usually, they are "zero-cost", see section 4).

[2]These may be regarded as "logical" versions of concepts like transition tables in Starburst [5] or delta relations in Heraclitus[Alg,C] [18].

Conflicting Rules and Integrity Constraints Similar to declarative query languages like Datalog, triggered rules are executed simultaneously in Statelog. This leads to a non-branching and therefore confluent and deterministic semantics of Statelog programs. Nevertheless, "semantic conflicts" of rules may arise:

Example 2.2 Consider a relation empl(Employee,BirthDay,Salary) again with the obvious meaning. We want to implement the following, informally given procedure: *Every employee shall be given a salary raise by 5% at his/her birthday; on christmas every employee shall get an extra \$1000.* This is accomplished in Statelog as follows.

$$[S+1]\ \mathsf{mod{:}empl(E,Bday,Sal{\leadsto}Sal1)} \leftarrow \\ [S]\ (\bigtriangledown\mathsf{daily(Today),\ Today{=}Bday,\ empl(E,Bday,Sal),\ Sal1{:=}\ Sal{*}1.05}). \tag{2.3}$$

$$[S+1]\ \mathsf{mod{:}empl(E,Bday,Sal{\leadsto}Sal1)} \leftarrow \\ [S]\ (\bigtriangledown\mathsf{daily(Today),\ xmas(Today),\ empl(E,Bday,Sal),\ Sal1{:=}\ Sal{+}1000}). \tag{2.4}$$

The atom mod:empl(E,Bday,Sal⇝Sal1) in the head is equivalent to the conjunction of del:empl(E,Bday,Sal), ins:empl(E,Bday,Sal1), i.e. to two rules, one defining deletions of old and the other insertions of new tuples. Here, $\bigtriangledown$daily(Today) denotes a *base event* which, in this case, signals the occurrence of an external temporal event. Base events are prefixed with $\bigtriangledown$ ("occurs") and correspond to base relations (EDB) in the logical framework (especially, they must not be defined by rules). Conceptually, base events act as an interface to external applications and are responsible for the initial triggering of transition rules.[3]

Rules (2.3), (2.4) (together with the obvious rules defining the semantics of ins:empl and del:empl) work fine with the intended meaning 365 days a year, unless there is some employee whose birthday is on christmas: e.g. assume that empl(brian, dec/24, 8000) is true in the current state S_0 of the database and the event $[S_0]$ $\bigtriangledown$daily(dec/24) occurs. According to the informal specification given above, it is not defined whether brian has his salary first raised by 5%, then increased by \$1000 or vice or versa, or whether only one salary raise should be performed. Since in Statelog rules are fired simultaneously and the semantics is deterministic, no preference is given to either variant. Instead there are *two* employees named brian in the final state, i.e. "the one" who got 5% and "the other" who got \$1000. Note, that this is not a problem of Statelog: the original informal definition simply ignores this case.

Usually, such overlapping of rules is no problem as it corresponds to the standard union of rule effects. However, in the presence of integrity constraints like functional dependencies such behaviour is intolerable. There are several solutions to this: one is to attach a priority to rules, e.g. by replacing the state term in the head of rule (2.3) by $[S+2]$. Thus, by rule (2.4) \$1000 are added in $[S+1]$ and then by rule (2.3) this new salary is raised further by 5% in $[S+2]$ which is the new final state. Another possibility is to include the negation of one body as an additional constraint in the other body (therefore only one raise is given).[4]

[3] We do not deal with the definition of composite events here, although several of these like conjunction, disjunction, negation and sequence can be defined by means of Statelog rules.

[4] Yet another solution is to choose non-deterministically one rule order or one single rule. This can be accomplished using a nondeterministic *choice* construct [34], however we do not elaborate on this in the present paper.

A more general solution is to use *integrity constraints* expressing the desired functional dependencies. It is well-known that any first-order definable integrity constraint can be coded into a deductive rule in the form of a *denial*. Here, we may use

$$[S]\ (\mathsf{false(E)} \leftarrow \mathsf{empl(E,B1,Sal1)},\ \mathsf{empl(E,B2,Sal2)},\ (\mathsf{B1} \neq \mathsf{B2} \vee \mathsf{Sal1} \neq \mathsf{Sal2})). \qquad (2.5)$$

to express the desired functional dependency. Whenever false(E) can be derived, a violation of the functional dependency for empl is detected (E identifies the critical tuples). In example 2.2, $[S_1]$ false(brian) is true which may be used by other rules to define repairing actions, e.g. to print or delete all integrity violating tuples, or to issue an "abort" operation (see section 4).

Apart from the presented examples, many other applications of active rules are expressible in Statelog including dynamic integrity constraints (by referring to different states), the *delete-rederive algorithm* DRed [19] for incremental maintenance of recursive views or view update translators[5]. This has been practically confirmed using our prototypical implementation of Statelog [20].

3 The Logical Framework

In this section, we describe the logical framework for Statelog more formally. The important notion of Σ-*stratification* is defined which implies the existence of a unique perfect model. Finally, we show that termination of arbitrary Statelog programs is undecidable (a similar result for the "one-state" update language $Datalog^{\neg *}$ has been given in [2]).

3.1 Syntax

Definition 3.1 (Statelog) A Statelog *rule* r is of the form

$$r:\ [S+k_0]H \leftarrow [S+k_1]B_1, \ldots, [S+k_n]B_n$$

where $n \geq 1$, the head H is a Datalog atom, B_i are Datalog literals (i.e. either atoms A or negated atoms $\neg A$), S is the distinguished *state variable* and the offsets $k_i \in \mathcal{N}_0$ are natural numbers. r is called *progressive* if $k_0 \geq k_i$ and *local* if $k_0 = k_i$, for all $i = 1, \ldots, n$. In the latter case, r is called a *query rule*, otherwise *transition rule*.

The *leap* l associated with a literal B_i wrt r is the distance between the state referred to in the head and the state for B_i in the body, i.e. $l = k_0 - k_i$.

A Statelog *program* P is a finite set of range restricted[6], *progressive* rules.

We partition the set of relation symbols of P into the following disjoint sets:

The set $\mathcal{R}_\nabla$ of base events (identified by the first character "∇") which may not be defined by rules, the set $\mathcal{R}_{EDB}$ of extensionally defined relations not occurring in the head of any query rule, the set $\mathcal{R}_{IDB}$ of intensionally defined relations not occurring in the head of any transition rule and, finally, all other relations (sometimes called Δ-*relations*) $\mathcal{R}_\Delta$ (by convention written as ins:R, del:R etc.).

[5]Generally, there are several incomparable alternatives to map changes in a view to changes on base relations. A view update translator specifies the desired alternatives.

[6]Every data variable occurs positively in the body. Hence, rules are bottom-up evaluable.

A Statelog *database instance* D is a finite set of ground atoms $[s_0]A$ over $\mathcal{R}_{EDB}$ where s_0 is some fixed, distinguished state (usually the "last final state"). $\square$

Using auxiliary relations, it is possible to rewrite Statelog rules which involve state terms $[S+k]$ for offsets $k \geq 2$ into equivalent rules where only offsets $k' \in \{0,1\}$ occur. Thus, in the sequel we may assume that every state term is either $[S]$ or $[S+1]$.

The requirement of progressiveness is a major difference to the related languages Datalog$_{1S}$ and Templog. In the context of defining the behavior of active rules and updates, it is not allowed to make assumptions about future states when defining the current state, or to change past states (where transactions have already committed). As a consequence, in our context the database evolution may be computed by "sliding" a window of exactly two states (S_{prev} and S_{curr}) over the (infinite) database history. All states before the previous state S_{prev} may be discarded since their effects are materialized in S_{prev}.

3.2 Semantics

The semantics of a Statelog program P is defined by a mapping to an ordinary logic program $P^\star$: every Statelog atom $[S+k]P(t_1,\ldots,t_n)$ of arity n in P maps to an $(n+1)$-ary atom $P(S+k,t_1,\ldots,t_n)$ in $P^\star$. Formally, an expression $S+k$ is regarded as a k-fold application of the unary function symbol "+1" to S. Thus, the set of *state identifiers* $\Sigma = \{0, (0)+1, ((0)+1)+1, \ldots\}$ is identified with the natural numbers $\mathcal{N}_0$.

The meaning of P is now simply the meaning of $P^\star$ under a suitable logic programming semantics. Here, we will focus on the decidable class of Σ-*stratified* programs P which imply that $P^\star$ has a unique *perfect model* [24]. The idea of Σ-stratification is that *within* a certain state S, no negative cyclic dependencies may exist. On the other hand, rules involving negative dependencies *across* states guarantee local stratification and thus the existence of a unique perfect model. E.g. rules (2.1) and (2.2) would not be stratified due to the negative cyclic dependency $\mathsf{del{:}emp} \overset{(2.1)}{\longleftarrow} \mathsf{emp} \overset{\neg(2.2)}{\longleftarrow} \mathsf{del{:}emp}$. However, since these literals refer to different states the rules correspond to a locally stratified program.

Definition 3.2 (Σ-Stratification) The *labeled dependency graph* $\mathcal{D}_P$ of a Statelog program P is defined as follows. For every rule r and every predicate symbol R occurring in P there is a vertex in $\mathcal{D}_P$. Let r be a rule

$$r: \quad \tau_0 H \quad \leftarrow \tau_1 B_1, \ldots, \tau_n B_n$$

of P where τ_i are state terms and H and B_i are Datalog atoms or literals, respectively. Then $\mathcal{D}_P$ contains

(i) an edge $r \overset{0}{\to} H$

(ii) and for every $i = 1, \ldots, n$:

  (a) a negative edge $A_i \overset{l,\neg}{\to} r$, if B_i is a negative literal $\neg A_i$

  (b) a positive edge $B_i \overset{l}{\to} r$, otherwise.

Here, l is the leap associated with B_i wrt r. A cycle C of $\mathcal{D}_P$ involving only edges with leap 0 is called a *local cycle*, otherwise C is *progressive*.

A program P is called Σ-*stratified* if no local cycle of $\mathcal{D}_P$ contains a negative edge. □

Example 3.3
Let P be the rules r_1: $[S+1]$ del:emp(E,Sal,D) $\leftarrow[S]$ (del:dept(D,_), emp(E,Sal,D)) and r_2: $[S+1]$ emp(E,Sal,D) $\leftarrow[S]$ (emp(E,Sal,D), $\neg$ del:emp(E,Sal,D)). The dependency graph of P is shown in figure 2. Note, that although there is a negative cycle in $\mathcal{D}_P$ this cycle is not local, so P is Σ-stratified.

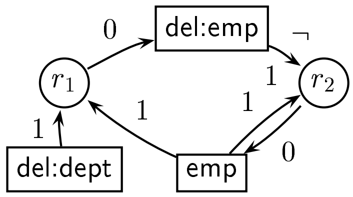

Figure 2: A Labeled Dependency Graph

It is easy to see, that the logic program $P^\star$ associated with a Statelog program P is locally stratified if P is Σ-stratified, hence $P \cup D$ has a unique perfect model $\mathcal{M}$. Since there are infinitely many state identifiers $s \in \Sigma$, the perfect model $\mathcal{M}$ consists of infinitely many "snapshots" $\mathcal{M}[s]$. However, $\mathcal{M}$ can be finitely represented due to its periodic structure: We call $\mathcal{M}$ *ultimately periodic*, if for some $s_0 \in \mathcal{N}_0$ there is a *period* $T \in \mathcal{N}_0$ such that $\mathcal{M}[s+T] = \mathcal{M}[s]$ for all $s \geq s_0$. Using this notion, it can be shown [22]:

Theorem 3.4 *Every Σ-stratified Statelog program P with database D has a unique perfect model $\mathcal{M}$ which is ultimately periodic for some period T.*

In the context of update transactions, a unique final state is reached iff $T = 1$:

Definition 3.5 A Σ-stratified program P *terminates* for a given database D if the perfect model $\mathcal{M}$ of $P \cup D$ is ultimately periodic with $T = 1$. □

In the sequel, we consider only Σ-stratified programs P and call $\mathcal{M}$ simply *the model* of $P \cup D$.

3.3 Termination, Expressiveness and Complexity

Due to theorem 3.4 the model $\mathcal{M}$ of a Statelog program P with database D is ultimately periodic. Therefore, it is possible to check for termination at *runtime* (i.e. for a given database instance) by watching for a repetition of — not necessarily adjacent — states which indicates the occurrence of the period. However, since the length of the period may be exponential in the size of the database (cf. [22, 11]) this check may be prohibitively expensive.

For this reason, it is desirable to consider programs which guarantee termination *independent* of the initial database D. Thus, a single static program analysis at compile time would be sufficient to guarantee termination. However, there is no effective procedure which identifies all terminating Statelog programs:

Theorem 3.6 *Given a Statelog program P, it is undecidable whether P terminates for* all *databases D.*

Proof Let φ be a first-order sentence without function symbols. We use the fact that validity of first-order sentences on finite structures is undecidable (Trakhtenbrot's theorem, see e.g. [15]). φ can be coded into a stratified Datalog program P_φ, and therefore into a set of Σ-stratified rules, such that φ is true in the stratified model of P_φ for all database instances D iff φ is valid on finite structures. Consider the following Σ-stratified Statelog program P:

$$[S]\ \varphi \leftarrow [S]\ P_\varphi. \qquad\qquad [S+1]\ \mathsf{q} \leftarrow [S]\ (\neg\ \mathsf{q},\ \neg\varphi).$$

If φ is valid then P has period $T = 1$ for all databases D since the second rule does not fire. Otherwise, $\neg\varphi$ is true for some D and $P \cup D$ has period $T = 2$ since q oscillates between true and false. ∎

Apart from checking termination at runtime, another solution is to consider restricted classes of programs for which termination may be guaranteed in advance.[7] Not surprisingly, such restrictions have an impact on the expressiveness of the language, as will be seen in the following. First, we consider *guarded programs* introduced in [22]:

Definition 3.7 (Guarded Rules) A set of rules P is called *guarded* if every progressive cycle C of $\mathcal{D}_P$ contains a positive occurrence of a base event ∇E. □

An example of a guarded program are the rules of example 2.2. In [22] it was shown that frame rules need not be considered when constructing $\mathcal{D}_P$. (Intuitively, this is due to the fact that a frame rule for R can never produce new information for R.) Base events act as guards in the body of progressive rules and guarantee that rules fire only once at the beginning of a transaction. The number of intermediate states generated by guarded rules is equal to the maximal sum of leaps of cycles in $\mathcal{D}_P$ and therefore fixed, independent of the database. In other words, there is no "recursion over states". This in turn implies that the expressiveness of guarded rules (in the sense of [9]) is equal to stratified Datalog. The class of Δ-*monotone Statelog* programs is even more powerful and introduced next.

3.3.1 Δ-Monotone Statelog

The basic idea of Δ-monotone Statelog is to guarantee that each relation $R \in \mathcal{R}_\Delta$ is either nondecreasing or nonincreasing, i.e. either $[S]R \subseteq [S+1]R$ or $[S+1]R \subseteq [S]R$. This implies that evaluation of rules always terminates in time polynomial in the size of the database D. However, this notion of monotony is not effective, since it is undecidable, given arbitrary P whether a relation R of P is monotone. Therefore some decidable approximation has to be used:

[7] Such an approach is taken e.g. in [5] in the context of the Starburst rule system where the triggering graph of a set of rules is required to be acyclic.

Definition 3.8 (Δ-monotone Statelog) For a set of rules P we define the following properties

(i) *Positive Recursion*: Every non-guarded, progressive cycle C in $\mathcal{D}_P$ contains an *even* number of negations.

(ii) *1-Cycles*: For every non-guarded, progressive cycle C holds: $\sum_{e \in C} l_e \leq 1$, where l_e is the leap of an edge e in C.

(iii) *Isolation*: Every relation R in a non-guarded, progressive cycle C depends only on Δ-relations from the same strongly connected component or from relations in $\mathcal{R}_{EDB}$ and $\mathcal{R}_{IDB}$.

P is called Δ-*monotone* if it satisfies (i)–(iii). □

(i) ensures that all relations are positive wrt recursive triggering, while (ii) states that each cycle corresponds to exactly one state transition, i.e. there are no "gaps". Finally, (iii) implies that Δ-relations depend on constant relations or relations from the same "iteration block" only.

(i)–(iii) are minimal in the sense that abandoning any of these may lead to non-terminating programs as is illustrated by the following examples:

Example 3.9 Let $\mathsf{q,r,u,o}$ be Δ-relations. Consider the following programs, each violating a single property from above.

$$
\begin{array}{lrcl}
P_{(i)}: & [S+1]\ \mathsf{q} & \leftarrow & [S]\ \neg\ \mathsf{q} \\
P_{(ii)}: & [S+1]\ \mathsf{q} & \leftarrow & [S]\ \neg\ \mathsf{r} \\
 & [S+1]\ \mathsf{r} & \leftarrow & [S]\ \neg\ \mathsf{q}
\end{array}
\qquad
\begin{array}{lrcl}
P_{(iii)}: & [S]\ \mathsf{o(X)} & \leftarrow & [S]\ (\mathsf{m(X,Y)},\ \neg\ \mathsf{u(Y)}) \\
 & [S+1]\ \mathsf{u(X)} & \leftarrow & [S]\ (\mathsf{m(X,Y)},\ \neg\ \mathsf{o(Y)}) \\
 & [S+1]\ \mathsf{m(X,Y)} & \leftarrow & [S]\ \mathsf{m(X,Y)} \\
 & [S]\ \mathsf{u(X)} & \leftarrow & [S]\ \mathsf{n(X)}
\end{array}
$$

Given an empty database D as input, both $P_{(i)}$ and $P_{(ii)}$ do not terminate: $P_{(i)}$ violates *positive recursion* which leads to the non-terminating sequence of $\mathcal{M}$:

[0] $\neg$ q, [1] q,[2] $\neg$ q, [3] q, ...

Similarly, $P_{(ii)}$ violates *1-Cycles* and the sequence oscillates between {q,r} and {$\neg$ q, $\neg$r}. Note that $P_{(ii)}$ contains "gaps" between states, in the sense that $[S+2]q \Leftrightarrow [S]q$ is valid. This possibility of gaps is ruled out by (ii).

The requirement of *isolation*, together with (i) and (ii), guarantees that every relation in a cycle C depends either only positively or only negatively on any other relation (with the exception of relations from $\mathcal{R}_{EDB}$ and $\mathcal{R}_{IDB}$ which are guaranteed to be constant between states.)

In $P_{(iii)}$ a frame rule is given for m but not for n. u and o belong to the same SCC and depend on the EDB relation m which is propagated by a frame rule. However, since u depends also on n which is neither from the same SCC nor a constant relation, *isolation* is violated: Let the initial database $D = \{\mathsf{n(a)}, \mathsf{m(a,b)}, \mathsf{m(b,c)}, \mathsf{m(c,a)}\}$ at [0] be given. n(a) changes from true in [0] to false in [1]. It is easy to see that $P_{(iii)}$ does not terminate since $\mathcal{M}$ has period $T = 3$. E.g. for u we get

[0] u(a), [1] u(b), [2] u(c), [3] u(a), ...

Theorem 3.10 *A Δ-monotone program P terminates for all databases D.*

Proof *(sketch)* By coding every strongly connected component P of Δ-relations into a first-order formula $\varphi_P(\bar{X}, R)$ which is positive in R[8] such that each state transition in the computation of the perfect model of P corresponds to one application of the fixpoint operator on $\varphi_P(\bar{X}, R)$. R involves all Δ-relations of P. It can be shown that φ_P is positive in R based on the assumption that P is Δ-monotone. From this the existence of the least fixpoint and thus termination follows. ■

3.3.2 Expressiveness & Complexity

We now present results characterizing the expressiveness and complexity of Statelog. The notion of expressiveness is based on the usual framework of Chandra and Harel[9]. Expressiveness is usually defined for classes of queries, where a query is a mapping from instances of an input scheme (EDB) to instances of an output scheme (IDB). Updates are regarded as mappings where input and output scheme are identical. This allows to investigate the expressive power of arbitrary database transformations, i.e. queries and updates.

Since Statelog distinguishes between local query rules and state-changing transition rules, two notions of expressiveness arise. The first, Statelog *query expressiveness* is given by only considering local rules. Since these are essentially equivalent to stratified Datalog rules, Statelog query expressiveness is equivalent to stratified Datalog.

The second, Statelog *transaction expressiveness* is given by database transformations using arbitrary transition rules. Since the latter characterizes the expressive power of the framework as a whole we refer to it as *the* Statelog expressiveness.

In [20] it is proven that Statelog rules may be encoded into equivalent While programs and vice versa. Similarly (cf. theorem 3.10), Δ-monotone rules can be coded into Fixpoint logic, i.e. the least fixpoint operator applied to a positive first-order formula (see e.g. [3] for the definition of While and Fixpoint). Therefore, we arrive at the following expressiveness results:

Theorem 3.11

- *Guarded Statelog is equivalent to* Stratified Datalog
- *Δ-monotone Statelog is equivalent to* Fixpoint
- *Statelog is equivalent to* While

As a consequence of this encoding, the complexity of evaluation of Statelog programs can be determined: Δ-monotone Statelog transactions can be executed in PTIME, while arbitrary Statelog transactions may require PSPACE. Assuming an ordered domain, all PTIME (PSPACE) queries can be expressed in Δ-monotone Statelog (general Statelog).

[8] R appears under an even number of negations (including 0). $\bar{X}$ are the free variables of φ_P and P, respectively.

Computing Well-Founded Views with Statelog The expressive power of Statelog stems from the ability to refer to different database states. This allows to give a surprisingly simple and elegant coding of well-founded models [30] in Statelog. The alternating fixpoint characterization of the well-founded model [29] is coded into Statelog as follows:

Attach state terms to a given non-stratified Datalog program P, such that all positive atoms refer to $[S+1]$ and all negated atoms refer to $[S]$. The resulting Statelog program P_{afp} computes the alternating fixpoint of P.

Example 3.12 Consider the well-known non-stratified program

$$P: \; win(X) \leftarrow move(X,Y), \neg win(Y).$$

Then P_{afp} is

$$[S+1] \; \mathsf{win(X)} \leftarrow [S+1] \; \mathsf{move(X,Y)}, \; \neg \; [S] \; \mathsf{win(Y)}.$$

As usually, we assume that the input relation move is kept constant by a frame rule. Given the input move $= \{(a,b),(b,a),(b,c),(c,d)\}$ the following states are computed:

[0] win $= \emptyset$
[1] win $= \{$a,b,c$\}$
[2] win $= \{$c$\}$
[3] win $= \{$a,b,c$\}$
[4] win $= \{$c$\}$
...

In accordance with [29], even numbered states correspond to a nondecreasing sequence of underestimates while odd numbered states represent a nonincreasing sequence of overestimates of true atoms. While this program terminates iff the well-founded model is total, a Δ-monotone encoding P'_{afp} can also be given. Indeed, discarding the last (Δ-monotony destroying) rule in $P_{(iii)}$ of example 3.9 yields a Δ-monotone program $P'_{(iii)}$ for the *win*-program: $[S]$ u(X) is an nondecreasing sequence of underestimates, while $[S]$ o(X) corresponds to a nonincreasing sequence of overestimates of the true atoms. Clearly, this allows to compute true and undefined atoms of the well-founded model by a Δ-monotone and thus terminating program.

In the following section, we present a common architecture for active and deductive databases within the logical framework of Statelog. The combination of increased expressiveness (due to state terms) and declarative semantics allows a quite intuitive control and understanding of rule behavior. The proposed architecture is based on the class of Δ-monotone programs.

4 A Statelog Architecture for Active Databases

On the one hand, transactions have an inherently non-monotonic behavior due to deletions and modifications; on the other hand, in order to ensure termination some form of monotonic behavior has to be presupposed. We propose to resolve this conflict by a unified architecture for queries and transactions based on the class of Δ-monotone Statelog.

During the intermediate iteration, every Δ-relation is monotonic, i.e. either nondecreasing or nonincreasing. The overall effect of a transaction is again non-monotonic. The basic execution model for active rules is as follows (cf. figure 3):

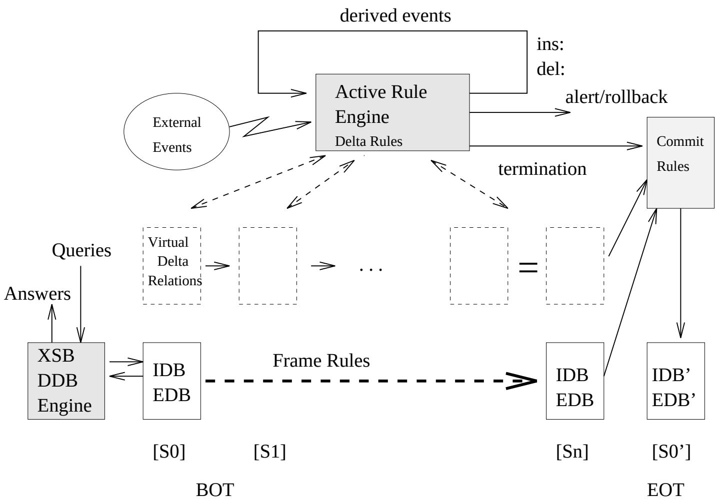

Figure 3: Architecture and Execution Model of an Active Statelog Database System

Given the current state S_0 of a database, local query rules may be used to query the contents of the database without changing the state.

Triggered by the occurrence of one or more base events, Δ-*rules* (i.e. transition rules having a Δ-relation in the head) become activated which modify the contents of the virtual Δ-*relations* $\mathcal{R}_\Delta$ accordingly. This results in the *transition* $\mathcal{T}_1 : S_0 \to S_1$ of the database to a new virtual state S_1.

Induced by the effect of a transition, more Δ-rules may be triggered etc. until there are no changes to the Δ-relations. By certain Statelog rules (cf. (4.3)) it is possible to check for changes in Δ-relations. After termination, *commit rules* (like e.g. (4.2)) materialize the effect which has been accumulated in Δ-relations provided no integrity constraint like (2.5) is violated. Otherwise, special commit rules define the new final state to be the initial state S_0 thus implementing an "abort" operation.

While an update iteration is running, special frame rules propagate base relations to subsequent states without change. Note that these frame rules are "zero cost" and do not involve any computations, in contrast to the frame rule (2.2) where deletions had to be considered: while the architecture of figure 3 implements a *deferred semantics* for updates, the (simplified) examples from section 2 (e.g. frame rule (2.2)) correspond to an *immediate semantics* because changes are materialized in base relations as soon as a delete (or insert) occurs.

In order to illustrate how different transactional behavior can be programmed, we present a variant of example 2.1 using deferred semantics.

Example 4.1 Instead of frame rule (2.2) we use an *unconditioned frame rule*

$$[S+1]\ \mathsf{emp(E,Sal,D)} \leftarrow [S]\ \mathsf{emp(E,Sal,D)},\ \mathit{running}. \tag{4.1}$$

and a *commit rule*

$$
\begin{aligned}
&[S+1]\ \mathsf{emp(E,Sal,D)} \leftarrow \\
&\quad [S]\ ((\mathsf{emp(E,Sal,D)} \vee \mathsf{ins{:}emp(E,Sal,D)}), \\
&\qquad \neg\ \mathsf{del{:}emp(E,S,D)}, \\
&\qquad \mathit{terminated}).
\end{aligned} \tag{4.2}
$$

For dept analogous rules are used. Finally, the transition rule (2.1) is used without change.

The events *running* and *terminated* can be implemented as built-ins in a real system, however, they have a precise logic semantics using rules of the form[9]

$$
\begin{aligned}
&[S+1]\ \mathit{running} \leftarrow [S+1]\ R(\bar{X}),\ [S]\ \neg R(\bar{X}) \\
&[S+1]\ \mathit{running} \leftarrow [S+1]\ \neg R(\bar{X}),\ [S]\ R(\bar{X}) \\
&[S]\ \mathit{terminated} \leftarrow [S]\ \neg\ \mathit{running}
\end{aligned} \tag{4.3}
$$

for all Δ-*relations* R. This ensures that *running* is true iff there are changes in Δ-relations which in turn guarantees that all relations with an unconditioned frame rule like (4.1) are constant during the *running* phase of a transaction.

5 Related Work

The idea of using state terms in Datalog has been proposed in [33, 34] to allow a unified semantics for active and deductive rules, and independently in [21, 22] as a means to specify update procedures in a declarative way.

X and Y-rules of [33] are very similar to local and progressive rules, respectively. However, while local and progressive rules may involve (stratified) recursion within a single state, XY-rules are essentially non-recursive in a given state. Nevertheless, since the effect of local recursions can be simulated by corresponding state transitions and frame rules[10], it can be shown that already "pure" XY-stratified programs (i.e. without extensions like nondeterministic choice) can express While. As a consequence, it is undecidable whether an arbitrary XY-stratified program terminates on all databases or might become periodic with a period $T > 1$. Moreover, there are terminating XY-stratified programs whose runtime is exponential in the size of the database (e.g. a variant of a program given in [11] which simulates a binary counter). Therefore, our results give new insights into the properties of XY-stratified programs. In particular, a notion corresponding to Δ-monotone Statelog can be used to define a large class of terminating and tractable XY-stratified programs.

From a logic point of view, Statelog can be viewed as a variant of the temporal deductive database languages *Datalog*$_{1S}$ and *Templog* [6]. On the one hand, Statelog is restricted to progressive rules (a natural prerequisite in the context of updates and active databases which allows to discard previous states) while Datalog$_{1S}$ allows to reason forward and backward in time. On the other hand, our notion of Σ-stratification is more general than the one given in [10] since we distinguish between local and progressive negative dependencies.

[9] $\bar{X}$ denotes a vector $X_1, \ldots, X_n$ of variables.

[10] E.g. compute the transitive closure by adding new edges only at state transitions instead of using a single state.

[1, 3] define a whole class of languages for queries and updates in Datalog. Similar to Statelog, the deterministic update language $Datalog^{\neg *}$ allows to define deletions in the head of rules. While for this language similar expressiveness results hold, the proposed semantics rely on a certain bottom-up evaluation strategy and can only be understood procedurally. Moreover, when restricted to programs without negation in the head, the proposed semantics do not coincide with the stratified or well-founded semantics.

6 Conclusion & Outlook

We have presented a unified logical framework for active state-changing and passive query rules based on Statelog, which is very simple, yet flexible and powerful enough to express typical applications of ECA rules.

Generalizing previous results of [22], the class of Δ-monotone programs has been presented which guarantee termination for all inputs — a property that has been proven to be undecidable in general. Δ-monotone Statelog expresses the Fixpoint transformations on databases while general Statelog is equivalent to While. The given results carry over to XY-stratified Datalog and establish an interesting class of terminating and tractable (polynomial-time computable) active rules.

In a sense, Statelog is a "minimal" extension of Datalog allowing just as much expressiveness as is necessary to express the essentials of active rules, i.e. monitoring and definition of changes in the database. Further extensions to the "pure" language kernel are possible and necessary for real world applications. Although such extensions usually can be integrated into an implementation with relative ease, their theoretical implications remain an interesting field of future research. One possibility e.g. is to investigate the interplay between nondeterministic extensions (like invented values or choice, see e.g. [34]) and states. These extensions are a powerful programming tool and have a similar impact on expressiveness like assuming an ordered database.

A prototypical implementation of Statelog using XSB-Prolog as a deductive database engine [26] has been completed recently and shows that programming in Statelog is feasible and intuitive [20]. The related approach of XY-stratified Datalog has been incorporated into LDL++ [34]. Due to simplicity of the language, integration of the Statelog approach into other deductive database systems like CORAL [25] or Glue-Nail [14] should be straightforward.

References

[1] S. Abiteboul. Updates, a new frontier. In *ICDT'88 (Second Intl. Conf. on Data Base Theory), Bruges*, number 326 in LNCS, pages 1–18. Springer, 1988.

[2] S. Abiteboul and E. Simon. Fundamental properties of deterministic and nondeterministic extensions of datalog. *Theoretical Computer Science*, 78, 1991.

[3] S. Abiteboul and V. Vianu. Datalog extensions for database queries and updates. *Journal of Computer and System Sciences*, 43, 1991.

[4] R. Agrawal and N. H. Gehani. Ode (object database and environment): The language and the data model. In *Proc. ACM SIGMOD Intl. Conference on Management of Data*, pages 36–45, May 1989.

[5] A. Aiken, J. Widom, and J. M. Hellerstein. Behavior of database production rules: Termination, confluence, and observable determinism. In M. Stonebraker, editor, *Proc. ACM SIGMOD Intl. Conference on Management of Data*, San Diego, California, 1992.

[6] M. Baudinet, J. Chomicki, and P. Wolper. Temporal deductive databases. In A. Tansel, J. Clifford, S. Gadia, S. Jajodia, A. Segev, and R. Snodgrass, editors, *Temporal Databases*. Benjamin/Cummings, 1993.

[7] A. P. Buchmann, J. Blakeley, J. A. Zimmermann, and D. L. Wells. "building an integrated active oodbms" requirements, architecture, and design decisions. In *11th Intl. Conference on Data Engineering*, Taiwan, 1995.

[8] S. Chakravarthy, V. Krishnaprasad, E. Anwar, and S.-K. Kim. Composite Events for Active Databases: Semantics, Contexts and Detection. In J. B. Bocca, M. Jarke, and C. Zaniolo, editors, *Proc. Intl. Conference on Very Large Data Bases*, pages 606–617, Santiago de Chile, 1994.

[9] A. Chandra and D. Harel. Structure and complexity of relational queries. *JCSS*, 25(1):99–128, 1982.

[10] J. Chomicki. *Functional Deductive Databases: Query Processing in the Presence of Limited Function Symbols*. PhD thesis, Rutgers University, 1990.

[11] J. Chomicki and T. Imieliński. Finite representation of infinite query answers. *ACM Transactions on Database Systems*, June 1993.

[12] U. Dayal. Active database management systems. In C. Beeri, J. W. Schmidt, and U. Dayal, editors, *Proc. of the 3rd Int. Conf. on Data and Knowledge Bases: Improving Usability and Responsiveness*, pages 150–169, San Matheo, CA, June 1988. Information Processing Association of Israel, Morgan Kaufmann.

[13] U. Dayal, B. Blaustein, A. Buchmann, U. Chakravarthy, M. Hsu, R. Ledin, D. McCarthy, A. Rosenthal, S. Sarin, M. J. Carey, M. Livny, and R. Jauhari. The HiPAC project: Combining active databases and timing constraints. In *sigmod*, pages 51–70, Mar. 1988.

[14] M. Derr, S. Morishita, and G. Phipps. The glue-nail deductive database system: Design, implementation, and evaluation. *The VLDB Journal*, 3(2):123–160, Apr. 1994.

[15] H.-D. Ebbinghaus, J. Flum, and W. Thomas. *Mathematical Logic*. Springer, 1984.

[16] S. Gatziu and K. R. Dittrich. Events in an active object-oriented database system. In Paton and Williams [23].

[17] N. H. Gehani, H. V. Jagadish, and O. Shmueli. Event specification in an active object-oriented database. In *Proc. ACM SIGMOD Intl. Conference on Management of Data*, 1992.

[18] S. Ghandeharizadeh, R. Hull, D. Jacobs, J. Castillo, M. Escobar-Molano, S.-H. Lu, J. Luo, C. Tsang, and G. Zhou. On implementing a language for specifying active database execution models. In *Proc. Intl. Conference on Very Large Data Bases*, 1993.

[19] A. Gupta, I. S. Mumick, and V. S. Subrahmanian. Maintaining views incrementally. In *Proc. ACM SIGMOD Conf.*, page 157, Washington, DC, May 1993.

[20] U. Hamann. A System for Specifying and Executing Updates in a State-Oriented Extension of Datalog (in german). Master's thesis, Institut für Informatik, Universität Freiburg, 1995.

[21] M. Kramer, G. Lausen, and G. Saake. Updates in a rule-based language for objects. In *Proc. Intl. Conference on Very Large Data Bases*, Vancouver, 1992.

[22] G. Lausen and B. Ludäscher. Updates by reasoning about states. In *2nd Intl. East-West Database Workshop*, Workshops in Computing, Klagenfurt, Austria, 1994. Springer.

[23] N. W. Paton and M. H. Williams, editors. *Proc. of the 1st Intl. Workshop on Rules in Database Systems (RIDS)*, Workshops in Computing, Edinburgh, Scotland, 1993. Springer.

[24] T. C. Przymusinski. On the declarative semantics of deductive databases and logic programs. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 191 – 216. Morgan Kaufmann, 1988.

[25] R. Ramakrishnan, D. Srivastava, and S. Sudarshan. CORAL: Control, relations and logic. In L.-Y. Yuan, editor, *Proc. Intl. Conference on Very Large Data Bases*, Vancouver, 1992.

[26] K. Sagonas, T. Swift, and D. S. Warren. XSB as an efficient deductive database engine. In *Proc. ACM SIGMOD Intl. Conference on Management of Data*, 1994.

[27] M. Stonebraker and G. Kemnitz. The postgres next generation database management system. *Communications of ACM*, 34(10):78–93, Oct. 1991.

[28] M. Stonebraker, L. Rowe, and M. Hirohama. The implementation of POSTGRES. *IEEE Transactions on Knowledge and Data Engineering*, 1990.

[29] A. Van Gelder. The alternating fixpoint of logic programs with negation. In *Proc. ACM Symposium on Principles of Database Systems*, pages 1–10, 1989.

[30] A. Van Gelder, K. Ross, and J. Schlipf. The well-founded semantics for general logic programs. *JACM*, 38(3):620 – 650, July 1991.

[31] J. Widom. Active databases. In M. Stonebraker, editor, *Readings in Database Systems, 2nd edition.* Morgan Kaufmann, 1994. Introduction to chapter 4.

[32] J. Widom and S. J. Finkelstein. Set-oriented production rules in relational database systems. In *Proc. of the ACM SIGMOD Symp. on the Management of Data*, pages 259–264, 1992.

[33] C. Zaniolo. A unified semantics for active and deductive databases. In Paton and Williams [23].

[34] C. Zaniolo, N. Arni, and K. Ong. Negation and Aggregates in Recursive Rules: the LDL++ Approach. In S. Ceri, K. Tanaka, and S. Tsur, editors, *Proc. Intl. Conference on Deductive and Object-Oriented Databases (DOOD)*, number 760 in LNCS. Springer, 1993.