Reconciling Active and Deductive Databases by States

Institut für Informatik — Report 70

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

September 5, 1995

Abstract

We propose a state-oriented extension to Datalog called *Statelog* which comprises two kinds of rules, passive *query rules* and active *transition rules* within a simple, yet flexible logical framework. Statelog programs have deterministic model-theoretic semantics for queries *and* updates thus reconciling active and deductive databases. Using the Statelog framework we prove that termination of active rules is undecidable in general and introduce the class of Δ-*monotone* programs for which termination is guaranteed. This class provides a generalization to previous results for Statelog and is of practical importance as it includes self-triggering recursive rules. Formal results characterizing the expressiveness and complexity are established which carry over to related state-oriented extensions of Datalog.

Computing Reviews Classification: H.2.3 [**Database Management**]: Languages — *data manipulation languages* (DML); H.2.4 [**Database Management**]: Systems — *query processing*

Additional Keywords: active databases, deductive databases, database updates, expressive power

1 Introduction

Deductive and active databases may be regarded as two ends of a spectrum of database rule languages [23] (cf. table 1). On the one end, Datalog-like query languages provide a concise and elegant representation of extensional and derived data. Due to the high level of abstraction, the effect of deductive rules can be more easily grasped by a programmer than the effect of active rules. Datalog extends the expressive power of traditional relational languages by the use of recursion and negation while retaining declarative semantics. However, Datalog does not provide enough expressiveness or control to support dynamic aspects like updates in a natural way. The latter is precisely the strength of active and production rule languages on the other, right end of the spectrum.

Active rules support automatic triggering of data manipulating operations as a response to external or internal events. Moreover, they often allow to exert explicit control on rule execution. Apart from the definition of update operations, production rules have been successfully applied in contexts like checking and enforcing integrity constraints, incremental view maintenance, authorization checking, versioning and condition monitoring. Commercial systems like Ingres, Oracle or Sybase provide triggers and alerters which are specific forms of active rules.

Deductive Rules ..... ≀ ..... *Active Rules*

|—————————————————————————————————|

Datalog .... RDL .. A-RDL .. Ariel .. Starburst .. Postgres

Table 1: Spectrum of database rule languages

A serious problem often referred to in the context of active databases 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*" [24].

In this paper we propose *Statelog* as a unifying framework which bridges the gap between active and deductive databases by incorporating states into Datalog. Statelog comprises two kinds of rules, passive *query rules* and active *transition rules* within a simple and elegant, yet flexible logical framework. Simplicity is achieved by a careful extension of Datalog using state terms at the language level thereby allowing to refer to different database states and changes in relations. Thus, Statelog combines the advantages of active and deductive rules: On the one hand, Statelog programs have a model-theoretic deterministic semantics, independent of a specific evaluation strategy like forward or backward chaining.[1] On the other hand, the use of state terms extends the expressiveness and control capabilities of the language which are necessary to describe active data manipulation operations.

The idea of using state terms in Datalog has been proposed in [26, 27] to allow a unified semantics for active and deductive rules, and independently in [15, 16] as a means to specify update procedures in a declarative way.

The main contributions of this paper are as follows. We demonstrate that the proposed language Statelog can afford both active rules and deductive rules within a unified frame-

[1] Although there is a natural correspondence between rule type and evaluation strategy, see section 4.4.

work. This provides logical semantics for active rules and allows an intuitive understanding of rule behavior. By means of illustrative examples, we show how transaction programming can be accomplished in Statelog. Formal results on expressiveness, complexity and termination of Statelog programs are given which also give new insights into the closely related *XY-stratified Datalog* programs [26]. Especially, using the Statelog framework we prove that termination of active rules is undecidable in general and introduce the class of Δ-*monotone* programs for which termination is guaranteed. This class is of practical importance since it allows self-triggering recursive rules and is more expressive than the class of *guarded programs* presented in [16], yet Δ-monotony can be checked by a static analysis at compile time.

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 including the computation of the perfect model. In section 4, a common architecture for active and deductive databases is presented based on Δ-monoton rules and a possible implementation of Statelog is outlined. After comparing our approach to related work in section 5, we summarize our main results in the last section.

2 Database Programming with States

2.1 Active Rules vs States

Active rules are usually described by the general *Event-Condition-Action* (ECA) scheme:

when *event* **if** *condition* **then** *action*

On occurrence of an *event* like an update to a relation, an *action* (e.g. another update, an alert or a rollback command just to name a few typical possibilities) is executed if the *condition* (an SQL-query or a boolean condition) holds.

A distinctive feature of many active and production 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. For example the database programming language *Heraclitus[Alg,C]* [13], an extension of C, supports multiple virtual states in rule conditions and elevates delta relations to "first-class citizens" of the language. In production rule languages like *Starburst*, references of the form deleted R, inserted R, old updated R and new updated R are used to refer to deleted and insert tuples of a relation R, or to changes in R.

2.2 Basic Ideas of Statelog

In order to obtain a unified logical framework of state-changing active and deductive rules, we propose to extend Datalog with the ability to refer to different database states. The resulting language *Statelog* 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* [26] (cf. section 5). The basic execution model of Statelog is as follows (cf. figure 1):

Given the current state S_0 of a database, non-state changing query rules are used to query the contents of the database. Query rules correspond to the usual Datalog rules for

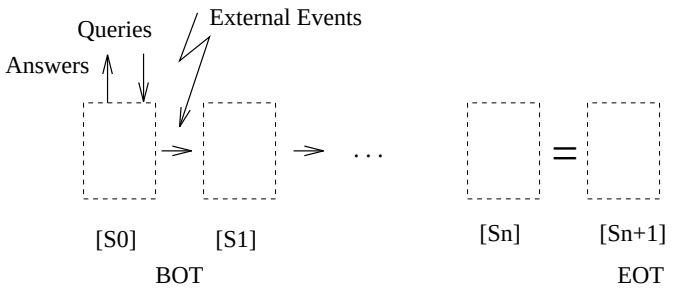

Figure 1: A Basic Statelog Execution Model

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 rule heads (e.g. insert or delete) provided the condition given by the body of the rules is satisfied. This results in a sequence $S_1, S_2, \ldots, S_n, \ldots$ of virtual states. Computation terminates if a fixpoint is reached, i.e. if two subsequent states are equal. We will show in section 3.3.1 that it is undecidable whether a set of transition rules terminates for all database instances. However, provided the rules belong to the class of Δ-*monotone transition rules* which can be recognized at compile time termination is guaranteed (cf. section 4).

2.3 Examples

In order to illustrate the above notions, consider the following simplified database schema (adapted from [25]): The relation emp(EmpName, Salary, DeptName) holds information on employees, i.e. a unique name, the salary and the department for which an employee works; dept(DeptName, MgrName) contains for each department the name of its manager.

Example 2.1 (Enforcing Referential Integrity)
Assume we want to ensure that whenever departments are deleted, all employees in the deleted departments are deleted as well. Using the SQL-style syntax from [25] the following production rule may be used:

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

On occurrence of a delete to dept the corresponding deletion to the emp relation is performed. In Statelog the same effect is accomplished by a *transition rule*:[2]

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

If in the current state S a tuple with department D is deleted from dept then all employees working in D are also deleted. All other tuples are propagated to the next state by a *frame*

[2]For notational convenience, we allow to "factor out" common state terms like e.g. $[S]$ in rules.

rule for emp (a corresponding rule is used for dept):

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

The update terminates, if there is no change in two successive states. In this example, if the delete request for dept is raised in S, the states $S+2$ and $S+3$ are equal, hence the update terminates. By convention, an atom del:R (ins:R) corresponds to an event if it occurs in the body of a rule (*has* R been deleted (inserted)?) and to an action if it occurs in the head (R *is to be* deleted (inserted)). While events require special handling in order to achieve efficient detection of applicable rules in a concrete implementation, they are regarded as ordinary relations in the formal logical framework. Moreover, del:R and ins:R have no 'built-in' semantics but are just a notational convention to express the intended meaning (to make this clear we could write del_R and ins_R instead). In other words, del:R and ins:R can be regarded as auxiliary relations where the to be inserted or deleted tuples are collected. The effect of deletion and insertion is given only be rules like e.g. the frame rule (2.2).

Note further, that without the use of negation nonmonotonic behavior like deletions or modifications could not be expressed. Moreover, without the introduction of states, 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 (cf. section 4).

In example 2.1 a deletion on dept could trigger a deletion on emp but not vice versa. In general, active rules may trigger themselves or each other recursively as is demonstrated by

Example 2.2 (Self-triggering Rules)
If we want to express that whenever managers are deleted, all employees in the departments managed by these are also deleted, along with the departments themselves then the resulting rule may trigger itself recursively (cf. [25]):

```
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)
```

Provided frame rules like (2.2) for emp and dept are given, this is accomplished in Statelog by the following rule:[3]

[3]Logically, a rule with multiple heads is equivalent to several rules with a single head and identical bodies.

$$[S+1]\ (\mathsf{del{:}emp(E1,Sal1,D)},\ \mathsf{del{:}dept(D,E)}) \leftarrow \\ [S]\ (\mathsf{del{:}emp(E,_,_)},\ \mathsf{dept(D,E)},\ \mathsf{emp(E1,Sal1,D)}). \tag{2.3}$$

Assume an initial database state S_0 as depicted in figure 2. 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) holds. This triggers rule (2.3) for the first time which results in the transition to the intermediate state S_1. Due to frame rule (2.2), employee md1 is removed in S_1 while the activated rule (2.3) 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 (2.3) recursively, issuing requests to delete department d2 together with its employees in S_2. Finally, the effect of these events is realized in the new final state of the database S_3.

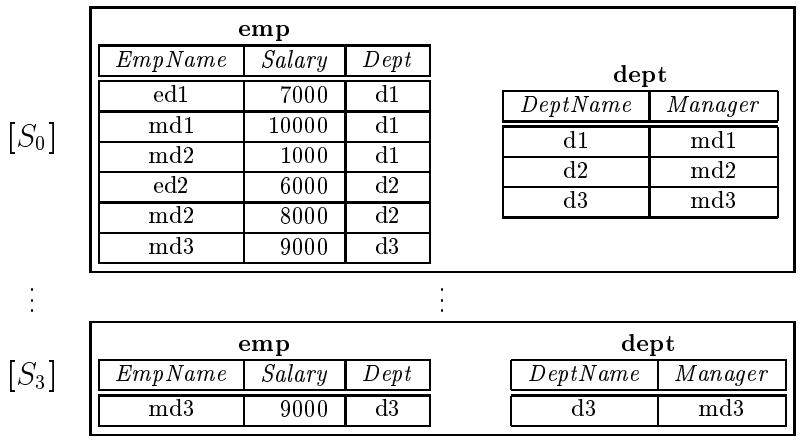

Figure 2: Initial and Final State for Example 2.2

While for this example termination of rule processing can be guaranteed, this is far from obvious in the general case. For example, [4] require the triggering graph of rules to be acyclic or assume some form of monotonic behavior in order to ensure termination.

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[4], progressive rules. The extensional database schema $edb(P)$ consists of the relation symbols occurring only in rule bodies of P. A Statelog *database instance* D (at state s_0) is a finite set of ground atoms A (respectively $[s_0]A$) over $edb(P)$, i.e. a finite set of *facts*. □

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* [18]. 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 del:emp $\overset{(2.1)}{\longleftarrow}$ emp $\overset{\neg(2.2)}{\longleftarrow}$ 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

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

(i) an edge $r \stackrel{0}{\rightarrow} H$

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

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

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

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

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 (2.1) and (2.2). The dependency graph is shown in figure 3. Note, that although there is a negative cycle in $\mathcal{D}_P$ this cycle is not local, so P is Σ-stratified.

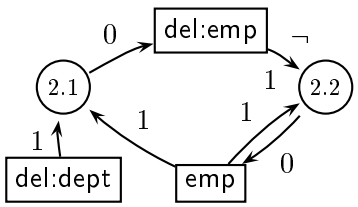

Figure 3: A Labeled Dependency Graph

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 [16]:

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 $T = 1$; for this reason we define:

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, unless explicitly stated otherwise, we assume all Statelog programs P to be Σ-stratified and call $\mathcal{M}$ simply *the model* of $P \cup D$.

3.3 Computation of the Model $\mathcal{M}$

The basic algorithm for computing $\mathcal{M}$ consists of an "outer" iteration proceeding from an already computed state $\mathcal{M}[s]$ to the next state $\mathcal{M}[s+1]$ until a fixpoint is reached, i.e., until $\mathcal{M}[s_f+1] = \mathcal{M}[s_f]$ for some final state s_f. Within the outer iteration there is a fixed number of "local" iterations according to the number of strata of P. Given $\mathcal{M}[s]$ the next state is computed by iterating the rules of P according to the Σ-stratification which is induced by $\mathcal{D}_P$. This computation is a slight modification of the well-known iterated fixpoint procedure [5]:

The rules in a strongly connected component SCC of local cycles correspond to one stratum. Since P is Σ-stratified, no SCC contains a negative dependency. Thus, the partial order on SCCs yields a stratification of rules $L_1, \ldots, L_m$. Finally, $\mathcal{M}[s+1]$ is computed by iterating the rules according to their stratum number until each stratum is saturated.

3.3.1 Termination

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 by storing the complete database history since the last final state and 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. [16, 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. [12]). 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:

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

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 a restricted class of programs for which termination may be guaranteed in advance. Such an approach is taken e.g. in [4] in the context of the Starburst rule system where the triggering graph of a set of rules is required to be acyclic. In [16] the class of *guarded Statelog programs* has been introduced which also guarantee termination by "blocking" certain rules by a base

event in the body. Not surprisingly, such restrictions have an impact on the expressiveness of the language. E.g. the transactions which can be expressed by guarded rules use at most a fixed number of intermediate states *independent of the database D*. Therefore, from a theoretical point of view, guarded rules have the same expressiveness as stratified Datalog.

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 a special class of programs for which termination is guaranteed.

4 A Statelog Framework for Active Databases

On the one hand, transactions have an inherently nonmonotonic 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*.

The basic idea of Δ-monotone Statelog is to separate nonmonotonic initialization and materialization steps of a transaction from a monotonic iteration of Δ-rules inbetween. During the intermediate iteration, every Δ-relation is monotonic, i.e. either nondecreasing or nonincreasing. The overall effect of a transaction is again nonmonotonic. Since the number of intermediate states is not bounded by a constant, Δ-monotone rules are strictly more expressive than the class of guarded rules introduced in [16]. The basic execution model for active rules is as follows (cf. figure 4):

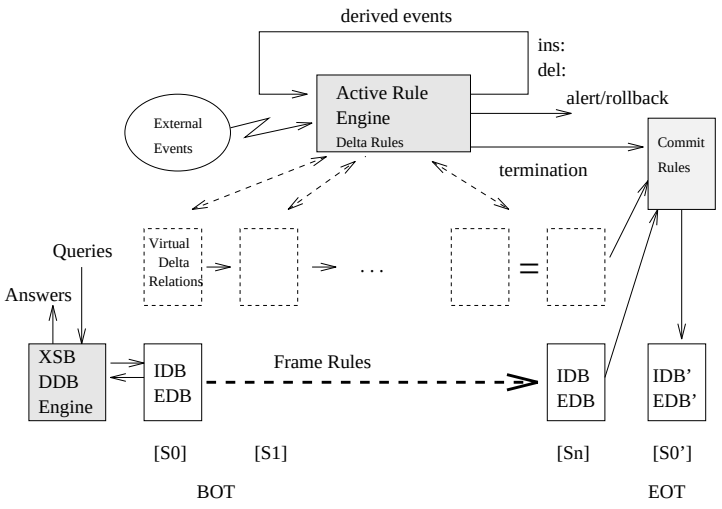

Figure 4: Architecture and Execution Model of an Active Statelog Database System

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. Therefore query rules correspond to the usual (stratified) Datalog rules for defining intensional relations.

Triggered by the occurrence of one or more external *base events*, Δ-*rules* 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. The effect $\mathcal{E}$ of a transition $\mathcal{T}_i$ is the set of inserted or deleted tuples induced by $\mathcal{T}$. As usual, modifications are modeled as simultaneous deletion of the old and insertion of the new, modified tuples.

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.

While the 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 4 implements a *deferred semantics* for updates, the (simplified) examples from section 2.3 with their frame rules correspond to an *immediate semantics* because changes are materialized in base relations as soon as a delete (or insert) occurs. The adoption of an execution model with deferred semantics allows to define an expressive class of terminating programs.

4.1 Transaction Programming in Statelog

In order to illustrate how different behavior can be programmed, we present a variant of example 2.2 using deferred semantics.

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

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

and a *commit rule*

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

For dept analogous rules are used. Finally, the transition rule (2.3) is used without change.

While the events *running* and *terminated* will be implemented as built-ins in a real system, they can be given a clear logic semantics using rules of the form[5]

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

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]Here and in the sequel, $\bar{X}$ denotes a vector $X_1, \ldots, X_n$ of variables.

Note that (4.2) assigns priority to deletions on emp whenever simultaneous insertions and deletions arise. In contrast, by using the scheme

$$[S+1]\ R(\bar{X}) \leftarrow [S]\ (R(\bar{X}), \neg\ \mathsf{del}{:}R(\bar{X}) \vee \mathsf{ins}{:}R(\bar{X}))$$

priority can be given to insertions. Due to the possibility to explicitly refer to different states, static as well as dynamic integrity constraints can be defined in the form of *denials* (see e.g. [7]). For instance, if simultaneous insertion and deletion is considered as an inconsistency, a rule

$$[S]\ \mathit{inconsistent} \leftarrow [S]\ (\mathsf{ins}{:}R(\bar{X}),\ \mathsf{del}{:}R(\bar{X}))$$

may be used. In general, any first-order condition may be expressed in the form of such a denial. Commit rules which materialize the effect of a transaction can use the *inconsistent* event to realize a rollback.

Assumptions In order to embed the described execution model within Statelog, the set of relation symbols of a Statelog database is partitioned into the following disjoint sets:

1. The set $\mathcal{R}_{\bigtriangledown}$ of *base events* which are identified by the first character $\bigtriangledown$ (meaning "occurs"). Base events may not be defined by rules; instead they are asserted as facts in the current state thus modeling external events.
2. The set $\mathcal{R}_{EDB}$ of extensionally defined relations not occurring in the head of any query rule.
3. The set $\mathcal{R}_{IDB}$ of intensionally defined relations.
4. The set $\mathcal{R}_{\Delta}$ of Δ-*relations*. Δ-relations are used to accumulate the desired insertions or deletions. As for intensional relations, Δ-relations must be defined by rules. We use terms of the form ins:R, del:R, mod:R to denote Δ-relations.

We assume that for all relations from $\mathcal{R}_{EDB}$ and $\mathcal{R}_{IDB}$ an unconditioned frame rule (cf. (4.1)) is used. A *query rule* is a local Statelog rule where the predicate symbol in the head is in $\mathcal{R}_{IDB}$. All other rules are *transition rules*, i.e. frame rules, Δ-*rules* (rules with a Δ-relation in the head) and control rules which are used to start, rollback or commit transactions.

4.2 Δ-Monotone Statelog

Using the assumptions for our overall architecture as described above, we now define the class of Δ-monotone Statelog programs which guarantee termination for all databases.

Definition 4.2 (Δ-monotone Statelog) Let P be a Σ-stratified Statelog program and $\mathcal{D}_P$ its labeled dependency graph. P is called Δ-*monotone* if for all Δ-relations $R \in \mathcal{R}_{\Delta}$ and every cycle C of $\mathcal{D}_P$ in which R occurs

(i) C contains an *even* number of negations.

(ii) $\sum_{e \in C} l_e \leq 1$, where l_e is the leap of an edge e in C.

(iii) R depends only on Δ-relations from the same strongly connected component or from relations in $\mathcal{R}_{EDB}$ and $\mathcal{R}_{IDB}$.

□

(i) ensures that all relations are positive wrt recursive triggering, while (ii) states that each cycle corresponds to exactly one state transition. (iii) implies that Δ-relations depend on constant relations or relations from the same "iteration block" only.

Note that the modified example 4.1 is Δ-monotone (2.1 can be adapted analogously). Relaxing any of the above requirements may lead directly to non-terminating programs:

Example 4.3 Let q,r,u,o be Δ-relations. Consider the following programs, each violating a single condition of definition 4.2:

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

Given an empty database D as input, both $P_{(i)}$ and $P_{(ii)}$ do not terminate since q (and r) oscillate between true and false in successive states.

For $P_{(iii)}$ assume that for m but not n an unconditioned frame rule is given. Let the initial database $D = \{\mathsf{n(a)}, \mathsf{m(a,b)}, \mathsf{m(b,c)}, \mathsf{m(c,a)}\}$ be given. n is not constant but changes: $[0]$ n(a) is true while $[S]$ n(a) is false for all $S \geq 1$. It is easy to see that $P_{(iii)}$ does not terminate since $\mathcal{M}$ has period $T = 3$.

When the last rule is removed from $P_{(iii)}$ or n is propagated by a frame rule, the resulting program constitutes a non-trivial example of a Δ-monotone program: u is monotonically increasing while o is monotonically decreasing.

Theorem 4.4 *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[6] 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. ■

4.3 Expressiveness & Complexity

We now present results characterizing the expressiveness and complexity of Statelog. Since Statelog distinguishes between local query rules and state-changing transition rules, two notions of expressiveness arise (both of which are based on the usual framework [8]). The

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

first, *Statelog query expressiveness* is given by only considering local rules. As these correspond to stratified Datalog, the expressiveness of Statelog queries also corresponds to stratified Datalog. The second, *Statelog transaction expressiveness* is given by 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.

It can be shown [14] that Statelog rules may be encoded into equivalent *while programs* and vice versa. Similarly (cf. theorem 4.4), 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, the following expressiveness results can be claimed:

Theorem 4.5

- *Statelog is equivalent to* WHILE
- Δ*-monotone Statelog is equivalent to* FIXPOINT

A consequence of this encoding is that 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, respectively).

Focusing only on expressive power, one might ask for the motivation for another formalism expressing FIXPOINT or WHILE, respectively. From a theoretical point of view, it is interesting to investigate the impact of states on Datalog since these results can be applied to related approaches like XY-stratified Datalog [26]. From a practical point of view, there is another important advantage of Statelog: We argue that the interaction between negation and recursion is more intuitive and comprehensible when considering states — a fact that can also be attributed to the underlying "declarative" semantics of Statelog. This is in contrast to other formalisms which involve only a "one-state logic": e.g., although closure and complement are fundamental notions expressible in fixpoint logic (by recursion and negation), the combination of both may lead to somewhat awkward and hard to understand programs. In particular, the complement of the transitive closure can be expressed in the obvious way in stratified Datalog (and therefore in Statelog) but involves a rather complicated definition in inflationary Datalog$^\neg$ (see e.g. [3, page 96]).

4.3.1 Computing Well-Founded Views with Statelog

To substantiate the claim that the interplay between negation and recursion using states is "well-behaved" and intuitive, we give a surprisingly simple and elegant coding of well-founded models [21] in Statelog. The alternating fixpoint characterization of the well-founded model [20] 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 4.6 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 $\mathsf{move} = \{(a,b),(b,a),(b,c),(c,d)\}$ the following non-terminating computation results:

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

In accordance with [20], even numbered states correspond to a nondecreasing sequence of underestimates while odd numbered states represent a nonincreasing sequence of overestimates of true atoms. It is easy to see that, independent of the database D

(i) P_{afp} terminates iff the well-founded model of P is total (a property which is undecidable).

(ii) A period $T \leq 2$ is always attained.

(ii) allows to give a Δ-monotone encoding P'_{afp} of well-founded views. Indeed, discarding the last (Δ-monotony destroying) rule in $P_{(iii)}$ of example 4.3 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.

4.4 Implementation of Statelog

The semantics of Statelog is defined model-theoretically, therefore it is independent of a specific evaluation strategy like top-down or bottom-up. Thus, optimized rule processing can be accomplished by adopting a suitable strategy for each kind of rules.

More precisely, we propose to use goal-oriented backward chaining strategies like SLG-resolution [9] or the family of Magic sets rewriting techniques for query rules (cf. figure 4) since these techniques focus on potentially relevant tuples wrt to a given query.

On the other hand, propagation of changes as defined by Δ-rules can be favorably achieved by forward chaining execution. Consequently, techniques known from the area of active databases (cf. figure 1) or production rule systems (like e.g. TREAT and Rete, [22]) should be useful for efficient testing of rule conditions in Δ-rules.

A prototypical implementation of the Statelog architecture in figure 4 is underway [14]. It uses XSB-Prolog as an efficient query engine [19]. Since state terms are "compiled away" there is no overhead in query processing.

5 Related Work

In [26] a similar approach, *XY-stratified Datalog* is proposed as a unified semantics for deductive and active databases. X and Y-rules 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[7], it can be shown that already "pure" XY-stratified programs (i.e. without extensions like nondeterministic choice) can express WHILE. As a consequence, undecidability of termination applies to these XY-stratified programs as well and 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 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.

[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. Especially, 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 proposed a unified logical framework for state-changing active and passive query rules thus providing at the same time intuitive deterministic semantics for active rules as well as increased expressiveness and control capabilities for rule execution. Generalizing previous results, the class of Δ-monotone programs has been presented which guarantee termination for all inputs — a property that has been proved 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 active rules.

We argue that Statelog can be efficiently implemented by applying different state-of-the-art techniques, i.e. (i) goal-oriented strategies for query rules and (ii) forward chaining execution for transition rules.

In a sense, Statelog is a "minimal" extension of Datalog allowing just as much expressiveness as is necessary to express updates and transactions in a declarative way by using states. Further extensions to the "pure" language kernel are possible and necessary for real world applications. Although such extensions usually can be integrated into an implemen-

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

tation with relative ease, their theoretical implications remain an interesting field of future research. One possibility e.g. is to allow nondeterministic extensions (like invented values or choice, see e.g. [27]) which are a powerful programming tool and have a similar impact on expressiveness like assuming an ordered database.

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] 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.

[5] K. Apt, H. Blair, and A. Walker. Towards a theory of declarative knowledge. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 89 – 148. Morgan Kaufmann, 1988.

[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] F. Bry. Towards intelligent databases. In *Proc. Intl. Symposium on Methodologies for Intelligent Systems*, LNCS. Springer, 1993.

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

[9] W. Chen and D. S. Warren. Query evaluation under the well founded semantics. In *Proc. ACM Symposium on Principles of Database Systems*, 1993.

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

[13] 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.

[14] 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.

[15] 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.

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

[17] 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.

[18] 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.

[19] 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.

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

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

[22] Y.-W. Wang and E. N. Hanson. A performance comparison of the rete and treat algorithms for testing rule conditions. In *Proc. IEEE Intl. Conference on Data Engineering*, 1992.

[23] J. Widom. Deductive and active databases: Two paradigms or ends of a spectrum. In Paton and Williams [17].

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

[25] 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.

[26] C. Zaniolo. A unified semantics for active and deductive databases. In Paton and Williams [17].

[27] 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.