Reihe Informatik
1 / 1994

Updates by Reasoning about States

Georg Lausen Bertram Ludäscher

Updates by Reasoning about States

Georg Lausen Bertram Ludäscher

Universität Mannheim, Seminargebäude A5
68131 Mannheim, Germany
{lausen,ludaesch}@pi3.informatik.uni-mannheim.de

Abstract

It has been argued that some sort of control must be introduced in order to perform update operations in deductive databases (see e.g. [1, 17]). Indeed, many approaches rely on a procedural semantics of rule based languages and often perform updates as side-effects. Depending on the evaluation procedure, updates are generally performed in the body (top-down evaluation) or in the head of rules (bottom-up evaluation).

We demonstrate that updates can be specified in a purely declarative manner using standard model based semantics without relying on procedural aspects of program evaluation. The key idea is to incorporate *states* as first-class objects into the language. This is the source of the additional expressiveness needed to define updates. We introduce the update language *Statelog*$^{+-}$, discuss various domains of application and outline how to implement computation of the perfect model semantics for *Statelog*$^{+-}$ programs.

1 Introduction

When modelling a dynamically changing world or revising knowledge about an application domain, databases must reflect these dynamics. In this context, updating a database is an issue of fundamental importance.

Deductive databases extend traditional relational systems and have become attractive in the past for several reasons. By describing relations intensionally, i.e. as rules, a more concise and conceptually clear representation is attained, which is often more user-friendly and facilitates database maintenance. These advantages are already known from view definitions of relational systems. In addition, the rule language of a deductive database extends the expressive power of traditional query languages and their view mechanisms by allowing recursive definitions. On the other hand, rule languages can also be used as database programming languages. Because of their logical foundation they allow programming on a high level of abstraction, thereby relieving the programmer of many details of implementation and optimization.

Despite its popularity, the most prominent deductive database language *Datalog* lacks concepts to specify the dynamics of updates: the model of a *Datalog* program is static. Existing extensions of *Datalog* often rely on procedural aspects of program evaluation by performing updates in the body (top-down evaluation, e.g. [16, 17]) or in the head of rules (bottom-up evaluation, e.g. [2, 9]), sometimes as side-effects.

By relying on procedural semantics, a major advantage of *Datalog*, its declarative semantics is usually sacrificed. However, a procedural semantics for updates does not inevitably exclude a declarative counterpart: e.g. in [16] a declarative semantics based on Kripke-structures is proposed. In fact, by leaving the simple model theoretic framework of *Datalog* declarative foundations of updates can be achieved. Examples for this trend are the more recent approaches of [20, 11, 5] (see also [25] for a survey of many other approaches). In contrast, we show that it is possible to specify update operations for databases in a purely declarative way using a language which closely resembles *Datalog*.

We introduce *Statelog*$^{+-}$, a *Datalog*-like language, which allows updating a database in a state-oriented way. We continue work started in [13], where a preliminary rule-based update language was suggested, which could only process a fixed number of states (because states had to be denoted by *constants*). In contrast, *Statelog*$^{+-}$ is a much more general state-oriented language which imposes less restrictions on states. In *Statelog*$^{+-}$ states may be denoted by *variables* which range over a potentially infinite set of states. Furthermore, while the semantics of the language in [13] was defined as the fixpoint of a modified immediate consequence operator, *Statelog*$^{+-}$ has a clean model-theoretic semantics thereby allowing the study of formal properties of programs and their implementation based on standard deductive database technology. *Statelog*$^{+-}$ has several desirable properties, e.g., updates are always deterministic and consistent. Evaluation of update programs can be done either bottom-up, e.g. implemented as a front-end to *CORAL* [19], or top-down, e.g. implemented as a front-end to *XSB-Prolog* [21].

The structure of the paper is as follows. In section 2 we exhibit the main underlying ideas of *Statelog*$^{+-}$. We continue with a discussion of various examples in section 3. Section 4 presents syntax and semantics of *Statelog*$^{+-}$ in a more formal manner. The perfect model semantics and its computation are covered in sections 5 and 6. We conclude with a short summary and outlook on future work.

2 Basic Ideas

In this section we try to clarify the basic ideas of our work by means of informal arguments. A formal presentation will be given subsequently.

We state updates in the head of rules. Such *update rules* can be read as follows: "if the body of the rule is true, then the specified update operation should be performed". Possible update operations are *insert* and *delete*. In such a framework, without any additional means, it is easy to write inconsistent programs. Let "+" denote an insert and "-" a delete operation. Then the following simple program contains conflicting update requests:

$$\begin{array}{l}[-]\ \mathsf{q(a)} \leftarrow \mathsf{p(a)}.\\ [+]\ \mathsf{q(a)} \leftarrow \mathsf{p(a)}.\end{array}$$

If p(a) is true in the database, then q(a) should be deleted and inserted at the same time, which clearly has to be considered as an inconsistent situation. An obvious solution would be to assign different priorities to *insert* and *delete*, and generally prefer one operation to the other. However, this approach is not compatible with a declarative reading of the rules. We therefore propose a different solution which preserves declarativeness.

Assume that the database is organized in different *states*. Let us denote the initial state of the database by $[\epsilon]$. Then the inconsistent situation in the above example can be removed by assuming two further states in addition to the initial state: a state denoted by $[+\sigma_1]$, to which the insert is directed, and a state $[-\sigma_2]$, to which the delete is directed. Since $[+\sigma_1]$ and $[-\sigma_2]$ are distinct states, no update conflict may arise. The question remains how to relate these different states.

A simple, yet general concept is to organize states in an *alternating* sequence of *insert* and *delete states*: updates of one type (i.e., insert or delete) may be accumulated in the same state. As soon as an update of the other type has to be performed, a transition of the database to a corresponding new state is necessary.

The ordering of states can be made explicit by denoting states as alternating strings of "+" and "-": $[\epsilon]$, [-], [+-], [-+-], [+-+-], etc. Delete states like [-], [-+-], ... have "-" as their leftmost character, while "+" is the leftmost character of insert states [+-], [+-+-],... (additionally, the initial state $[\epsilon]$ is an insert state). The correspondence between states and update operations is straightforward: if the current state is an insert state, tuples can only be inserted into relations, while in delete states only deletions are possible. Obviously, by directing insert and delete operations to different states, inconsistent situations become impossible. In addition, the order in which update rules are applied during evaluation does not affect the result of the updates. In this way we achieve *determinism* in an area where usually *nondeterminism* prevails.[1]

Returning to our previous example, we may choose to let $\sigma_1 = -$ and $\sigma_2 = \epsilon$. Thus [+-] is the final state of the update, while [-] is only an intermediate state. After modification of the first rule, the rules can be read as:

- Delete q(a) in state [-].
- Insert q(a) into state [+-].

[1] Rule-languages, which allow *insert* and *delete* operations are usually nondeterministic. A well known example is OPS5.

It is crucial to understand the difference in comparison to the aforementioned ad-hoc priorities. Since such priorities are defined outside the language, they can only be considered during *evaluation* of rules. Therefore, they are a *procedural* means. In contrast, we treat states as *first-class objects* by incorporating them into the semantics of our language. In this way, rules can be given a *declarative* reading.

In addition to the explicit insert and delete operations, states are related by a *frame axiom* which intuitively says: whenever something is true in some insert state and it is not explicitly deleted in the next delete state then it will also be true in the next insert state. Thus, tuples which have been inserted in some insert state (including the initial state) are propagated into subsequent states, until a deletion prevents further propagation. The final state will then consist of the updated relations.

A set of update rules is called an *update program*. What is the *result* of such a program, i.e. what is the new state of the database? Every update program defines a sequence of states. The final state of such a sequence is taken as the result of the whole program and is used to replace the initial state of the database. Thus, *after* having evaluated the program on intermediate, temporary states, the update can be materialized. As will be seen later, this simple, intuitive approach is not sufficient when using variables to denote states. Then an update program may refer to arbitrarily many states. Fortunately, the notion of a *final state* can still be given a well-defined meaning, as we will show later in the paper.

In the following sections we shall introduce $\mathit{Statelog}^{+-}$, a variant of *Datalog* with negation in rule bodies, extended by a facility to explicitly manipulate states in the aforementioned way. We discuss two variants of $\mathit{Statelog}^{+-}$. The first variant is called $\mathit{Statelog}^{+-}_{\rightharpoonup}$. Its distinctive feature is that reasoning is *progressive*, i.e. directed from "past" to "future" states: in $\mathit{Statelog}^{+-}_{\rightharpoonup}$ the state referred to in the head of a rule is no "earlier" in the ordering of states than any of the states in the body of that rule. The other variant, called $\mathit{Statelog}^{+-}_{\rightleftharpoons}$, extends $\mathit{Statelog}^{+-}_{\rightharpoonup}$ to allow full temporal reasoning about states. In $\mathit{Statelog}^{+-}_{\rightleftharpoons}$ one can also reason backwards in time, i.e. from states to preceding states. In the context of database updates it is generally not necessary and often unnatural to change the history of the database by reasoning backwards in time. Therefore we shall concentrate on the progressive language $\mathit{Statelog}^{+-}_{\rightharpoonup}$.

3 $\mathit{Statelog}^{+-}$ by Example

We start by discussing several examples of updates on extensional (EDB) relations. A typical update operation is *modification*: for some tuples, an existing value has to be modified. The declarative counterpart of modification is insertion of the "new" (updated) tuples and retraction of the "old" tuples.

Consider the proverbial and oversimplified *employee-salary* example. Assume the EDB contains facts es(E,S) describing the *initial* employee-salary relation, i.e. in state $[\epsilon]$. If we want to increase the salary of all employees by 5%, we can simply write the following insert rule:

$$[+-]\ \mathsf{es(E,S1)} \leftarrow [\epsilon]\ \mathsf{es(E,S)},\ \mathsf{S1 := 1.05 * S}.$$

Because of the built-in frame axiom, a modify operation additionally requires for removal of old tuples, hence a corresponding delete operation has to be defined. This is accomplished by the following delete rule

$$[-]\ \mathsf{es(E,S)} \leftarrow [\epsilon]\ \mathsf{es(E,S)}.$$

A more concise notation can be achieved by allowing heads with multiple atoms. The two rules above can be rewritten into a single rule:

$$\begin{aligned}&[+-]\ \mathsf{es(E,S1)},\ [-]\ \mathsf{es(E,S)} \leftarrow \\ &\qquad [\epsilon]\ \mathsf{es(E,S)},\ \mathsf{S1 := 1.05 * S}.\end{aligned} \tag{3.1}$$

Then this rule declaratively describes a modify operation. Notice further that the use of arithmetic in this rule is *safe*. This may not be self-evident since the rule is recursive in the relation es. However, only a finite number of new states — in this case exactly one state [+−] — is created.

In general, rules with arithmetic may become unsafe as is indicated by the following extended example

$$\begin{aligned}&[+-\lambda]\ \mathsf{es(E,S1)},\ [-\lambda]\ \mathsf{es(E,S)} \leftarrow \\ &\qquad [\lambda]\ \mathsf{business\_boom}, \\ &\qquad [\lambda]\ \mathsf{es(E,S)},\ \mathsf{S1 := 1.05 * S}.\end{aligned} \tag{3.2}$$

Here λ is a variable ranging over arbitrary insert states. Moreover, assume that this rule is part of a larger update program, in which business_boom is defined by some other update rules, which are not of interest here. As long as the business booms at finitely many states (which seems to be a reasonable assumption), only finitely many new salary amounts are created. Let us add the fact

```
[ϵ] business_boom.
```

which expresses that initially, i.e. in state [ϵ], business booms. However, the frame rule for business_boom propagates this fact to all subsequent states. This results in triggering infinitely many salary increases and the generation of infinitly many growing salary amounts, which is clearly not desired (not only from the employers point of view, but also because it is unsafe).

For relations like business_boom which are true only at certain states, propagation by the frame rule has to be "disabled". This can be simply achieved by a corresponding delete rule:

```
[-λ] business_boom ← [λ] business_boom.
```

The rule guarantees that business_boom only becomes true in those states, where a corresponding insert rule explicitly supports this.

The next example shows how *Statelog*$^{+-}$ can be used for some kind of *hypothetical reasoning*. We intend to determine whether after a hypothetical (non-linear) salary-raise to all employees, the employee peter would be the highest-paid employee of the enterprise:

```
[+-] es(E,S1), [-] es(E,S) ←
        [ϵ] es(E,S),
        [ϵ] factor(E,F), S1 := S * F.

[+-+-] es(E,S), [-+-] es(E,S1) ←
        [+-] es(E,S1), [ϵ] es(E,S).

[+-] highest_paid(peter) ←
        true.
[-+-] highest_paid(peter) ←
        [+-] es(E,S),
        [+-] es(peter,S1), S > S1.
```

Here the first two rules realize the hypothetical salary-raise by performing and revising it right away. For each employee the salary in state [+-+-] is identical to the salary in the initial state. The hypothetically raised salary is contained in state [+-]. The third and fourth rule determine — by referring to the state of the hypothetical raise — whether *peter* would be the highest paid employee of the enterprise. Note, that in the final state [+-+-] the initial salaries are valid; in addition, if it is not retracted by the fourth rule, highest_paid(peter) is valid.

In the next example, we do not know in advance how many states are required to perform the update. The program simulates a version of the *game of life* (cf. [2]): a graph is defined using the relation edge. The nodes of the graph correspond to a set of cells, some of which are initially "alive", while the edges are used to describe the neighborhood relation between cells. Let the initial state in the database contain the edges {(a,b), (a,c), (a,d),(b,c),(c,d)} (represented as facts edge(X,Y)) and a relation alive to record the living cells. Assume that initially alive(b) and alive(c) holds.[2]

[2]The use of round brackets in this example should be self-explanatory.

```
% define the neighborhood relation
[ε] neighbor(X,Y) ← [ε] edge(X,Y).
[ε] neighbor(X,Y) ← [ε] edge(Y,X).

% ≥ 3 neighbors ⇝ cell dies
[-λ] alive(C) ←
        [λ] (alive(C), has3neighbors(C)).

% = 2 neighbors ⇝ cell is born
[+-λ] alive(C) ←
        [λ] has2neighbors(C),
        [λ] ¬ has3neighbors(C).

% cell has ≥ 2 living neighbors
[+λ] has2neighbors(C) ←
        [ε] neighbor(C,N1),
        [ε] neighbor(C,N2),
        N1 ≠ N2,
        [+λ] (alive(N1), alive(N2)).

% cell has ≥ 3 living neighbors
[+λ] has3neighbors(C) ←
        [ε] neighbor(C,N1),
        [ε] neighbor(C,N2),
        [ε] neighbor(C,N3),
        N1 ≠ N2, N2 ≠ N3, N1 ≠ N3,
        [+λ] (alive(N1), alive(N2), alive(N3)).

% disable frame-rule for has2neighbors
[-λ] has2neighbors(C) ←
        [λ] has2neighbors(C).

% disable frame-rule for has3neighbors
[-λ] has3neighbors(C) ←
        [λ] has3neighbors(C).
```

The update to be performed is to compute the final state of the game in the database. To fully understand this example, the reader should observe how the frame axiom acts here. We provide rules to capture birth resp. death of cells. The frame axiom guarantees that a cell will stay alive once it has been born until a corresponding rule states the death of a cell. On the other side, the last two rules disable the frame axiom with respect to has2neighbors and has3neighbors because these relations have to be computed anew for each state. We could also disable the frame axiom with respect to neighbor and edge; this might be reasonable to keep new states as small as possible.

It is easy to verify that a naïve bottom-up evaluation of the rules will not terminate because the cell-culture will grow and shrink indefinitely. By generalizing a result in [8] we will show later, that the perfect model of a stratified *Statelog*$^{+-}$ program is ultimately periodic. Therefore, we can take the first occurrence of the period as a *generalized final state* of the database. In our example, the period is defined by the states [ϵ],[-] and [+-].

So far, we have focused on updates to extensionally defined relations. However, *Statelog*$^{+-}$ can be equally well applied for updating intensionally defined (IDB) relations. This is due to the fact that no distinction is made whether the to be updated relation in the head of a rule is extensionally defined or intensionally via rules. In the previous example, we have already updated IDB-relations, e.g. the relation has2neighbors. The way updates are defined by *Statelog*$^{+-}$ is close in spirit to deterministic updates as described in [14] for a language operating on a single state only: insertions and deletions of facts can be seen as (positive or negative) *exceptions* to the rules defining an IDB-relation. In fact, we can apply the same ideas to materialize updates as presented there. A detailed discussion is beyond the scope of this paper and will be presented in a forthcoming paper.

4 A *Statelog*$^{+-}$ Framework

We review some basic terminology and fix our notation. The syntax and a general framework for the semantics of *Statelog*$^{+-}$ is given. As an important example we define the *perfect model* semantics for *Statelog*$^{+-}$ in section 5.

4.1 Syntax

We adopt standard logic programming and deductive database terminology and assume the reader is familiar with the basic concepts. The new concepts of *Statelog*$^{+-}$ are defined first.

Definition 4.1 (States)
The sets of *insert states* Σ^+ and *delete states* Σ^- are defined as

$$\begin{array}{lcccl} \Sigma^+ & = & (\texttt{+-})^* & = & \{\epsilon, \texttt{+-}, \texttt{+-+-}, \ldots\} \\ \Sigma^- & = & \texttt{-}(\texttt{+-})^* & = & \{\texttt{-}, \texttt{-+-}, \texttt{-+-+-}, \ldots\} \end{array}$$

The set of all states $\Sigma = \Sigma^+ \cup \Sigma^-$ can be identified with the set of natural numbers $\mathcal{N}_0$ via the bijection

$$\sigma \in \Sigma \mapsto |\sigma| \in \mathcal{N}_0$$

where $|\sigma|$ denotes the length (i.e., the number of +'s plus the number of -'s) of the representation of $\sigma \in \Sigma$. When we use a state σ in an arithmetical expression, we refer to the natural number $|\sigma|$. Obviously, Σ^+ and Σ^- correspond to the sets of even and odd numbers, respectively.

Every state $\sigma \neq \epsilon$ is determined by the number of occurrences of its leftmost character. Therefore we can use a *shorthand notation* where we leave out the intermediate characters: e.g. +-+- can be abbreviated to ++.

Definition 4.2 (State-Terms)
Let $\Lambda = \{\lambda_1, \lambda_2, \ldots\}$ be the set of *state-variables*, which is distinct from the set of "ordinary" *data-variables* $\mathcal{V}$.

A *state-term* τ can be either a state σ, or a state-variable which is preceded by a *prefix* π:

$$\tau = \begin{cases} [\sigma] & ; \quad \sigma \in \Sigma \\ [\pi\lambda] & ; \quad \pi \in \Pi, \lambda \in \Lambda \end{cases}$$

The set of possible prefixes is $\Pi = \Sigma \cup \Sigma.\texttt{+} = \{\epsilon, \texttt{-}, \texttt{+}, \texttt{-+}, \texttt{+-}, \ldots\}$.

Depending on its particular form a state-term τ represents a (possibly infinite) set of states, its *extension* $[\![\tau]\!]$. Let σ be a state, λ a state-variable and π a prefix, then we define

$$\begin{aligned} [\![\sigma]\!] &= \{\sigma\} \\ [\![\pi\lambda]\!] &= \{\pi\sigma \mid \pi\sigma \in \Sigma\} \end{aligned}$$

Note that $[\pi\lambda]$ denotes *either* a set of insert states *or* a set of delete states if $\pi \neq \epsilon$, i.e., either $[\![\pi\lambda]\!] \subseteq \Sigma^+$ or $[\![\pi\lambda]\!] \subseteq \Sigma^-$. Therefore we have $[\![\pi\lambda]\!] = \{\sigma + 2k \mid k \in \mathcal{N}_0\}$ for some state σ.

The shorthand notation for states is extended to include state-terms. An abbreviation denotes the smallest "proper" state-term which can be built from the abbreviation, for instance one can write [--λ] instead of [-+-λ], or [++-λ] instead of [+-+-λ] etc.

Definition 4.3 (*Statelog*$^{+-}$)
A *Statelog*$^{+-}$ program P is a finite set of *rules*. A rule r is of the form

$$\tau_0 A_0 \leftarrow \tau_1 B_1, \ldots, \tau_n B_n$$

where A_0 (the *head* of the rule) is an atom.[3] $B_1, \ldots, B_n$ are literals (the *body* of r), $\tau_0, \ldots, \tau_n$ are state-terms. r is called a *fact* if the body is empty, i.e. if $n = 0$. The *extensional database* (EDB) of P consists of those predicates which occur only as facts, all other predicates belong to the *intensional database* (IDB).

[3]We do not consider negation in the head of rules like in *Datalog*$^{\neg *}$ [2]. This notion of deletion is replaced by our delete rules.

A *literal* is either an atom $p(t_1, \ldots, t_m)$ or a negated atom $\neg p(t_1, \ldots, t_m)$, where the *terms* t_i are constants or *data-variables* of $\mathcal{V}$. Especially, state-variables may not occur inside literals.

We say r is an *insert rule* if the state-term τ_0 in the head of r denotes insert states (i.e., if $[\![\tau_0]\!] \subseteq \Sigma^+$), otherwise r is a *delete rule* (since τ_0 denotes delete states).

Note that r is a *Datalog*$^{\neg}$ rule [22], if all τ_i are equal to ϵ. Therefore *Statelog*$^{+-}$ is a proper extension of *Datalog*$^{\neg}$.

In order for a *Statelog*$^{+-}$ rule r to be correct, we have to ensure that all occurrences of state-variables in r are well-defined and consistent with each other. This syntactical notion is captured by the following definition.

Definition 4.4 ($^{+-}$Correct Rules)
A rule r is $^{+-}$*correct* or *correct* for short if for each state-variable λ occurring in r

1. there is a state-term $[\pi\lambda]$ in r, such that $\pi \neq \epsilon$, and
2. $[\![\pi\lambda]\!] \cap [\![\pi'\lambda]\!] \neq \emptyset$ for any two state-terms $[\pi\lambda]$, $[\pi'\lambda]$ occurring in r.

Condition 1 states that an "undefined" rule like $[\lambda]$ $p \leftarrow [\lambda]$ q is not correct: it is not clear whether p should be inserted or deleted.

In contrast, condition 2 circumvents "inconsistent" rules like [+λ] $p \leftarrow$ [-λ] q, where it is impossible to instantiate λ such that both [+λ] and [-λ] denote valid states.

4.2 Semantics

We outline a general framework for the two versions of *Statelog*$^{+-}$.

Updates and *Statelog*$^{+-}_{\_}$ In the context of database updates, the evolution of a database over time is modelled with states. Therefore, it is reasonable to define a new state using knowledge about previous states. In contrast, as soon as a new state has been introduced, it should not be allowed to change past states (i.e. by insertions or deletions) nor should it be possible to make assumptions about future states. Both requirements are met as long as rules are interpreted *progressively*.

Then the intuitive reading of a rule like $[+\lambda_1]\ p \leftarrow [-\lambda_2]\ q$ is

> "Insert p into state $[+\lambda_1]$ provided q has been deleted in a state $[-\lambda_2]$ which *precedes* $[+\lambda_1]$."

Progressiveness seems to be a very natural prerequisite for traditional update policies. The language where the meaning of rules is defined by their *progressive* extension $[\![r]\!]_{\_}$ (see def. 4.5) is called *Statelog*$^{+-}_{\_}$.

Temporal Deductive Databases and *Statelog*$^{+-}_{=}$ In the context of *temporal deductive databases*, it is necessary to abandon the restriction of progressiveness in order to reason forward and backward in time. Then the meaning of the above rule becomes

> "Insert p into state $[+\lambda_1]$ provided q is deleted in *some* state $[-\lambda_2]$."

The resulting language is called *Statelog*$^{+-}_{=}$ and can be conceived as a *temporal deductive database* language in the sense of [4].

We use the term *Statelog*$^{+-}$ to refer to both versions of the language.

The Frame Axiom A basic requirement for a semantics *SEM* to be suitable for our state-oriented approach is that *SEM* is compatible with the following *frame axiom:*

> If a proposition (tuple) p is true in some insert state σ, and it is *not* marked for deletion in $-\sigma$ (i.e., $[-\sigma]\ p$ is false) then p must still hold at the *next* insert state $+-\sigma$.

Possible candidates for *SEM* include the *perfect model semantics* [18], the *well-founded semantics* [24] and the *stable semantics* [10], respectively. Since it can be computed efficiently in our context, the perfect model semantics is of particular interest (sections 5 and 6).

Definition 4.5 (Extensions of Rules)
A rule r without state-variables, i.e., of the form

$$[\sigma_0]\ A_0 \leftarrow [\sigma_1]\ B_1, \ldots, [\sigma_n]\ B_n$$

where all σ_i are states is called Σ-*grounded*.

A Σ-grounded rule is called *progressive* if $\sigma_0 \geq \sigma_i$ for $i = 1, \ldots, n$; *strictly progressive* if $\sigma_0 > \sigma_i$.

A Σ-grounded rule r^* which is obtained from a rule r by substituting state-variables is called a Σ-*instance* of r.

The *full extension* $[\![r]\!]_{=}$ of a correct rule is the set of all Σ-instances which can be constructed from r. We say that r is *strictly progressive* if all rules in $[\![r]\!]_{=}$ are strictly progressive.

The *progressive extension* $[\![r]\!]_{\_}$ of r is the subset of *progressive* rules of $[\![r]\!]_{=}$.

If $[\![r]\!]_{\_} \neq \emptyset$ then r is called *(potentially) progressive*.[4]

We write $[\![r]\!]$, whenever we want to refer to both $[\![r]\!]_{=}$ and $[\![r]\!]_{\_}$.

Example 4.6
The rule $r : [+\lambda]\ p \leftarrow [-+\lambda]\ q$ is correct but not potentially progressive, since $[\![r]\!]_{\_} = \emptyset$.

On the other hand, $r' : [++]\ p \leftarrow [-\lambda]\ q$ is potentially progressive, since

$$[\![r']\!]_{\_} = \{[++]\ p \leftarrow [-]\ q, [++]\ p \leftarrow [--]\ q\}.$$

No other Σ-instance of r' is progressive.

The Semantical Framework The language $\mathcal{L}_P$ of a *Statelog*$^{+-}$ program P is determined by the set of constants $\mathcal{C}$ and predicate symbols $\mathcal{P}$ occurring in P.

The *Herbrand universe* $\mathcal{U}_P$ and the *Herbrand base* $\mathcal{B}_P$ are defined as usual, i.e., $\mathcal{U}_P = \mathcal{C}$ and $\mathcal{B}_P$ is the set of ground atoms that can be constructed from $\mathcal{U}_P$ and $\mathcal{P}$. Note that we do not consider states nor state-terms τ_i of P to be part of $\mathcal{U}_P$. This is to exclude "ill-typed" atoms like $p(-a, +, b)$ from $\mathcal{B}_P$. Another possibility is to define a two-sorted language $\mathcal{L}_P^{\Sigma}$, with Σ and the state-variables Λ belonging to one sort and the "ordinary" constants $\mathcal{C}$ and variables $\mathcal{V}$ to the other sort.

Since we generally disallow the use of function symbols[5], $\mathcal{U}_P$ and $\mathcal{B}_P$ are finite.

Definition 4.7 (Σ-Interpretation)
A Σ-*interpretation* I^{Σ} is a mapping from Σ to

[4]In the context of *Statelog*$^{+-}_{\_}$, we can leave out "potentially" (since $[\![r]\!]_{\_}$ is the "meaning" of r) and simply call r *progressive*.

[5]This restriction can be relaxed to include a *safe* use of function symbols.

$2^{\mathcal{B}_P}$ which is compatible with the frame axiom in the following sense. Each state $\sigma \in \Sigma$ is assigned a Herbrand interpretation $I^\sigma \subseteq \mathcal{B}_P$

$$I^\Sigma : \sigma \mapsto I^\sigma$$

such that

$$I^{{+}{-}\sigma} \quad \supseteq \quad I^\sigma \setminus I^{-\sigma}. \tag{4.1}$$

Sometimes I^σ is called the *snapshot* of I^Σ at σ. A (possibly infinite) sequence of successive snapshots $I^\sigma, I^{\sigma+1}, \dots, I^{\sigma+w-1}$ where $w \in \mathcal{N}_0 \cup \{\infty\}$ is called a *window* of I^Σ. It is denoted $I^{\langle\sigma,w\rangle}$, where σ is the *start*, w the *width* of $I^{\langle\sigma,w\rangle}$.

Since the frame axiom (4.1) is first-order, it can be expressed with a finite set of *frame rules* $\mathcal{F}(P)$:

For every n-ary relation R of a given program P, $\mathcal{F}(P)$ contains the rule

$$[{+}{-}\lambda]\ R(X_1,\dots,X_n) \leftarrow \\ [\lambda]\ R(X_1,\dots,X_n), [{-}\lambda]\ \neg\ R(X_1,\dots,X_n).$$

Definition 4.8 (Models)
A Σ-interpretation I^Σ is a *model of a Σ-grounded rule r*, i.e.

$$I^\Sigma \models \quad [\sigma_0]\ A_0 \leftarrow [\sigma_1]\ B_1, \dots, [\sigma_n]\ B_n$$

iff

$$I^{\sigma_1} \models B_1, \dots, I^{\sigma_n} \models B_n \text{ implies } I^{\sigma_0} \models A_0.$$

In other words, if for all i, B_i holds at σ_i then A_0 must be true at σ_0.

Given a rule r, we define $I^\Sigma \models r$ iff $I^\Sigma \models r'$ for all $r' \in [\![r]\!]$.

I^Σ is called a *model of a program P* iff $I^\Sigma \models r$ for all rules $r \in P \cup \mathcal{F}(P)$.

It is possible to stay within the standard framework of deductive databases by *reifying* states, i.e., states are placed in one distinguished argument of predicate symbols (e.g. the first). This is desirable because it enables us to apply any semantics for logic programs to *Statelog*$^{{+}{-}}$ programs. For instance, a fixpoint semantics can be defined using the standard immediate consequence operator T_P [15] instead of defining a modified operator which incorporates states [13].

Definition 4.9 (State-Reification $P^\star$)
With a *Statelog*$^{{+}{-}}$ program P we associate a corresponding set of rules $P^\star$, the *state-reification* of P:

$$P^\star = \{r^\star \mid r \in [\![P \cup \mathcal{F}(P)]\!]\}.$$

Given a rule $r \in [\![P \cup \mathcal{F}(P)]\!]$ of the form

$$[\sigma_0]\ A_0(\bar{X}_0) \leftarrow \\ [\sigma_1]\ B_1(\bar{X}_1), \dots, [\sigma_m]\ B_m(\bar{X}_m)$$

the *reified rule* $r^\star$ is

$$A_0(\sigma_0, \bar{X}_0) \leftarrow \\ B_1(\sigma_1, \bar{X}_1), \dots, B_m(\sigma_m, \bar{X}_m).$$

Here $\bar{X}_i$ is shorthand for a sequence of terms $t_{i_1}, \dots, t_{i_k}$.

A program P and its reified version $P^\star$ are directly related:

Proposition 4.10
Every model $M^\star$ of $P^\star$ defines a model M^Σ of P and vice versa in the following way:

$$M^\sigma \models R(x_1, \dots, x_n)$$

iff

$$M^\star \models R(\sigma, x_1, \dots, x_n)$$

for all $\sigma \in \Sigma$, $R(x_1,\dots,x_n) \in \mathcal{B}_P$.

Obviously, since $M^\star$ is a model of the reified frame rules, M^Σ satisfies (4.1) and thus is a Σ-interpretation.

Strictly speaking, we have to exclude models $M^\star$ that make true some $R(\sigma, \dots, \sigma_0, \dots)$, where $\sigma_0 \in \Sigma$, since this would result in defining $M^\sigma \models R(\dots, \sigma_0, \dots)$ for some $R(\dots, \sigma_0, \dots) \notin \mathcal{B}_P$. It is obvious, how this can be accomplished by using two-sorted logic. Then $\mathcal{B}_{P^\star}$ does not contain "ill-typed" atoms like $R(\dots, \sigma_0, \dots)$.

Due to proposition 4.10 we may use M^Σ and $M^\star$ synonymously in the sequel.

5 Perfect Model Semantics

The perfect model semantics is generally accepted to be the "right" respectively "intended" semantics for the class of locally stratified programs [18]. Since it restricts the use of negation in a certain way (no recursion through negation is allowed), the meaning of a (locally) stratified program can be easily grasped by a programmer.

Even more important, it is amenable to efficient implementation provided a stratification can be easily computed.

In the presence of state-terms, the usual notion of stratification is not sufficient and a slightly generalized version called Σ-*stratification* is necessary. To see why, consider the following example:

$$\begin{array}{lcl} [\texttt{+-}]\ \mathsf{p} & \leftarrow & [\epsilon]\ \neg\ \mathsf{q}. \\ [\epsilon]\ \mathsf{q} & \leftarrow & [\epsilon]\ \mathsf{p}. \end{array}$$

The standard notion of stratification only incorporates the names of relations in order to determine dependencies between rules. Hence, this program would be rejected as not stratifiable, since p seems to depend negatively on itself (the dependency graph is $p \overset{\neg}{\rightarrow} q \rightarrow p$). However, it is clear that in an actual computation, the truth of p will never depend negatively on itself since the two occurrences of p refer to different states $[\epsilon]$ and $[\texttt{+-}]$.

In the following, we formally define the notion of Σ-stratification. For every Σ-stratified program there exists a unique perfect model M_{perf}. The computation of M_{perf} is outlined in section 6.

Definition 5.1 (Σ-Stratification)
Let P be a *Statelog*$^{\texttt{+-}}$ program. The *dependency graph* $\mathcal{D}_P$ is a directed graph whose nodes correspond to the rules of $P \cup \mathcal{F}(P)$. Given two rules $r, r' \in P$

$$\begin{array}{rll} r: & \tau_0 A_0 & \leftarrow \tau_1 B_1, \ldots, \tau_n B_n \\ r': & \tau'_0 A'_0 & \leftarrow \tau'_1 B'_1, \ldots, \tau'_m B'_m \end{array}$$

there is a *positive arc* from r to r', denoted $r' \rightarrow r$ iff there are Σ-instances $r^* \in [\![r]\!], r'^* \in [\![r']\!]$ of the form

$$\begin{array}{rll} r^*: & \sigma_0 A_0 & \leftarrow \sigma_1 B_1, \ldots, \sigma_i B_i, \ldots \sigma_n B_n \\ r'^*: & \sigma'_0 A'_0 & \leftarrow \sigma'_1 B'_1, \ldots, \sigma'_m B'_m \end{array}$$

such that $\sigma_i = \sigma'_0$ and $B_i\Theta = A'_0\Theta$ for some substitution Θ.[6]

There is a *negative* arc $r' \overset{\neg}{\rightarrow} r$, if B_i is a negated atom, i.e. of the form $\neg A_i$ and $A_i\Theta = A'_0\Theta$.

P is called Σ-*stratified* iff $\mathcal{D}_P$ contains no cycles with negative arcs.

[6]We assume that rules are variable disjoint.

Note, that the dependency graph $\mathcal{D}_P$ contains non-ground rules. This is important from a practical point of view, since a premature instantiation of program rules is avoided.

It is easy to verify that $P^\star$ is locally stratified, if P is Σ-stratified. The converse is not true, however: the program with the single rule

$$[\texttt{+-}\lambda]\ \mathsf{p} \leftarrow [\lambda]\ \neg\ \mathsf{p}.$$

is not Σ-stratified since the rule depends negatively on itself, but

$$P^\star = \quad \{ \quad \begin{array}{l} \mathsf{p}(\texttt{+-}) \leftarrow \neg\ \mathsf{p}(\epsilon), \\ \mathsf{p}(\texttt{+-+-}) \leftarrow \neg\ \mathsf{p}(\texttt{+-}), \\ \quad\vdots \end{array} \quad \}$$

is locally stratified.

It is well-known that a locally stratified program has a unique perfect model [18], which can be computed according to a given stratification. Although it is in general undecidable whether a logic program (with function symbols) is locally stratified [6], the more restrictive notion of Σ-stratification is decidable and can be efficiently computed. Summarizing, we have the following result.

Theorem 5.2
Every Σ-stratified Statelog$^{\texttt{+-}}$ program P has a unique perfect model M_{perf}.

It will be shown in section 6 that for *Statelog*$^{\texttt{+-}}$ basically an *iterated fixpoint computation* [3] is sufficient to compute M_{perf}.

Recall that in context of database updates, rules are interpreted progressively. This fact can be exploited when computing the dependency graph $\mathcal{D}_P$ of a *Statelog*$^{\texttt{+-}}_{\pm}$ program, as we will show in the following.

Proposition 5.3
A Statelog$^{\texttt{+-}}_{\pm}$ program P is Σ-stratified iff the subset of rules of P which are progressive, but not strictly *progressive is Σ-stratified.*

Proof For the non-trivial direction, assume that P is not Σ-stratified. Then $\mathcal{D}_P$ contains a cycle with a negative arc. Let $r_1^* \rightarrow \ldots \rightarrow r_n^* \rightarrow r_1^*$ be this negative cycle and $r_i^* \rightarrow r_j^*$ an arbitrary arc thereof. Let σ_i^H, σ_j^B be the states occurring in the head respectively body of r_i^* and r_j^* that belong to the arc $r_i^* \rightarrow r_j^*$ (cf. definition 5.1). We have $\sigma_i^H = \sigma_j^B$ and, since all rules in the negative cycle are progressive, the states occurring in it satisfy the following inequalities:

$$\sigma_1^H = \sigma_2^B \leq \sigma_2^H = \sigma_3^B \leq \ldots \leq \sigma_n^H = \sigma_1^B \leq \sigma_1^H$$

Therefore σ_1^H occurs in the head of *all* rules r_i^*. This implies that *no* rule of the cycle is strictly progressive. Hence, the cycle is also present in the subset P' of rules of P which are not strictly progressive. It follows that P' is not Σ-stratified.

■

An important consequence of proposition 5.3 is that strictly progressive rules of P need not be considered when computing $\mathcal{D}_P$, since cycles in $\mathcal{D}_P$ can only occur *within* but *never across* states. Especially, the dependency graph can be computed without considering the frame rules $\mathcal{F}(P)$.

6 Computing M_{perf} for $Statelog^{+-}_{\rightharpoondown}$

In the following we outline how M_{perf} can actually be computed. Most of the steps can be performed using standard evaluation techniques. The computation of $\mathcal{D}_P$ needs some refinement, however, in order to avoid "pseudo-dependencies" between rules.

6.1 Rule Normalization

In definition 5.1 dependencies are defined using the (possibly infinite) extensions of rules. On the other hand, it is desirable to determine dependencies only from "looking" at the rules directly, i.e. by using unification. In the case of $Statelog^{+-}_{\rightleftharpoons}$ this could indeed be done. For $Statelog^{+-}_{\rightharpoondown}$ however, one has to take care not to introduce more dependencies than necessary.

Consider the following rules:

$$\begin{array}{rrcl}
r_1: & [\texttt{+++}]\ \mathsf{p} & \leftarrow & [\texttt{-}\lambda]\ \neg\ \mathsf{q}. \\
r_2: & [\texttt{----}]\ \mathsf{q} & \leftarrow & [\epsilon]\ \mathsf{b}. \\
\\
s_1: & [\texttt{+}\lambda_0]\ \mathsf{m} & \leftarrow & [\texttt{-}\lambda_0]\ \mathsf{n},\ [\texttt{---}\lambda_1]\ \mathsf{c}. \\
s_2: & [\texttt{--}]\ \mathsf{n} & \leftarrow & [\epsilon]\ \mathsf{b}.
\end{array}$$

Apparently, there are dependencies $r_2 \xrightarrow{\neg} r_1$ and $s_2 \rightarrow s_1$ between these rules. However, looking at the progressive extensions of r_1 and s_1, it is easy see that the state in the head of r_2 is "too large" to match the body of r_1; similar for s_2 which is "too small".

The solution to this problem is to use *normal forms* for progressive rules in order to exclude these "pseudo-dependencies". E.g. the normal form of s_1 is

$$[\texttt{+++}\lambda_0]\ \mathsf{m} \leftarrow [\texttt{---}\lambda_0]\ \mathsf{n}, [\texttt{---}\lambda_1]\ \mathsf{c}.$$

Given this normal form, it is obvious that s_1 does *not* depend on s_2 since the corresponding terms $[\texttt{---}\lambda_0]\ n$ and $[\texttt{--}]\ n$ no longer "unify", similar for r_1 and r_2.

Definition 6.1 (Normal Form of Progressive Rules)
Given a progressive $Statelog^{+-}_{\rightharpoondown}$ rule r of the form

$$\tau_0 A_0 \leftarrow \tau_1 B_1, \ldots, \tau_n B_n$$

we define the *normal form* of r as follows:

If $\tau_0 = [\sigma_0]$, $\sigma_0 \in \Sigma$ then the normal form of r is the finite set of progressive Σ-instances of r, i.e. $[\![r]\!]_{\rightharpoondown}$.

Otherwise let λ_0 be the state-variable that occurs in τ_0. For $i = 0, \ldots, n$ we define

$$\begin{array}{rcl}
\sigma_i & = & \min [\![\tau_i]\!] \\
\Delta & = & \left\lceil \dfrac{\max \sigma_i - \sigma_0}{2} \right\rceil \cdot 2 \\
\pi_i & = & \begin{cases} \sigma_i + \Delta & \text{if } \lambda_0 \text{ occurs in } \tau_i, \\ \sigma_i & \text{otherwise.} \end{cases}
\end{array}$$

It is straightforward to verify that first, $\pi_0 \geq \pi_i$ and second, every progressive rule in $[\![r]\!]_{\rightharpoondown}$ is of the form

$$\begin{array}{l}
[\pi_0 + 2k_0]\ A_0 \leftarrow \\
\qquad [\pi_1 + 2k_1]\ B_1, \ldots, [\pi_n + 2k_m]\ B_n
\end{array} \tag{6.2}$$

where $(k_0, \ldots, k_m) \in \mathcal{I}_0 \times \ldots \times \mathcal{I}_m$ such that $\pi_0 + 2k_0 \geq \pi_i + 2k_i$. Here $\mathcal{I}_i = \mathcal{N}_0$ if τ_i contains a state-variable, $\mathcal{I}_i = \{0\}$ otherwise.

The π_i together with the occurrences of state-variables in r define the *normal form* of r. A more conventional way is to write the normal form as a rule (note, that some λ_i may coincide)

$$\begin{array}{l}
[\pi_0 \lambda_0]\ A_0 \leftarrow \\
\qquad [\pi_1 \lambda_1]\ B_1, \ldots, [\pi_n \lambda_m]\ B_n.
\end{array}$$

Provided that normal forms of rules are used, the dependency graph of a $Statelog^{+-}$ program P can be computed in a standard way: a rule

r depends on a rule r' iff the head of r' "unifies" with some subgoal in the body of r. It is straightforward to extend unification to include state-terms: `+` and `-` have to be interpreted as unary function symbols and the shorthand notation for state-terms has to be expanded. Thus, e.g. [`++`λ] becomes `+(-(+(`λ`)))` with the additional proviso that λ may only be bound to delete states.

6.2 Iterated Computation of M_{perf}

Given $\mathcal{D}_P$, the computation of M_{perf} can be accomplished by a modification of the well-known *iterated fixpoint computation* [3]. In what follows, we confine ourselves to the basics of this computation.

First, the subset P' of rules of P which are *not* strictly-progressive (cf. proposition 5.3) is partitioned into *strata* $S_1, \ldots, S_k$. This stratification can be computed using standard algorithms (see e.g. [22, 12]) in time linear in the size of the IDB. Note, that the perfect model of a program is independent from the chosen stratification.

After this preprocessing step, the actual computation of M_{perf} proceeds by successively computing the snapshots for ϵ, `-`, `+-`, ... until a termination condition is satisfied (see below).

For a given state σ, the computation of the snapshot M^{σ}_{perf} is accomplished as follows:

In the first step, the strictly progressive rules P' (including frame rules) are iterated until a fixpoint is reached (note, that each snapshot is finite). P' corresponds to the lowest stratum S_0 w.r.t. the current state σ, since the rules in P' solely depend on already computed predecessor states of σ.

The second step consists in iterating the remaining (not strictly progressive) rules according to the stratification $S_1, \ldots, S_k$, until the complete snapshot M^{σ}_{perf} is computed.

Termination The question arises how termination of the above described iteration can be guaranteed in the presence of an infinite number of states. It turns out, that although M_{perf} is infinite, it can be finitely represented due to its periodic structure:

In appendix A we show that M_{perf} is *ultimately periodic*, i.e., $M^{\sigma+T}_{perf} = M^{\sigma}_{perf}$ for all $\sigma \geq \sigma_0$ and some σ_0 and T.

Therefore, the finite initial window $M^{\langle\epsilon,\sigma_0\rangle}_{perf}$ together with the periodic window $M^{\langle\sigma_0,T\rangle}_{perf}$ completely characterize M_{perf}. The period T can be determined as follows:

For every *Statelog*$^{\underline{+}-}$ rule r of a given program P, the width w_r of the window of predecessor states on which the head of r depends is determined. Call w_r the *range* of r. Provided r is in normal form (def. 6.1), the range w_r can be calculated by a simple syntactical operation. Then, the maximal range w of rules of P can be computed.

During the computation of M_{perf} one has to check for the earliest repetition of a window $M^{\langle\sigma,w\rangle}_{perf}$ of width w. As soon as a repetition is encountered, the first occurrence of the period $M^{\langle\sigma_0,T\rangle}_{perf}$ is declared to be the *generalized final state* of the database. The length T of this period is simply the distance between the repeated windows $M^{\langle\sigma,w\rangle}_{perf}$. The situation can be depicted as follows:

$$\cdots \circ \underbrace{\underbrace{\overset{M^{\sigma}}{\circ} \cdots \overset{M^{\sigma+w-1}}{\circ}}_{M^{\langle\sigma,w\rangle}} \circ\ \circ \cdots \circ\ \circ}_{T} \underbrace{\overset{M^{\sigma}}{\circ} \cdots \overset{M^{\sigma+w-1}}{\circ}}_{M^{\langle\sigma,w\rangle}} \cdots$$

Note further, that it is not necessary to store the complete history of states in order to answer queries on the (generalized) final state of the database. Instead, it is sufficient to remember $M^{\langle\sigma,w\rangle}_{perf}$.

7 Conclusion and Future Work

In this paper we have presented an extension of *Datalog* called *Statelog*$^{+-}$ which allows to specify updates in a clean and declarative manner. Declarativeness is accomplished by directly incorporating *states* into the language. Through the introduction of two kinds of states, *insert states* and *delete states*, conflicting update requests and inconsistencies are avoided.

We have shown that a clear, model-based semantics can be given to updates without sacrificing efficiency. It is planned to implement *Statelog*$^{\underline{+}-}$ on top of *CORAL* [19]. First results using *XSB-Prolog* [21] as an implementation language confirm the principle feasibility of the approach.

Theoretical research is focusing on the connections to the related field of temporal deductive databases and the applicability of our approach in that area.

Acknowledgments We would like to thank Jürgen Frohn, Paul Th. Kandzia, Heinz Uphoff and Martin Vorbeck for fruitful discussions.

References

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

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

[3] 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.

[4] 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.

[5] A. J. Bonner and M. Kifer. Transaction logic programming. Technical Report CSRI-270, Computer Systems Research Institute, University of Toronto, 1993.

[6] P. Cholak. Post correspondence problem and prolog programs. Technical report, Dep. of Mathematics Wisconsin, 1989.

[7] J. Chomicki and T. Imieliński. Temporal deductive databases and infinite objects. In *Proc. of the ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, 1988.

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

[9] C. de Maindreville and E. Simon. Modelling a production rule language for deductive databases. In *Proc. of the Intl. Conf. on Very Large Data Bases, Los Angeles*, 1988.

[10] M. Gelfond and V. Lifschitz. The stable model semantics for logic programming. In R. Kowalski and K. Bowen, editors, *Intl. Conference on Logic Programming*, pages 1070–1080, 1988.

[11] G. Grahne, A. Mendelzon, and P. Revesz. Knowledgebase transformations. In *Proc. of the ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, 1992.

[12] S. Greco, M. Romeo, and D. Saccà. Evaluation for negative logic programs. In P. Atzeni, editor, *LOGIDATA+ : Deductive Databases with Complex Objects*, number 701 in LNCS. Springer-Verlag, 1993.

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

[14] D. Laurent, V. Phan Luong, and N. Spyratos. Updating intensional predicates in deductive databases. In *IEEE Intl. Conference on Data Engineering*, 1993.

[15] J. W. Lloyd. *Foundations of Logic Programming*. Springer Verlag, second edition, 1987.

[16] S. Manchanda and D. Warren. A logic-based language for database updates. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 363–394. Morgan-Kaufmann, Los Altos, CA, 1988.

[17] S. Naqvi and R. Krishnamurthy. Database updates in logic programming. In *Proc. of the ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 251–262, 1988.

[18] T. Przymusinski. On the Declarative Semantics of Deductive Databases and Logic Programs. In J.Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 193–216. Morgan Kaufmann Publishers Inc., 1988.

[19] R. Ramakrishnan, D. Srivastava, and S. Sudarshan. CORAL: Control, relations and logic. In *Intl. Conference on Very Large Data Bases*, 1992.

[20] D. Saccà, B. Verdonk, and D. Vermeir. Evolution of knowledge bases. In *Intl. Conference on Extending Database Technology*, LNCS. Springer, 1992.

[21] K. Sagonas, T. Swift, and D. S. Warren. The XSB programmer's manual. Technical report, Department of Computer Science, SUNY at Stony Brook, 1993.

[22] J. D. Ullman. *Principles of Database and Knowledge-Base Systems, Volume I.* Computer Science Press, New York, 1988.

[23] A. Van Gelder. Negation as failure using tight derivations for general logic programs. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 149–176. Morgan Kaufmann, 1988.

[24] A. Van Gelder, K. Ross, and J. Schlipf. The well-founded sematics for general logic programs. *JACM*, 38(3):620 – 650, 7 1991.

[25] M. Winslett. *Updating Logical Databases.* Cambridge University Press, 1990.

A Periodicity of M_{perf}

In section 5 we showed that every Σ-stratified *Statelog*$^{+-}$ program has a unique perfect model M_{perf}. In our framework for database updates, M_{perf} is the intended model of a Σ-stratified *Statelog*$^{+-}_{\_}$ program.

The feasibility of using M_{perf} in a practical application essentially depends on whether it is possible to finitely represent query answers w.r.t. M_{perf}.

Fortunately, since M_{perf} has a *periodic* structure this is indeed the case, as we will show in the following.

Generally, in the presence of negation, periodicity of models can not always be guaranteed. The intuitive reason is that we have two sources of complexity in *Statelog*$^{+-}$, one is the *infinite number of states* which results in infinite models, the other is *negation*. In [7, 8] it has been shown that the *minimal model* of a *Datalog*$_{1S}$ program without negation is ultimately periodic. However, their result can not be directly used in our context. Especially, the progressive reading of *Statelog*$^{+-}_{\_}$ rules can be seen as an additional *constraint* imposed on state-terms which can not be directly expressed in *Datalog*$_{1S}$. Furthermore, negation is needed in our context and the distinction between insert and delete states is crucial.

In the following definitions we identify Σ and $\mathcal{N}_0$.

Definition A.1
A Σ-interpretation M^Σ is called *ultimately periodic* if there exist $\sigma_0, T \in \mathcal{N}_0$ such that

$$M^{\sigma+T} = M^\sigma \text{ for all } \sigma \geq \sigma_0.$$

Definition A.2
A Σ-interpretation M^Σ is called *progressively deterministic* if there exist $\sigma_0, w \in \mathcal{N}_0$ such that for all $\sigma \geq \sigma_0$ the snapshot M^σ functionally depends on its w predecessor states, i.e.

$$M^\sigma = f(M^{\langle \sigma - w, w\rangle}) \text{ for all } \sigma \geq \sigma_0$$

and some function $f : (2^{\mathcal{B}_P})^w \rightarrow 2^{\mathcal{B}_P}$.

In what follows, we generally assume that a *Statelog*$^{+-}$ program P is function free, so the Herbrand universe $\mathcal{U}_P$ and the Herbrand base $\mathcal{B}_P$ are finite.[7]

It is easy to see that the following proposition holds.

[7]If we want to use function symbols however, e.g. to express arithmetical operations, we may relax this restriction and admit programs with the *bounded term size property* [23]. Then for every query Q with maximal term size k, it is sufficient to consider derivations where the size of terms is bound by k. Consequently, if we include Q in the program P, a finite subset of the Herbrand universe is sufficient to answer Q. Since it is in general undecidable whether a given program has the bounded term size property one has to define some decidable criterion which approximates the bounded term size property. An example of this are syntactical safeness conditions like in [22].

Lemma A.1 *Every progressively deterministic Σ-interpretation M^Σ is ultimately periodic.*

Proof Since $\mathcal{B}_P$ is finite there are at most $2^{w \cdot |\mathcal{B}_P|}$ many different windows of width w. Hence there must be a window repetition in M^Σ, i.e. there exist states $\psi_1 > \psi_0 \geq \sigma_0$ such that

$$M^{\langle \psi_0, w\rangle} = M^{\langle \psi_1, w\rangle}.$$

As every state $\sigma \geq \sigma_0$ is determined by its w predecessors, we have $M^{\psi_0+i} = M^{\psi_1+i}$ for all $i \in \mathcal{N}_0$. So let $T = \psi_1 - \psi_0$, and the lemma follows. ■

Theorem A.3
The perfect model M_{perf} of a Σ-stratified Statelog$^{+-}_{\_}$ program is ultimately periodic.

Proof Let P^* be the program which results from substituting data-variables in P by terms from $\mathcal{U}_P$ in all possible ways. Since $\mathcal{B}_P$ is finite we may assume without loss of generality that the rules of P^* are built from finitely many propositional atoms $p_1, \ldots, p_k$. Similar to definition 4.9, we can reify the rules of P^* such that state-terms τ_i are considered as the arguments of unary predicates $p_1(\tau_1), \ldots, p_k(\tau_l)$.

Without loss of generality, we further assume that rules are in normal form and do not contain Σ-grounded state-terms. Then the general form of a rule r (including frame rules) is

$$\begin{array}{lll} p_r(\pi_0\lambda_0) \leftarrow & & \\ & L_1(\pi_1\lambda_0), \quad \ldots \quad, L_m(\pi_m\lambda_0), & (A) \\ & L'_1(\pi'_1\lambda_1), & \\ & \vdots & (B) \\ & L'_n(\pi'_n\lambda_n). & \end{array}$$

All literals L_i which contain the same state-variable λ_0 as the head are grouped together in (A). π_i, π'_i are prefixes from Π such that $\pi_0 \geq \pi_i, \pi'_j$ (since r is progressive and in normal form).

Let M_{perf} be the unique perfect model of P^*. We show that M_{perf} is progressively deterministic, from which the desired result follows.

Consider an arbitrary Σ-instance r' of r. Let $p_r(\sigma')$ be the head of r', i.e. $\sigma' = \pi_0\lambda_0\Theta$ for some Σ-grounding substitution Θ. Whether $p_r(\sigma')$ can be derived by r depends on the subgoals in (A) and (B). In (A), only a *fixed* number of predecessor states of σ' and possibly σ' itself are referenced, while the validity of (B) apparently depends on arbitrarily many predecessors of σ'. However, P^* can be rewritten into P' such that $M_{perf}(P^*) = M_{perf}(P')$, and all rules in P' have (B) vacuous. This is accomplished as follows.

Let (B') be a Σ-instance of (B) such that $M_{perf} \models (B')$ (for definiteness we may choose the smallest Σ-instance). If no such (B') exists, i.e. $M_{perf} \not\models (B)$ then r can be deleted. Clearly, the resulting program will have the same perfect model M_{perf} (recall that all true facts in M_{perf} are supported by some rule). Otherwise, we show that (B) can be discarded since it is true for all states $\sigma \geq \sigma_0$ for some σ_0.

Let σ_r be the maximal state occurring in (B'). Then for all states $\sigma \in [\![\pi_0\lambda_0]\!]$, $\sigma \geq \sigma_r$, the head $p_r(\sigma)$ is derived by r iff (A) becomes true.

(B) can be ignored since the state-variable λ_0 may be instantiated independently from the state-variables $\lambda_1, \ldots, \lambda_n$ of (B). Thus, as soon as (B) becomes true at some state σ_r, it remains true for all subsequent states.

Let $\sigma_0 = \max_{r \in P^*}(\sigma_r)$. For all $\sigma \geq \sigma_0$ the derivation of $p_r(\sigma)$ depends on at most w predecessor states of σ and possibly σ itself (w is the maximal *range* of rules of P'; cf. section 6.2).

Finally, if a window $M_{perf}^{\langle \sigma - w, w \rangle}$ of w predecessor states of σ is given (note, that σ does not belong to the window), there is one and only one snapshot M_{perf}^{σ} which is supported by the rules of the program. Hence M_{perf} is progressively deterministic.

■