Universität Freiburg
Institut für Informatik

Bericht 59, September 1994

Updates by Reasoning about States

Georg Lausen
Bertram Ludäscher

Updates by Reasoning about States*

Georg Lausen Bertram Ludäscher
Institut für Informatik, Albert-Ludwigs-Universität
D-79104 Freiburg, Germany
{lausen,ludaesch}@informatik.uni-freiburg.de

Abstract

We present a language extension to Datalog which allows to specify deterministic update procedures in a declarative way. Existing update languages either rely on procedural aspects of program evaluation or leave the framework of deductive databases and logic programming. In contrast, we show that declarative update semantics can be attained by incorporating state terms into the language. Since the resulting language corresponds to a certain class of logic programs, well-known semantics and evaluation techniques can be applied to it.

Computing Reviews Classification: H.2.3 [**Database Management**]: Languages — *data manipulation languages, programming languages, query languages*

Additional Keywords: updates, deductive databases

1 Introduction

While Datalog [19], the most prominent deductive database language, has proven to be very powerful as a *query language*, it lacks means to specify database *dynamics*. The model of a Datalog program statically describes one state of the modelled world. However, updating the contents of a database — like querying — is one of the most fundamental operations that a database system must provide. Consequently, numerous extensions to Datalog have been proposed to integrate the required update concepts.

Clearly, in order to express updates, the expressiveness of Datalog has to be increased. In [1] it is argued that *update = logic + control*, i.e. additional expressiveness is attained by additional control. However, the proposed extensions rely on procedural aspects of program evaluation and incorporate updates into a (bottom-up) evaluation procedure [1, 2]. Consequently, the resulting semantics can only be understood procedurally and a major advantage of logic-based languages, viz their declarative semantics is lost.

In contrast, we propose to increase expressiveness by incorporating *state terms* into the language. This allows to specify complex and deterministic update "procedures" by referring to different database states, yet a declarative semantics is retained. Since the proposed language *Statelog* stays within the well-known framework of logic programming, a plethora of semantics and evaluation techniques can be applied.

*A slightly shorter version of this paper was presented at the 2nd International East/West Database Workshop, Klagenfurt, Austria, September 1994

We continue work started in [12] where a fixpoint semantics was defined for an update language which could only process a constant number of intermediate states. In contrast, by the use of state variables, Statelog allows to refer to arbitrary database states, which results in greater expressiveness. It turned out that Statelog is closely related to the temporal deductive database language $Datalog_{1S}$ [9]. In fact, we use the same notion of periodicity to define the final state of an update (cf. section 5.1).

The structure of the paper is as follows. In section 2 we introduce the basic ideas of our approach. We continue with a discussion of various examples in section 3. Section 4 presents the syntax and a general framework for the semantics of Statelog. As an important example, we consider the perfect model semantics and its computation in section 5. In section 6 we summarize our approach and compare it to related work.

2 Basic Ideas

Statelog is based on the following two observations. First, the evolution of a database is given by a sequence of database states. Thus, in order to describe the transition from one state to another declaratively, the *language* must allow to refer to different states. Second, in order to describe deletions declaratively, facts which are implied by a logic database may not be removed at will. Instead, some information (a delete request) has to be added which prevents the derivation of the to be deleted facts. This requires a nonmonotonic formalism like the use of negation in logic programming. Statelog allows for exactly these requirements by incorporating *state terms* into the language and by the use of a *frame rule* which handles delete requests.

More precisely, every state in the database evolution is identified by a natural number, where 0 refers to the initial state and $S+1$ to the immediate successor state of S. The transition of a database from state S to $S+1$ is then given by the following simple frame axiom:

$$\mathit{True}(S+1) \quad = \quad \mathit{True}(S) \backslash \mathit{Del}(S+1) \;\cup\; \mathit{Ins}(S+1) \tag{1}$$

The first part $\mathit{True}(S)\backslash \mathit{Del}(S+1)$ indicates that atoms which were true in the predecessor state S and which are not "marked for deletion" in $S+1$ continue to be true in $S+1$. Additionally, the newly inserted atoms $\mathit{Ins}(S+1)$ are true in $S+1$.

$\mathit{True}(S)$ constitutes the actual logical model of state S. All atoms which are not in $\mathit{True}(S)$ are considered false. In order to represent (1) within Statelog, it is sufficient to partition every state S into the set $\mathit{True}(S)$ of true atoms and the set $\mathit{Del}(S)$ of atoms which are to be deleted. $\mathit{Ins}(S)$ is then given implicitly by the corresponding update program.

Intuitively, for an atom R, the expression $[S]{+}R$ corresponds to the *goal* $R \in \mathit{True}(S)$, if it occurs in the body of a Statelog rule, or to the *assertion* $R \in \mathit{True}(S)$, if it occurs in the head. In the latter case, the rule specifies an insert operation. For brevity we usually write $[S]R$ instead of $[S]{+}R$. Similarly, the expression $[S]{-}R$ corresponds to $R \in \mathit{Del}(S)$. In particular, if $[S]{-}R$ occurs in the head of a rule, the rule specifies a delete operation.

Given an n-ary relation R, the part of (1) which preserves the "old information" $\mathit{True}(S)\backslash \mathit{Del}(S+1)$ can be expressed in Statelog by the *frame rule* for R:

$$[S+1]\ R(\bar{X}) \leftarrow [S]\ R(\bar{X}), \neg[S+1]{-}R(\bar{X}). \tag{2.1}$$

Here, S is a distinguished *state variable* which may only be bound to a state identifier (a natural number), while $\bar{X}$ is a vector $X_1, \ldots, X_n$ of ordinary (data) variables.

A Statelog database consists of an extensional database EDB and an intensional database IDB which are defined as usual and describe the initial state 0 of the database. A corresponding frame rule (2.1) is used for all EDB relations and may be applied to derived relations as well (cf. section 3).

Additionally, there is a *transition base* TB which contains *update rules* describing the possible transitions of the database from a current state s via intermediate (virtual) states $s+1, s+2, \ldots$, to a new final state s'. Usually update rules are blocked by certain goals, so-called *events* which act as *guards*.

An event is conceived as a built-in predicate whose validity in a state s is asserted externally. The set of occurred events is recorded as a finite set of ground facts, the *event base* EB. To distinguish symbols denoting events from "ordinary" predicate symbols, we stipulate to prefix them with a special character $\triangledown$ (pronounced "occurs"). Whenever an event $\triangledown E$ occurs in a state s, a ground fact $[s]\triangledown E$ is added to EB. This may trigger update rules which generate new database states through the insertion or deletion of EDB or IDB atoms until the final state s' is reached. s' is defined using the periodic structure of the model of the database (section 5.1).

3 Statelog by Example

In the following, we present some simple examples which demonstrate how updates may be specified in Statelog.

EDB Updates Consider the proverbial (and oversimplified) "employee-salary" example. Assume the EDB contains facts es(E,Sal) describing the initial employee-salary relation in state 0. We want to allow the user to issue a request raise(E,F) with the desired effect of increasing E's salary by a factor F. We can use the following update rule[1]

$$\begin{aligned} &[S+1]\ \mathsf{es(E,Sal1)},\ [S+1]{-}\mathsf{es(E,Sal)}\ \leftarrow \\ &\qquad [S]\ \triangledown\mathsf{raise(E,F)},\ [S]\ \mathsf{es(E,Sal)},\ \ \mathsf{Sal1} = \mathsf{Sal}\cdot\mathsf{F}. \end{aligned} \tag{3.1}$$

Assume the current database state is s, and es(john,1000) is true in the current state. In order to raise john's salary by 7%, we have to assert that $\triangledown$raise(john,1.07) is true in s. This can be given explicitly by the user, e.g. in a query shell: `[s]> raise(john,1.07).`

Such a shell allows to assert events for the current state and to query a window of predecessor states. The occurrence of the event $\triangledown$raise(john,1.07) triggers rule (3.1), and a new database state $s+1$ is created: the first atom in the head causes the insertion of es(john,1070), while the second results in the deletion of es(john,1000). The frame rule for es

$$[S+1]\ \mathsf{es(E,Sal)} \leftarrow [S]\ \mathsf{es(E,Sal)},\ \neg\ [S+1]{-}\mathsf{es(E,Sal)}. \tag{3.2}$$

guarantees that the other employees' salaries remain the same in the new state.

[1] Rules with multiple heads like (3.1) are an abbreviation for several rules with the same body, thus the head denotes a conjunction.

Integrity Constraints One way to express integrity constraints in deductive databases is in the form of *denials* [6]. Since Statelog allows to refer to different states, not only static but also dynamic integrity constraints may be expressed. The following rule states that salaries may never decrease.

$$[S+1]\ \mathsf{false}\ \leftarrow [S]\ \mathsf{es(E,Sal)},\ [S+1]\ \mathsf{es(E,\ Sal1)},\ \mathsf{Sal1} < \mathsf{Sal} \tag{3.3}$$

Whenever the salary of an employee in state $S+1$ is less than in S, false is derived in $S+1$ which indicates a violation of the integrity constraint.

Another possibility is to use rules to *enforce* integrity constraints, e.g. by undoing the effect of an update: assume we want to reject a salary raise if some employee earns more than 2100. We use the rule

$$[S]\ \mathsf{undo\_raise}\quad \leftarrow\quad [S]\ \mathsf{es(E,Sal)},\ \mathsf{Sal} > 2100. \tag{3.4}$$

In order to enforce this constraint, we add rule (3.5) to the program. (Here parenthesis are used to group literals with the same state term.)

$$\begin{aligned}&[S+2]\ (\mathsf{es(E,Sal)},\ -\mathsf{es(E,NewSal)}) \leftarrow\\ &\qquad [S]\ (\nabla\mathsf{raise(E,F)},\ \mathsf{es(E,Sal)}),\\ &\qquad [S+1]\ (\mathsf{es(E,NewSal)},\ \mathsf{undo\_raise}).\end{aligned} \tag{3.5}$$

If undo_raise is true in state $S+1$ a new final state $S+2$ is created where the effect of the update rule (3.1) is undone. In a sense (3.4) expresses the negation of a *post-condition*, which says that after an update on es all employees' salaries must be less or equal to 2100.

Finally, rule (3.5) also allows a kind of *what-if* reasoning by always undoing a salary raise, or equivalently, by removing the subgoal undo_raise. Then querying $S+1$ yields the (hypothetical) es relation *after* the update on es, while the final state $S+2$ holds the *original* es relation, i.e. *before* the raise.

IDB Updates One possibility to handle updates to IDB relations in Statelog is to apply the frame rule (2.1) to derived relations as well. Consider the following example (cf. [6]): assume we want to represent a relation flight(Flight_No, Day, Departure, Arrival). Since departure and arrival times of flights often coincide for different days, it is advantageous to use rules of the form

$$\mathsf{flight(lh4356,Day,0725,0900)} \leftarrow \mathsf{day(Day)},\ \mathsf{Day} \neq \mathsf{sunday}. \tag{3.6}$$

The flight relation is propagated to subsequent states (modulo insertions and deletions) by the frame rule (2.1). (3.6) states that there is a flight lh4356 between 7:25 and 9:00, which operates daily except for Sundays. In a sense, (3.6) derives a default relation in state 0 for flight, while subsequent changes by update rules act as *exceptions* to (3.6) (this resembles the approach for deterministic IDB updates [13]). The update rule

$$\begin{aligned}&[S+1]\ (\mathsf{flight(F,Day2,D,A)},\ -\mathsf{flight(F,Day1,D,A)}) \leftarrow\\ &\qquad [S]\ (\nabla\mathsf{move(F,Day1,Day2)},\ \mathsf{flight(F,Day1,D,A)}).\end{aligned}$$

when triggered by the event ∇move(lh4356,saturday,sunday) moves the flight lh4356 from Saturday to Sunday. A different way to handle IDB relations is shown in the following paragraph.

Update Procedures The last example is a variant of the *game of life* (cf. [2]) and shows how more complex update procedures can be formulated using Statelog. Assume the EDB contains a symmetric binary relation neighbor(C1,C2) describing which cells are neighbors and a relation alive(C) to record the living cells. The IDB contains rules to determine whether a cell has (at least) 2 respectively 3 neighbors that are alive:

$$\begin{array}{l} \mathsf{has2neighbors(C)} \leftarrow \\ \qquad \mathsf{neighbor(C,N1),\ neighbor(C,N2),\ N1} \neq \mathsf{N2}, \\ \qquad \mathsf{alive(N1),\ alive(N2).} \end{array}$$

$$\begin{array}{l} \mathsf{has3neighbors(C)} \leftarrow \\ \qquad \mathsf{neighbor(C,N1),\ neighbor(C,N2),\ neighbor(C,N3),} \\ \qquad \mathsf{N1} \neq \mathsf{N2},\ \mathsf{N2} \neq \mathsf{N3},\ \mathsf{N1} \neq \mathsf{N3}, \\ \qquad \mathsf{alive(N1),\ alive(N2),\ alive(N3).} \end{array}$$

The relation has2neighbors(C) has to be derived anew in every state S in order to reflect the changes of alive(C). Hence, instead of applying the frame rule to has2neighbors(C), we use the following rule (analogously for has3neighbors(C)):

$$\begin{array}{l} [S]\ \mathsf{has2neighbors(C)} \leftarrow \\ \qquad [S]\ \mathsf{(neighbor(C,N1),\ neighbor(C,N2),\ N1} \neq \mathsf{N2),} \\ \qquad [S]\ \mathsf{(alive(N1),\ alive(N2)).} \end{array} \tag{3.7}$$

The update procedure for alive consists of rules which define when a living cell dies or when a new cell is born. This is accomplished by corresponding delete and insert operations. The event $\triangledown$next_gen is used to guard the activation of rules. Since at any moment only finitely many events may have occurred, the program terminates (cf. section 5.1). A new generation is computed by asserting $\triangledown$next_gen in the final state.

$$\begin{array}{l} \textit{\% less than 2 neighbors: cell dies} \\ [S+1]\mathsf{-alive(C)} \leftarrow [S]\ (\triangledown\mathsf{next\_gen,\ alive(C),}\ \neg\ \mathsf{has2neighbors(C)).} \\ \textit{\% more than 2 neighbors: cell dies} \\ [S+1]\mathsf{-alive(C)} \leftarrow [S]\ (\triangledown\mathsf{next\_gen,\ alive(C),\ has3neighbors(C)).} \\ \textit{\% exactly 2 neighbors: cell is born} \\ [S+1]\ \mathsf{alive(C)} \leftarrow [S]\ (\triangledown\mathsf{next\_gen,\ has2neighbors(C),}\ \neg\ \mathsf{has3neighbors(C)).} \end{array} \tag{3.8}$$

4 A Statelog Framework

In this section we define the syntax and a general framework for the semantics of Statelog. By mapping Statelog programs to a certain class of standard logic programs, declarative semantics like the perfect model semantics [16], the well-founded semantics [21] and the stable semantics [10] can be applied to Statelog.

4.1 Syntax

Definition 4.1 (Signature) The Statelog language $\mathcal{L}$ is built using the following standard items: the set of predicate symbols $\mathcal{P}$, the set of function symbols $\mathcal{F}$ and the set of data variables $\mathcal{V}$. $\mathcal{C} \subseteq \mathcal{F}$ is the set of constants (function symbols with arity 0). We partition $\mathcal{P}$ into two disjoint sets: the set $\mathcal{R}$ of "ordinary" relation symbols and the set $\mathcal{E}$ of *event symbols* (built-in predicates). By convention event symbols start with a special character ∇.

Additionally, we have the following non-standard items: a set $\mathcal{S}$ of *state variables*, where $\mathcal{V} \cap \mathcal{S} = \emptyset$, a distinguished constant $0 \notin \mathcal{C}$ and a distinguished unary function symbol $+1 \notin \mathcal{F}$ (usually written in postfix).

From 0 and +1 we build the set of *state identifiers* $\Sigma = \{0, (0)+1, ((0)+1)+1, \ldots\}$ which may be identified with the set of natural numbers $\mathcal{N}_0$. Thus 2 is the same as $((0)+1)+1$ etc. $\square$

In the context of deductive databases, usually $\mathcal{F} = \mathcal{C}$, but we may also allow a restricted (safe) use of function symbols.

Definition 4.2 (State Terms) A *state term* is of the form $[S+k]$+ or $[S+k]-$, where $S \in \mathcal{S} \cup \{0\}, k \in \mathcal{N}_0$. $S+k$ stands for the k-fold application of the function symbol +1 to S. State terms without state variable are called Σ*-ground*. $\square$

State terms $[S+k]$+ and $[S+k]-$ are used to refer to the two parts $\mathit{True}(S+k)$ and $\mathit{Del}(S+k)$ of a state $S+k$. For notational convenience, we write $[S+k]\pm$ to denote either $[S+k]$+ or $[S+k]-$. $[S+k]$ is a shorthand for $[S+k]$+. The Σ-ground state terms $[0+k]\pm$ are abbreviated to $[k]\pm$.

Definition 4.3 (Statelog) From the set of data variables $\mathcal{V}$ and function symbols $\mathcal{F}$ we build *data terms* in the usual way.

A Statelog *atom* is of the form $\tau\ p(t_1, \ldots, t_n)$ where τ is a state term, p is an n-ary relation or event symbol from $\mathcal{P}$ and t_i are data terms. A *literal* is an atom A or its negation $\neg A$.

A *rule* r is of the form $H \leftarrow B_1, \ldots, B_n$ where H is an atom and each B_i is a literal. If $n = 0$, r is called a *fact*, if the head H is omitted r is a *query*. The rule $H_1, \ldots, H_k \leftarrow B_1, \ldots, B_n$ is an abbreviation for k rules $H_i \leftarrow B_1, \ldots, B_n$, i.e. $H_1, \ldots, H_k$ denotes a conjunction.

If r contains no state variables it is Σ*-ground*, if it contains no variables at all it is called *ground*. A Σ-ground rule r^* which is obtained from a rule r by substituting state variables by state identifiers is called a Σ*-instance* of r.

A Statelog *program* P is a finite set of range restricted[2] rules. P is partitioned as follows: facts and rules where only state terms [0]+ occur, define the initial state of the database and make up the EDB and IDB. Since we allow to omit [0]+, EDB and IDB constitute a Datalog program.

Ground facts, where the predicate symbols are events ∇E build the *event base* EB, all other rules describe database transitions and belong to the *transition base* TB. $\square$

[2] Every data variable occurring in a rule occurs in at least one positive literal in the body.

4.2 Semantics

Statelog models the evolution of a database as a sequence of states. Therefore rules have to be interpreted *progressively*, i.e. it is not possible to change past states or make assumptions about future states.

In the frame axiom (1), $True(S+1)$ depends on $Del(S+1)$. This induces a natural order $Del(0), True(0), \ldots Del(n), True(n), \ldots$ on the sets $True(s)$ and $Del(s)$. Formally, we define an order on the set of Σ-ground state terms by a mapping $ord : \tau \mapsto |\tau|$ into the set of natural numbers

$$|\tau| \quad = \quad \begin{cases} 2s & \text{if } \tau = [s]\text{−}, \ s \in \Sigma \\ 2s+1 & \text{if } \tau = [s]\text{+}, \ s \in \Sigma \end{cases} \tag{2}$$

Given this ordering the meaning of a Statelog rule is defined as its *progressive extension*:

Definition 4.4 (Extensions) A Σ-ground rule $\tau_0\ H \leftarrow \tau_1\ B_1, \ldots, \tau_n\ B_n$ is called *progressive* if $|\tau_0| \geq |\tau_i|$ for $i = 1, \ldots, n$; *strictly progressive* if $|\tau_0| > |\tau_i|$.

The *extension* $[\![r]\!]$ of a rule r is the set of all *progressive* Σ-instances which can be constructed from r. We say that r is *progressive* if $[\![r]\!] \neq \emptyset$, *strictly progressive* if additionally all rules in $[\![r]\!]$ are strictly progressive.

r is called *local* if for every rule $r^* \in [\![r]\!]$, all τ_i are equal, i.e. r^* only refers to one state. The extension $[\![P]\!]$ of a Statelog program P is the union of all extensions of rules of P. □

Example 4.5 Due to the progressive reading of rules, the meaning of the progressive but not strictly progressive rule $[S]$ p $\leftarrow [T]$−q is *"for all $S \geq T$: if* q *was deleted in T then* p *is true (inserted) in S"*.
The rules $[S]$ p $\leftarrow [S+1]$ q and $[S]$−p $\leftarrow [S]$ q are not progressive. The frame rule (2.1) and $[S]$ p $\leftarrow [S]$−q are strictly progressive rules, (3.7) is a local rule.

The semantics of a Statelog program P is defined using Herbrand structures. The Herbrand universe $\mathcal{U}_P$ is the set of all ground data terms which can be constructed from P, the Herbrand base $\mathcal{B}_P$ is the set of all ground atoms $p(t_1, \ldots, t_k)$ where $t_i \in \mathcal{U}_P$ and p is a predicate symbol occurring in P. In the sequel, we assume that $\mathcal{B}_P$ is finite, i.e. P contains no function symbols (apart from state terms).[3]

Definition 4.6 (Σ-Interpretation) A Σ-*interpretation* I^Σ is a mapping from Σ to $2^{\mathcal{B}_P} \times 2^{\mathcal{B}_P}$, i.e. to each state identifier $s \in \Sigma$ two Herbrand interpretations $Del(s), True(s) \subseteq \mathcal{B}_P$ are assigned $I^\Sigma : s \mapsto (Del(s), True(s))$, which hold the to be deleted respectively true atoms of state s.

$I^s = (Del(s), True(s))$ is called the *snapshot* of I^Σ at s. A (possibly infinite) sequence of successive snapshots $I^s, I^{s+1}, \ldots, I^{s+w-1}$ where $w \in \mathcal{N} \cup \{\infty\}$ is called a *window* of I^Σ. It is denoted $I^{\langle s,w \rangle}$, where s is the *start* and w the *width* of $I^{\langle s,w \rangle}$. □

[3] We may relax this restriction and admit programs with the *bounded term size property* [20]. Then for every query Q with maximal term size k, it is sufficient to consider derivations where the size of terms is bound by a function of 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 it [20].

Definition 4.7 (Models) Given a Σ-interpretation I^Σ, $s \in \Sigma$ and $R \in \mathcal{B}_P$ we define $I^\Sigma \models [s]{-}R$ *iff* $R \in Del(s)$ and $I^\Sigma \models [s]{+}R$ *iff* $R \in True(s)$,

For a Σ-ground rule $r^* : \tau_0\ H \leftarrow \tau_1\ B_1, \ldots, \tau_n\ B_n$ we define $I^\Sigma \models r^*$ *iff* $I^\Sigma \models \tau_i\ B_i$ for $i = 1, \ldots, n$ implies $I^\Sigma \models \tau_0\ H$.

$I^\Sigma \models r$ for a Statelog rule r *iff* $I^\Sigma \models r^*$ for all $r^* \in [\![r]\!]$. Finally, I^Σ is a *model* of a program P *iff* $I^\Sigma \models r$ for all $r \in P$. □

Statelog programs can be regarded as a syntactic variant of a class of logic programs augmented by the concepts which are required in our framework for database updates, viz the *progressive reading of rules*, *bipartite states* and rules with *events acting as guards*. By mapping Statelog to "ordinary" logic programs it is possible to apply well-known declarative semantics and evaluation techniques to Statelog.

Definition 4.8 (State Reification $P^\star$) The *state reification* of a Statelog program P is the set of rules $P^\star = \{r^\star \mid r \in [\![P]\!]\}$, where $r^\star$ is the result of substituting atoms $[s] \pm p(t_1, \ldots, t_k)$ in r by $p_\pm(s, t_1, \ldots, t_k)$. □

Note that the language $\mathcal{L}_{\mathcal{P}^\star}$ of the reified program contains two relations p_+ and p_- denoting the true, respectively to be deleted tuples instead of the relation p of the original Statelog program.

The Herbrand universe $\mathcal{U}_{P^\star}$ of the reified program $P^\star$ consists of $\mathcal{U}_P \cup \Sigma$. Since we are only interested in certain Herbrand interpretations, we define as the Herbrand base of the reified program the set of all ground atoms $p_\pm(s, t_1, \ldots, t_k)$ for every $(k+1)$-ary $p_\pm$ of $P^\star, s \in \Sigma, t_i \in \mathcal{U}_P$. The following is a straightforward consequence of the previous definitions and links P and $P^\star$:

Proposition 4.9 *Every model M^Σ of P defines a Herbrand model $M^\star$ of $P^\star$ and vice versa by defining for all $s \in \Sigma$, $p(t_1, \ldots, t_k) \in \mathcal{B}_P$:*

$$\begin{array}{lll} p_-(s, t_1, \ldots, t_k) \in M^\star & \textit{iff} & p(t_1, \ldots, t_k) \in Del(s) \textit{ and} \\ p_+(s, t_1, \ldots, t_k) \in M^\star & \textit{iff} & p(t_1, \ldots, t_k) \in True(s) \end{array}$$

Due to proposition 4.9 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 [16]. 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 which only considers predicate symbols is not sufficient and a slightly generalized version called Σ-*stratification* is necessary. To see why, consider rules (3.1) and (3.2) from section 3. The atom −es(E,Sal) in the head of rule (3.1) depends on the goal es(E,Sal) which in turn depends negatively on −es(E,Sal) due to frame rule (3.2). However, looking at the state terms one easily verifies that this negative cycle is not present in the extension of the program; in fact, the reified program is locally stratified.

This leads to the following observation which is crucial for our definition of Σ-stratification: given a Σ-instance of a progressive rule r, the literals in the body of r refer to the same state s as the head of r or to predecessor states of s. As a consequence, cyclic dependencies between atoms can only occur *within*, but never *across* states. Therefore, strictly progressive rules including frame rules need not be considered when computing the dependency graph of a Statelog program.

Definition 5.1 (Σ-Stratification) Let P be a Statelog program, P' the subset of rules of P which are progressive but *not strictly progressive*. The *dependency graph* $\mathcal{D}_P$ is a directed graph whose nodes correspond to the rules of P'. Given two rules $r, r' \in P'$ with disjoint variables

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

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

$$\begin{array}{rll} r^*: & s_0 A_0 & \leftarrow s_1 B_1, \ldots, \boldsymbol{s_i B_i}, \ldots s_n B_n \\ r'^*: & \boldsymbol{s'_0 A'_0} & \leftarrow s'_1 B'_1, \ldots, s'_m B'_m \end{array}$$

such that $s_i = s'_0$ and $B_i\Theta = A'_0\Theta$ for some substitution Θ. There is a *negative* arc $r' \stackrel{\neg}{\to} r$, 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* if $\mathcal{D}_P$ contains no cycles with negative arcs. □

The next theorem implies that every Σ-stratified program P has a unique perfect model $\mathcal{M}$.

Theorem 5.2 *If a Statelog program P is Σ-stratified then the state-reified program $P^\star$ of P is locally stratified.*

Proof We show: if $P^\star$ is not locally stratified then P is not Σ-stratified.

If $P^\star$ is not locally stratified then the dependency graph built from the rules $r^\star$ of $P^\star$ contains a cyclic dependency with a negative arc. Let $r_1^\star \to \ldots \to r_k^\star \to r_1^\star$ be this negative cycle and $r_i^\star \to r_j^\star$ an arbitrary arc thereof. Given $r_i^\star \to r_j^\star$ let s_i^h be the state identifier occurring in the first argument of the head of $r_i^\star$, and let s_j^b be the state identifier of a literal of the body of $r_j^\star$. We have $s_i^h = s_j^b$ and, since all rules in the negative cycle are progressive, the states occurring in it satisfy the following inequalities:

$$s_1^h = s_2^b \leq s_2^h = s_3^b \leq \ldots \leq s_k^h = s_1^b \leq s_1^h$$

Therefore s_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. ■

If $P^\star$ is locally stratified it has a unique perfect model $\mathcal{M}^\star$ [16] which according to proposition 4.9 corresponds to a model $\mathcal{M}^\Sigma$ (or $\mathcal{M}$ for short) of P. For the rest of the paper we assume that a Σ-stratified Statelog program is given and that $\mathcal{M}$ is its perfect model.

5.1 Termination and Periodicity of $\mathcal{M}$

Although $\mathcal{M}$ consists of infinitely many snapshots it can be finitely represented due to its periodic structure. Note that by assumption (the relevant part of) $\mathcal{B}_P$ is finite. We need a few definitions.

Definition 5.3 A Σ-interpretation M^Σ is called *ultimately periodic* if there exist $s_0, T \in \mathcal{N}_0$ such that $M^{s+T} = M^s$ for all $s \geq s_0$. Given the least such s_0 and T, we define as the *final state* of M^Σ the last snapshot M^{s_0+T-1} of the periodic window $M^{\langle s_0,T\rangle}$. □

Definition 5.4 A Σ-interpretation M^Σ is called *progressively deterministic* if there exist $s_0, w \in \mathcal{N}_0$ such that for all $s \geq s_0$ the snapshot M^s functionally depends on its w predecessor states, i.e. $M^s = f(M^{\langle s-w,w\rangle})$ for all $s \geq s_0$ and some function $f : (2^{\mathcal{B}_P})^w \rightarrow 2^{\mathcal{B}_P}$. □

The following proposition is a direct consequence of the previous definitions.

Proposition 5.5 *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 $s_2 > s_1 \geq s_0$ such that $M^{\langle s_1,w\rangle} = M^{\langle s_2,w\rangle}$. As every state $s \geq s_0$ is determined by its w predecessors, we have $M^{s_1+i} = M^{s_2+i}$ for all $i \in \mathcal{N}_0$. So let $T = s_2 - s_1$, and the claim follows. ■

Obviously, an ultimately periodic model is characterized by an initial window $M^{\langle 0,s_0\rangle}$ and a periodic window $M^{\langle s_0,T\rangle}$ and thus can be finitely represented. In the presence of negation, such a periodicity cannot be guaranteed for all common semantics: e.g. the program with the two rules $[S]\ \mathsf{p} \leftarrow [S]\ \neg\ \mathsf{q}$ and $[S]\ \mathsf{q} \leftarrow [S]\ \neg\ \mathsf{p}$ has two different stable snapshots $\{\mathsf{p}\}$ and $\{\mathsf{q}\}$ for *every* state $s \in \Sigma$. Consequently, not all stable models of P are periodic. However, for the perfect model $\mathcal{M}$ we have the following result.[4]

Theorem 5.6 *The perfect model of a Σ-stratified Statelog program P is ultimately periodic.*

Proof Let P_{inst} 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_{inst} are built from finitely many propositional atoms.

We first rewrite the rules of P_{inst}, such that every rule contains at most one state variable S. This is accomplished as follows:

Given a rule r of P_{inst} which contains the state variable S in the head and some other state variable T in the body, we group together all literals of the body which contain T:[5]

$$[S]\ \ldots\ \leftarrow\ \ldots\ [T+k_1]L_1, \ldots, [T+k_m]L_m\ \ldots$$

[4] A similar result has been established in [7]. A difference is, that we need not consider strictly progressive rules when computing the dependency graph, which generalizes the usual notion of stratification. This is due to the progressive reading of rules, which in addition allows a simpler proof.

[5] It is easy to see that Σ-ground state terms can be eliminated from rules by the introduction of auxiliary predicates. Therefore we assume that rules do not contain Σ-ground state terms.

We rewrite r by replacing the group $[T+k_1]L_1, \ldots, [T+k_m]L_m$ by the atom $[S]L_{new}$ where L_{new} is a new auxiliary predicate. We add the following rules to P_{inst}

$$\begin{array}{rcl} [S+k_{max}]L_{new} & \leftarrow & [S+k_1]L_1, \ldots, [S+k_m]L_m \\ [S+1]L_{new} & \leftarrow & [S]L_{new} \end{array}$$

where $k_{max} = \max(k_1, \ldots, k_m)$. The purpose of the first rule is to remember in L_{new} whether the group of literals has been satisfied at some state s_0 in the past. If this is the case, this information is propagated to all future states by the second rule. It should be clear that this rewriting, when applied for all groups with different state variables, yields an equivalent set of rules P'_{inst} where each rule contains at most one state variable.[6]

Now it is easy to show that $\mathcal{M}$ is progressively deterministic: the Σ-instances of rules of P'_{inst} only refer to a fixed number of states. Since only progressive rule extensions are considered, the state s in the head of the rule is defined using predecessors of s and s itself. Given the input of w predecessors $\mathcal{M}^{\langle s-w,w\rangle}$ the perfect model semantics defines a *unique* snapshot $\mathcal{M}^s$. Therefore $\mathcal{M}$ is progressively deterministic. ■

Remark Given a progressive rule $r:\ [S+k_0] \pm H \leftarrow [S+k_1] \pm B_1, \ldots, [S+k_n] \pm B_n$, the number of predecessor states w on which the derivation of H depends is called the *range* of r. We have $w = \max_{0 \le i \le n}(k_0 - k_i)$. The *range of a program* P is the maximal range of rules of P.

5.2 Computing the Perfect Model

The actual computation of $\mathcal{M}$ can be accomplished along the lines of the well-known *iterated fixpoint computation* [3].

First, the dependency graph $\mathcal{D}_P$ of rules P' which are progressive but not strictly progressive is computed using a straightforward modification of unification (essentially, one has to take into account state terms and the progressive reading of rules). Next, the strongly connected components (SCCs) are determined. If a SCC contains a negative arc, P is not Σ-stratified. Otherwise, a topological sort of the SCCs yields a partition of P', the stratification $L_1, \ldots, L_k$.

After these preprocessing steps, the computation of $\mathcal{M}$ proceeds by successively computing the snapshots $\mathcal{M}^0, \mathcal{M}^1, \ldots$ until the period is detected: given the range w of P, one has to check for the earliest repetition of a window $\mathcal{M}^{\langle s,w\rangle}$ during the computation of $\mathcal{M}$.

For a given state s, the computation of the snapshot $\mathcal{M}^s = (\mathit{Del}(s), \mathit{True}(s))$ is accomplished as follows: the $\mathit{Del}(s)$ part is computed before $\mathit{True}(s)$ since it is smaller according to the order on state terms (cf. (2) in section 4.2). In a first step the strictly progressive rules which define $\mathit{Del}(s)$ are iterated until a fixpoint is reached (note that each snapshot is finite). These rules correspond to the lowest stratum L_0 w.r.t. the current state s, since they solely depend on already computed predecessor states of s. The next step consists in iterating the remaining (not strictly progressive) rules which define $\mathit{Del}(s)$ according to the stratification $L_1, \ldots, L_k$, until the $\mathit{Del}(s)$ part of the snapshot is computed. $\mathit{True}(s)$ is computed in the same way, i.e. first the strictly progressive rules (including frame rules), then the remaining rules according to the given stratification are iterated.

[6] [7] uses a similar rewriting. The main difference is that for Statelog information is propagated only from past to future states.

5.2.1 State Transitions between Final States

We now turn to the question how the transition from the current to a new final state is modelled in our framework. It turns out that we can view the database evolution as an incremental computation of $\mathcal{M}$.

Given a Statelog program P we assume that the EDB, IDB and the update procedures in TB are fixed while the event base EB is successively enlarged by the occurrence of events. In other words, we conceive the event base as a parameter of the perfect model. In order to emphasize this, we write $\mathcal{M}(EB)$ for the perfect model of P.

Assume that at a certain time instant the event base is EB_{t_i}. Since $\mathcal{M}(EB_{t_i})$ is ultimately periodic, a unique final state s_f is defined by EB_{t_i}. We require that when new events occur, these occurrences are assigned to the final state s_f or some $s > s_f$, thereby extending EB_{t_i} to $EB_{t_{i+1}}$. It is easy to see that $\mathcal{M}(EB_{t_{i+1}})$ and therefore a new final state can be computed as an *extension* of $\mathcal{M}(EB_{t_i})$: in order to compute $\mathcal{M}(EB_{t_{i+1}})$, it is sufficient to remember a window of w predecessor states of the final state s_f. Earlier states have no influence on $\mathcal{M}(EB_{t_{i+1}})$ because (i) events may not occur in the past (i.e. for some $s < s_f$) and (ii) rules only effect the present and future states due to their progressive reading.

5.2.2 Guarded Programs

As already mentioned, the computation of a snapshot $\mathcal{M}^s$ depends on a fixed window of w predecessors of s. While for a given transition base, w is a constant,[7] the length T of the periodic window may be exponential in the size of the EDB (this can be seen using a program from [8] whose semantics coincides with our semantics $\mathcal{M}$, cf. appendix A). Thus detecting the period may require storing exponentially many snapshots of $\mathcal{M}$. Clearly, it is desirable to identify program classes which allow to detect and store the periodic window efficiently. An example for such a class are the *guarded programs* defined below. The main idea is that rules which define a new state are guarded by some event in the body and that events are not defined by rules. While such user-definable events allow event-based programming similar to [22], they make detection of the period difficult. Although guarded programs are less expressive than general Statelog programs, many practical update procedures including the examples from section 3 can be expressed with guarded rules.

Definition 5.7 (Guarded Rules) A non-local Statelog rule r with head $[S+k] \pm H$ is *guarded* if there is a *positive* occurrence of some event $[S+k']\nabla E$ in the body of r for some $k' \leq k$. $[S+k']\nabla E$ is called a *guard* for r.

A Statelog program is called guarded if all of its non-local rules, apart from frame rules, are guarded. □

Although local rules and frame rules are not guarded, they have no influence on the period if all other rules are guarded:

Theorem 5.8 *The perfect model of a guarded Σ-stratified program P is ultimately periodic with period $T = 1$.*

[7] Typically, rules refer only to few predecessor states, e.g. in section 3 the maximal range w of rules is 2.

Proof We assume that in P data variables have been substituted by terms from $\mathcal{U}_P$ in all possible ways.

Assume for the moment, that P only contains local and frame rules. The snapshot $\mathcal{M}^{s+1}$ is defined by the "input" $\mathcal{M}^s$ through frame rules of the form $[S+1]R \leftarrow [S]R, \neg[S+1]{-}R$ and by local rules which only refer to $s+1$. Since $Del(s+1)$ is only defined by local rules which refer to the $Del(s+1)$ part of $\mathcal{M}^{s+1}$ in the body,[8] it follows that for all states $s+1$, $Del(s+1)$ is equal to some fixed set $Del \subseteq \mathcal{B}_P$. Therefore we can eliminate the dependency on $Del(s+1)$ by a rule rewriting which "reduces" P by Del:
Let L by a literal in the body of a rule r such that L refers to the Del part of a state. If $Del \models L$ then L is removed from r. If $Del \not\models L$ then the rule r itself is removed. After this model-preserving reduction, the remaining rules of P are either local rules or of the form $[S+1]R \leftarrow [S]R$. The latter rules copy the "input" (from the initial state) to the next state, but do not derive any new information in subsequent states: they only support atoms which have already been derived in the previous state. Consequently, there is no change in subsequent snapshots, so the perfect model is periodic with $T = 1$.

Now consider the general case, where P also contains guarded rules. Since the event base is always finite, all but finitely many Σ-instances of these guarded rules are blocked. Therefore, for all sufficiently large s, the only rules which define $\mathcal{M}^s$ are local and frame rules from which the desired result follows. ■

6 Conclusion and Outlook

We have presented Statelog, an extension of Datalog which allows to specify deterministic procedures for EDB and IDB updates, as well as static and dynamic integrity constraints. By incorporating state identifiers into the language, declarative, model-based semantics such as the well-founded or stable semantics can be applied to it. In this paper, we have concentrated on the perfect model semantics for Σ-stratified programs. The periodic structure of the perfect model is used to define the final database state and termination of the update program. We have introduced the class of guarded programs which allow efficient computation of the final state.

In [22] an approach to unify deductive and active databases is presented. The way updates are treated is similar to our approach. Update requests to EDB relations are stored in a so-called *event-queue* and processed successively. In addition to events which allow the insertion and deletion of facts, the user may define new event types. While this results in great flexibility, guaranteeing program termination seems to be more difficult than in Statelog and lies — at least partially — in the responsibility of the programmer.

From a logic point of view, Statelog is closely related to the temporal deductive database language $Datalog_{1S}$ [9]. In [7] it is shown that the perfect model of a stratified Datalog_{1S} program is ultimately periodic. However, [9] do not deal with database updates but use Datalog_{1S} for temporal reasoning. The main difference between Statelog and Datalog_{1S} is that Statelog interprets rules by their progressive extension. This seems to be a natural prerequisite in our context and allows to model database evolution as an incremental computation of the perfect model semantics.

[8] Consider the ordering on state terms (2) in section 4.2

The Datalog extensions proposed in [2] have a procedural semantics which is defined by a bottom-up evaluation and which does not coincide with model-based semantics known from the area of logic programming. [14, 15] use dynamic logic to define semantics of updates but also rely on a certain evaluation strategy (top-down). [4] propose *Transaction Logic*, a very general and powerful *framework* for numerous phenomenons of state change including database updates and planning and consequently leave the standard framework of logic programming. In contrast, we have presented a "minimal" extension of Datalog which is sufficient to specify updates in much the same way as queries.

Future work will focus on alternative ways for the treatment of IDB updates, e.g. by integrating intensional updates [5]. Another important issue is the identification of tractable program classes like the class of guarded programs, for which the overhead of update evaluation is small.

Statelog may be implemented on top of existing deductive systems like XSB-Prolog [18] or Coral [17]. Alternatively, the techniques from the database programming language Heraclitus[Alg,C] [11] which supports *deltas* as first class citizens, multiple virtual states, active rules and a wide variety of semantics, may be useful for an efficient implementation of state transitions.

Acknowledgments We wish to thank Jürgen Frohn, Paul Th. Kandzia, Heinz Uphoff and Klaus Ries for fruitful discussions and comments.

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

[5] F. Bry. Intensional updates : Abduction via deduction. In *Proc. Intl. Conference on Logic Programming*, 1990.

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

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

[8] J. Chomicki and T. Imieliński. Temporal deductive databases and infinite objects. In *Proc. ACM Symposium on Principles of Database Systems*, 1988.

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

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

[11] 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.

[12] 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.

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

[14] S. Manchanda and D. S. 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.

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

[16] 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.

[17] R. Ramakrishnan, D. Srivastava, S. Sudarshan, and P. Seshadri. Implementation of the CORAL deductive database system. In *Proc. ACM SIGMOD Intl. Conference on Management of Data*, 1993.

[18] 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.

[19] J. D. Ullman. *Principles of Database and Knowledge-Base Systems*, volume 1. Computer Science Press, New York, 1988.

[20] 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.

[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] C. Zaniolo. A unified semantics for active and deductive databases. In 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.

A Appendix

The following program (1.1) from [8, 9] has a period T which is exponential in the size of the database. Its Datalog$_{IS}$ semantics coincides with our Statelog semantics since all rules are progressive. It describes the successive addition of 1 to an n-bit number. The following predicates are used:

one(X): bit X is one

zero(X): bit X is zero

changed(X): bit X is changed

unchanged(X): bit X is not changed

The EDB is given by the following set of facts:

[0] (zero(bit$_1$),..., zero(bit$_n$)).

[0] (next(bit$_1$,bit$_2$),..., next(bit$_{n-1}$,bit$_n$).

[0] changed(bit$_1$).

Given an n-bit number $N = (bit_n, \ldots, bit_1)$ the program (1.1) successively adds 1 to N. The first periodic repetition occurs after 2^n steps.

$$
\begin{array}{lcl}
[S+1]\ \mathsf{changed(bit_1)} & \leftarrow & [S]\ \mathsf{changed(bit_1)}. \\
[S+1]\ \mathsf{changed(Y)} & \leftarrow & [0]\ \mathsf{next(X,Y)}, \\
 & & [S]\ \mathsf{one(X)},\ [S+1]\ \mathsf{changed(X)}. \\
[S+1]\ \mathsf{unchanged(Y)} & \leftarrow & [0]\ \mathsf{next(X,Y)}, \\
 & & [S]\ \mathsf{zero(X)},\ [S+1]\ \mathsf{one(X)}. \\
[S]\ \mathsf{unchanged(Y)} & \leftarrow & [0]\ \mathsf{next(X,Y)}, \\
 & & [S]\ \mathsf{unchanged(X)}. \\
[S+1]\ \mathsf{one(X)} & \leftarrow & [S+1]\ \mathsf{changed(X)},\ [S]\ \mathsf{zero(X)}. \\
[S+1]\ \mathsf{one(X)} & \leftarrow & [S+1]\ \mathsf{unchanged(X)},\ [S]\ \mathsf{one(X)}. \\
[S+1]\ \mathsf{zero(X)} & \leftarrow & [S+1]\ \mathsf{changed(X)},\ [S]\ \mathsf{one(X)}. \\
[S+1]\ \mathsf{zero(X)} & \leftarrow & [S+1]\ \mathsf{unchanged(X)},\ [S]\ \mathsf{zero(X)}.
\end{array}
\tag{1.1}
$$