Answer Update for Rule-based Stream Reasoning*

Harald Beck and **Minh Dao-Tran** and **Thomas Eiter**
Institute of Information Systems, Vienna University of Technology
Favoritenstraße 9-11, A-1040 Vienna, Austria
{beck,dao,eiter}@kr.tuwien.ac.at

Abstract

Stream reasoning is the task of continuously deriving conclusions on streaming data. To get results instantly one evaluates a query repeatedly on recent data chunks selected by window operators. However, simply recomputing results from scratch is impractical for rule-based reasoning with semantics similar to Answer Set Programming, due to the trade-off between complexity and data throughput. To address this problem, we present a method to efficiently update models of a rule set. In particular, we show how an answer stream (model) of a LARS program can be incrementally adjusted to new or outdated input by extending truth maintenance techniques. We obtain in this way a means towards practical rule-based stream reasoning with nonmonotonic negation, various window operators and different forms of temporal reference.

1 Introduction

Stream reasoning [Della Valle *et al.*, 2009] emerged from stream processing [Babu and Widom, 2001] to reason about information from data streams in real time, and to perform evaluation continuously in order to obtain latest results. In this way, users may be provided with live results for instant decision making, depending on data streams that often come from sensors and may be combined with other information.

This pertains to many application scenarios, such as in public transport where real-time information about current vehicle locations, together with time tables may be used to reason about expected arrival times and ensuing travel options.

Example 1 Consider Fig. 1 (a), which depicts a tram line ℓ_1 and a bus line ℓ_2. Kurt is on his way to station m to go to the theater, located near station s. It usually takes the same time for the tram and the bus to get from m to s. Thus, Kurt could simply take the bus, which is expected to arrive first at m. However, he recalls that there are frequent traffic jams on the bus line. In this case, the tram could be the better option. ■

Unbounded data and high frequency of data arrival pose a challenge, especially to advanced reasoning over streams that

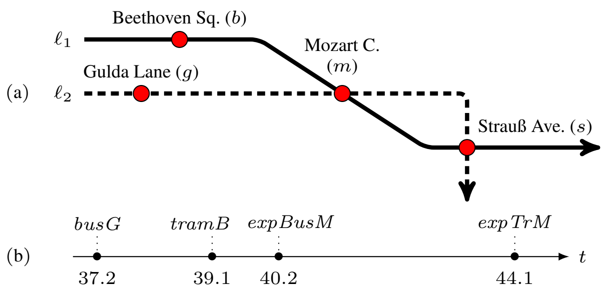


Figure 1: (a) Transportation map (b) Timeline (minutes)

aims to solve AI problems such as planning, monitoring, or decision making. The rule-based LARS language [Beck *et al.*, 2015] faces this by using *windows*, which were introduced in stream processing as partial stream snapshots for efficient evaluation and move on the timeline in query evaluation. Window operators may select data differently from a stream, e.g. all data in a time frame or the last n tuples. LARS has an Answer Set Programming (ASP) like semantics and supports also different means for time reference.

However, while windows reduce the data volume, efficient re-evaluation of a query remains an issue. A full re-evaluation of an (expensive) query may be impractical, the more as often only part of the data may have changed and moreover the final answer (e.g., a plan or decision) remains the same.

Example 2 (cont'd) Fig. 1 (b) illustrates actual and expected arrival times of vehicles. A bus appears at station g ($busG$) at minute 37.2. Based on this, we can immediately conclude the expected arrival time at m ($expBusM$). This inference does not need to be re-evaluated after the tram appears at b. ■

It is thus desired that re-evaluation can be done incrementally and fast. For expressive (intractable) languages such as LARS, this poses a particular challenge. Aiming at a good trade-off between efficiency and expressiveness, we consider incremental re-evaluation for a class of LARS programs, that is, a new answer stream (model) is computed from the previous answer stream and the new incoming data, under a suitable management of temporal information.

Our contributions can be briefly summarized as follows.

(1) *Stream-stratified programs.* Towards efficient evaluation, we introduce stream stratified programs, a fragment of LARS programs that can be split along the stream flow into layers

*This research has been supported by the Austrian Science Fund (FWF) projects P24090 and P26471.

that can be evaluated sequentially, where the output of lower layers serves as input stream for the next layer.

(2) *TMS extension.* We extend techniques from truth maintenance systems (TMS) [Rich and Knight, 1991], which were conceived to maintain models of non-monotonic theories, to deal with the streaming and temporal aspects of LARS on top of ASP. We extend the maintenance method for justification-based TMS [Doyle, 1979], which amount to logic programs under stable model semantics [Elkan, 1990], in two regards: (i) We introduce an extended set of atoms comprising window and temporal operators, and for temporal data management *time labels* to record validity information of atoms. (ii) We similarly extend central notions of *consequences* and *affected consequences* of TMS [Beierle and Kern-Isberner, 2000].

(3) *Generic Windows.* Our method works at a generic level, and any concrete window operator can be plugged in if certain basic functions to process incoming data, identify outdated input, and change the temporal status of atoms are available; time- and tuple-based windows have such functions, as well as typical partition-based windows.

The emerging algorithm is nontrivial and inherently involved due to the properties of temporal operators. For space reasons, we must confine to show how the method works and omit the technical details. Notably, for relevant yet less expressive settings it is much simpler.

Our results contribute to the foundations of answer maintenance in expressive stream reasoning languages; to the best of our knowledge, no similar results exist to date. The presented ideas and principles may be transferred to similar languages.

2 Preliminaries

2.1 Streams, Windows and Time Reference

We will gradually introduce the central concepts of LARS [Beck *et al.*, 2015] tailored to the considered fragment. If appropriate, we give only informal descriptions.

Throughout, we distinguish *extensional atoms* $\mathcal{A}^{\mathcal{E}}$ for input data and *intensional atoms* $\mathcal{A}^{\mathcal{I}}$ for derived information. By $\mathcal{A} = \mathcal{A}^{\mathcal{E}} \cup \mathcal{A}^{\mathcal{I}}$, we denote the set of *atoms*.

Definition 1 (Stream) *A stream $S = (T, v)$ consists of a* timeline T, *which is an interval in* $\mathbb{N}$, *and an* evaluation function $v : \mathbb{N} \mapsto 2^{\mathcal{A}}$. *The elements $t \in T$ are called* time points.

Intuitively, a stream S associates with each time point a set of atoms. We call S a *data stream*, if it contains only extensional atoms. To cope with the amount of data, one usually considers only recent atoms. Let $S = (T, v)$ and $S' = (T', v')$ be two streams s.t. $S' \subseteq S$, i.e., $T' \subseteq T$ and $v'(t') \subseteq v(t')$ for all $t' \in T'$. Then S' is called a *window* of S.

Definition 2 (Window function) *Any (computable) function w that returns, given a stream $S = (T, v)$ and a time point $t \in T$, a substream S' of S is called a* window function.

Important are *tuple-based* window functions, which select a fixed number of latest tuples, and *time-based* window functions, which select all atoms appearing in last n time points.

Example 3 (cont'd) To model the public transport scenario, we take a timeline $T = [1, 30000]$ with unit 0.1 secs. For readability, we write m for minutes, i.e., $T = [1, 50m]$. We have the data stream $D = (T, v)$, where v can be seen as a set of mappings $37.2m \mapsto \{busG\}$ and $39.1m \mapsto \{tramB\}$. We implicitly take $v(t) = \emptyset$ for all $t \in T \setminus \{37.2m, 39.1m\}$. Fig. 2 depicts the window $S' = (T', v)$ of D, where $T' = [36.7m, 39.7m]$. Obtained by applying a time-based window function with a size of 3 min (1801 time points) at time point $t = 39.7m = 23820$. There, a tuple-based window of size 1 would return $([39.1m, 39.7m], \{39.1 \mapsto \{tramB\}\})$. Note that D itself does not contain expected arrival times. ■

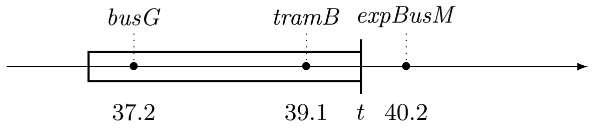


Figure 2: Window of last 3 min (1801 time points) at t=39.7

Window operators $\boxplus$. Window functions can be accessed in formulas by window operators. That is to say, an expression $\boxplus\alpha$ has the effect that α is evaluated on the "snapshot" of the data stream delivered by its associated window function $w_{\boxplus}$.

By dropping information based on time, window operators specify temporal *relevance*. For each atom in a window, we then want to control the semantics of its temporal *reference*.

Time Reference. Let $S = (T, v)$ be a stream, $a \in \mathcal{A}$ and $\mathcal{B} \subseteq \mathcal{A}$ static *background data*. Then, at time point $t \in T$,

- a holds, if $a \in v(t)$ or $a \in \mathcal{B}$;
- $\Diamond a$ holds, if a holds at some time point $t' \in T$;
- $\Box a$ holds, if a holds at all time points $t' \in T$; and
- $@_{t'} a$ holds, if $t' \in T$ and a holds at t'.

Next, the set $\mathcal{A}^+$ of *extended atoms* is given by the grammar

$$a \mid @_t a \mid \boxplus @_t a \mid \boxplus \Diamond a \mid \boxplus \Box a\,.$$

where $a \in \mathcal{A}$ and t is any time point. The expressions of form $\boxplus \star a$, where $\star \in \{@_t, \Diamond, \Box\}$, are the *window atoms*.

Example 4 (cont'd) Given D from Ex. 3, $busG$ holds at $t = 37.2m$, and thus $@_{37.2m} busG$ holds at any time point. Let $\boxplus^k$ denote the window operator associated with a time-based window function of size k. Then, $\boxplus^{3m} @_{37.2m} busG$ and $\boxplus^{3m} \Diamond busG$ hold at all time points $t \in [37.2m, 40.2m]$. ■

2.2 LARS Programs

We present a fragment of the formalism in [Beck *et al.*, 2015].

Syntax. A *rule* r is of the form $\alpha \leftarrow \beta(r)$, where $H(r) = \alpha$ is the *head* and $\beta(r) = \beta_1, \ldots, \beta_j, \text{not}\, \beta_{j+1}, \ldots, \text{not}\, \beta_n$ is the *body* of r. Here, α is of form a or $@_t a$, where $a \in \mathcal{A}^{\mathcal{I}}$, and each β_i is either an ordinary or a window atom.

We let $B(r) = B^+(r) \cup B^-(r)$, where $B^+(r) = \{\beta_i \mid 1 \leq i \leq j\}$ is the *positive* and $B^-(r) = \{\beta_i \mid j < i \leq n\}$ is the *negative body* or r. A (LARS) *program* P is a set of rules.

We say an extended atom α *occurs* in a program P, denoted by $\alpha \mathrel{\hat{\in}} P$, if $\alpha \in H(r) \cup B(r)$ for some rule $r \in P$. We define the α-*rules* of P as $P^H(\alpha) = \{r \in P \mid H(r) = \alpha\}$.

Example 5 (cont'd) Consider the program P in Figure 3. Rule (r_1) says that, if within the last 3 min ($\boxplus^{3m}$), at some time time T ($@_T$) a bus appeared at station g ($busG$) and

$$\begin{array}{lrcl}
(r_1) & @_{T+3m}\,\mathit{expBusM} & \leftarrow & \boxplus^{3m}@_T\,\mathit{busG}, \mathit{on}.\\
(r_2) & @_{T+5m}\,\mathit{expTrM} & \leftarrow & \boxplus^{5m}@_T\,\mathit{tramB}, \mathit{on}.\\
(r_3) & \mathit{on} & \leftarrow & \boxplus^{1m}\Diamond \mathit{request}.\\
(r_4) & \mathit{takeBusM} & \leftarrow & \boxplus^{+5m}\Diamond \mathit{expBusM}, \text{not}\ \mathit{takeTrM},\\
 & & & \text{not}\ \boxplus^{3m}\Diamond \mathit{jam}.\\
(r_5) & \mathit{takeTrM} & \leftarrow & \boxplus^{+5m}\Diamond \mathit{expTrM}, \text{not}\ \mathit{takeBusM}.
\end{array}$$

Figure 3: Program P, encoding the running example

the rule is activated (on) (triggered by (r_3) for one 1 min upon request), then the bus is expected to arrive 3 min later at station m ($@_{T+3m}\,expBusM$). Rules (r_4) and (r_5) express a choice to take a bus or tram at station m, if both are expected to arrive within the *next* 5 min ($\boxplus^{+5m}$). The bus recommendation further depends on the traffic status (not $\boxplus^{3m}\Diamond jam$). ■

Semantics. For a data stream $D = (T_D, \upsilon_D)$, any stream $I = (T, \upsilon) \supseteq D$ that coincides with D on $\mathcal{A}^{\mathcal{E}}$ is an *interpretation stream* for D. A tuple $M = \langle T, \upsilon, W, \mathcal{B}\rangle$, where W is a set of window functions and $\mathcal{B}$ is the background knowledge, is then an *interpretation* for D. Throughout, we assume W and $\mathcal{B}$ are fixed and thus also omit them.

Satisfaction by M at $t \in T$ is as follows: $M, t \models \alpha$ for $\alpha \in \mathcal{A}^+$, if α holds in (T, υ) at time t; $M, t \models r$ for rule r, if $M, t \models \beta(r)$ implies $M, t \models H(r)$, where $M, t \models \beta(r)$, if (i) $M, t \models \beta_i$ for all $i \in \{1, \dots, j\}$ and (ii) $M, t \not\models \beta_i$ for all $i \in \{j{+}1, \dots, n\}$; and $M, t \models P$ for program P, i.e., M is a *model* of P (for D) at t, if $M, t \models r$ for all $r \in P$. Moreover, M is *minimal*, if in addition no model $M' = \langle T, \upsilon', W, \mathcal{B}\rangle \neq M$ of P exists such that $\upsilon' \subseteq \upsilon$.

Definition 3 (Answer Stream) *An interpretation stream I is an answer stream of program P for the data stream $D \subseteq I$ at time t, if $M = \langle T, \upsilon, W, \mathcal{B}\rangle$ is a minimal model of the* reduct *$P^{M,t} = \{r \in P \mid M, t \models \beta(r)\}$. By $\mathcal{AS}(P, D, t)$ we denote the set of all such answer streams I.*

Example 6 (cont'd) Let $D' = (T, \upsilon')$ be the data stream which adds to the stream D from Example 3 the input $39.7m \mapsto \{request\}$. We get two answer streams $I_1 = (T, \upsilon_1)$ and $I_2 = (T, \upsilon_2)$ of P for D' at $t = 39.7m$, which both contain, in addition to the mappings in υ', $40.2m \mapsto \{expBusM\}$ and $44.1m \mapsto \{expTrM\}$. Answer stream I_1 additionally contains the recommendation $t \mapsto \{takeTrM\}$ and I_2 suggests $t \mapsto \{takeBusM\}$, since both vehicles are expected to arrive in the next 5 minutes. ■

2.3 Truth Maintenance Systems

Truth maintenance systems (TMS) [Rich and Knight, 1991] aim to implement nonmonotonic reasoning. We recall the general idea of *justification-based TMS* (JTMS) [Doyle, 1979], using notation as in [Beierle and Kern-Isberner, 2000].

A TMS $\mathcal{T} = (\mathcal{N}, \mathcal{J})$ consists of a set $\mathcal{N}$ of nodes and a set $\mathcal{J}$ of justifications J of the form $\langle I | O \to n\rangle$, where $I = \{in_1, \dots in_l\}$ and $O = \{o_1, \dots, o_m\}$ are sets of nodes. Intuitively, n is by J in (true), if all in_i's are in and no o_j's are in (they are all out), i.e., J "fires." A *model* M of $\mathcal{T}$ labels each node n with status in or out. Notably, the *admissible models* of $\mathcal{T}$ correspond to the answer sets (stable models) of the logic program $P_{\mathcal{T}} = \{r_J \mid J \in \mathcal{J}\}$ where r_J is the rule $n \leftarrow in_1, \dots, in_l, \text{not}\, o_1, \dots, \text{not}\, o_m$ [Elkan, 1990].

Given an admissible model M of $\mathcal{T}$ and a new justification $J_0 = \langle I_0 | O_0 \to n_0\rangle$, the TMS algorithm aims to update M to an admissible model M' of $\mathcal{T}' = (\mathcal{N}, \mathcal{J} \cup \{J_0\})$. In that, key notions for nodes n are the following. The *consequences* ($Cons(n)$) are nodes with a justification that involves n. The *support* consists of a set ($Supp(n)$) of nodes witnessing the (in)validity of n. Finally, the *affected consequences* ($ACons(n)$) are nodes whose support involves n.

Briefly, if J_0 fires in M and n_0 was not in, only the nodes in $ACons(n_0)$ are re-evaluated (after setting their status to $unknown$) proceeding along the dependencies, where $Supp$ is updated. For a node n, its justifications are checked whether they are (i) founded (in)valid, or (ii) unfounded (in)valid. In case (i), the new status of n (in or out) is settled, as definitely some (resp. no rule) will fire; if not, in case (ii) a guess is made and propagated, with backtracking on failure.

3 Stream-stratified Programs

Stratification [Apt *et al.*, 1988] is a well-known method to split a logic program with acyclic negation into strata that can be evaluated successively. We carry this over to stream inputs and define *stream stratification*, which will allow us to evaluate and predict the temporal validity of atoms hierarchically.

In the sequel, $\mathcal{A}^+_{sub}(P)$ are the extended atoms α occurring in P, plus the extended atoms of form $@_t a$ and a inside α.

Definition 4 *The* (stream) dependency graph *of a program P is a directed graph $G_P = (V, E)$, where $V = \mathcal{A}^+_{sub}(P)$ and E contains edges*

- $\alpha \to^{\geq} \beta$, *if $\exists\, r \in P$ s.t. $\alpha \in H(r)$ and $\beta \in B(r)$,*
- $@_t a \leftrightarrow^{=} a$, *if $@_t a \in \mathcal{A}^+(P)$, and*
- $\boxplus \star a \to^{>} a$, *if $\boxplus \star a \in \mathcal{A}^+(P)$, where $\star \in \{@_t, \Diamond, \Box\}$.*

The intuition of an edge $\alpha \to^{\succeq} \beta$ is that the truth value of α depends on the one of β. The label $\succeq \in \{\geq, =, >\}$ indicates the possibility to slice the program into parts (strata) that can be evaluated levelwise. For instance, $\boxplus\Diamond a$ can only be evaluated *after* the value of the ordinary atom a is known. This dependency would be reflected by an edge $\boxplus\Diamond a \to^{>} a$. On the other hand, $@_t a$ must be co-evaluated with a.

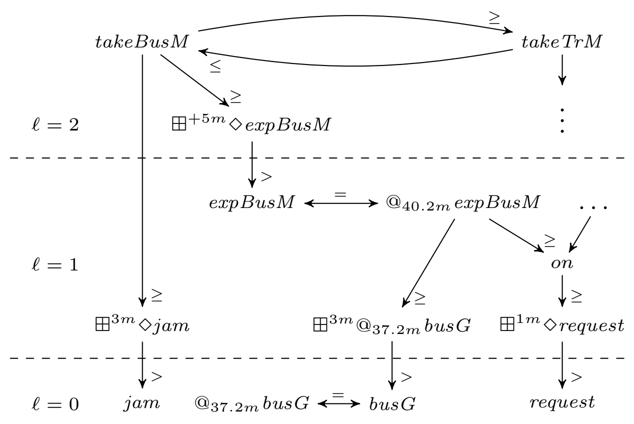
Example 7 Let $P' {=} \{@_t x \leftarrow \boxplus^3 @_t y\}$. Then, $G_{P'} = (V, E)$, where $V = \mathcal{A}^+_{sub}(P') = \{@_t x, x, \boxplus^3 @_t y, @_t y, y\}$ and

$$E = \left\{ \begin{array}{ll} @_t x \to^{\geq} \boxplus^3 @_t y, & \boxplus^3 @_t y \to^{>} y, \\ @_t x \leftrightarrow^{=} x, & @_t y \leftrightarrow^{=} y \end{array} \right\}. \qquad ■$$

Based on this, we define the notion of stream stratification.

Definition 5 (Stream stratification) *Let P be a program with stream dependency graph $G_P = (V, E)$. A mapping $\lambda\colon \mathcal{A}^+(P) \to \{0, \dots, n\}$, $k \geq 0$, is called a* stream (s-)stratification *for P, if $\alpha \to^{\succeq} \beta \in E$ implies $\lambda(\alpha) \succeq \lambda(\beta)$ for all $\succeq \in \{\geq, =, >\}$. We call P* stream (s-)stratified, *if it has a stream stratification.*

Stream stratification captures the intuition that the dependency graph has no cycle with an edge $\alpha \to^{>} \beta$, i.e., no recursion through window operators. Using standard methods,



takeBusM
takeTrM
$\ell = 2$
$\boxplus^{+5m}\Diamond expBusM$
expBusM
$@_{40.2m}\, expBusM$
$\ell = 1$
on
$\boxplus^{3m}\Diamond jam$
$\boxplus^{3m}@_{37.2m}\, busG$
$\boxplus^{1m}\Diamond request$
$\ell = 0$
jam
$@_{37.2m}\, busG$
busG
request

Figure 4: Stream dependency graph (partial)

we can decide in linear time whether P is s-stratified, i.e., check that no strongly connected component of G_P contains an edge $\alpha \rightarrow^{>} \beta$ and if so, compute an s-stratification λ.

Example 8 (cont'd) Program P' of Ex. 7 is s-stratified. We take $\lambda(@_t y)=\lambda(y)=0$ and $\lambda(\boxplus^3 @_t y)=\lambda(@_t x)=\lambda(x)=1$. ■

From λ, we naturally obtain a partition of P into *strata* $P_0, \ldots, P_n$ by $P_i = \{r \in P \mid \lambda(H(r)) = i\}$. Without loss of generality, we may assume that each P_i is non-empty.

Example 9 Figure 4 depicts a partial s-stratification of the program P in Ex. 5 and the induced three strata P_0, P_1, P_2. ■

Like in usual stratified logic programs, we can evaluate an s-stratified program P stepwise, i.e., stratum per stratum, where the results of the previous strata serve as input to the current stratum. This is formally captured by the following result.

Proposition 1 *Let D be a data stream and let P be an s-stratified program with strata $P_0, \ldots, P_n$. Define $\mathcal{I}_0^{P,t}=\mathcal{AS}(P_0, D, t)$ and $\mathcal{I}_{k+1}^{P,t}=\bigcup_{I \in \mathcal{I}_k^{P,t}} \mathcal{AS}(P_{k+1}, I, t)$ for $0 \leq k < n$. Then, we have $I \in \mathcal{AS}(P, D, t)$ iff $I \in \mathcal{I}_n^{P,t}$.*

Intuitively, at each layer k, s-stratification ensures that inferences of $P_0 \cup \cdots \cup P_k$ will not be retracted by adding P_{k+1}, and new inferences come exclusively from P_{k+1}.

4 TMS Extension

We now present technical ingredients of our update algorithm for answer streams, which are reflected in a TMS extension suitable for incremental evaluation of temporal information.

Towards an according TMS data structure $\mathcal{M}$, we introduce a *label* of an extended atom α as $L(\alpha) = (s, \mathcal{T})$ where $s \in \{in, out, unknown\}$ is the *status* of α and $\mathcal{T}$ is a set of time intervals, meaning that α has status s during each interval in $\mathcal{T}$. (We will write $[t_1, t_2]$ instead of $\{[t_1, t_2]\}$.) Given a partial labeling of atoms, i.e., some atoms have status $unknown$, we define different types of (un)founded (in)validity of rules, where for readability, we use obvious abbreviations. For instance, "$B^+(r)$ some out" means that some atom in the positive body of r has status out:

$$\begin{array}{rcl}
fVal(r) & \Leftrightarrow & B^+(r) \text{ all } in \text{ and } B^-(r) \text{ all } out \\
fInval(r) & \Leftrightarrow & B(r) \text{ none } unknown \text{ and} \\
 & & (B^+(r) \text{ some } out \text{ or } B^-(r) \text{ some } in) \\
ufVal(r) & \Leftrightarrow & B^+(r) \text{ all } in \text{ and } B^-(r) \text{ none } in \text{ and} \\
 & & B^-(r) \text{ not all } out \\
ufInval(r) & \Leftrightarrow & \neg fVal(r) \text{ and } \neg fInval(r) \text{ and } \neg ufVal(r)
\end{array}$$

Informally, $fVal(r)$ (resp. $fInval(r)$) fully determines satisfaction (resp. falsification) of the rule body based on the atoms appearing in r. On the other hand, $ufVal(r)$ also holds for partial labelings and thus leaves falsification open.

We extend the TMS notions in Section 2.3 to deal with the temporal and streaming aspects in our setting as follows.

Consequences. Let $P_0, \ldots, P_n$ be the strata of P. To capture the structural dependencies between atoms $a \in \mathcal{A}$ and extended atoms $\alpha \in \mathcal{A}^+$ in P, we define the *consequences*

$$Cons(\alpha) = Cons_h(\alpha) \cup Cons_w(\alpha) \cup Cons_@(\alpha), \text{ where}$$

- the *head consequences* are all heads of rules with α in the body, i.e., $Cons_h(\alpha) = \{H(r) \mid \exists r \in P, \alpha \in B(r)\}$,
- the *window consequences* are $Cons_w(a) = \{\boxplus \star a \mathrel{\hat{\in}} P\}$, where $\star \in \{\Diamond, \Box\}$ and $a \in \mathcal{A}$; and $Cons_w(@_t\beta) = \{\boxplus @_t \beta\}$,
- the @-*consq.* are $Cons_@(@_{t'} a) = \{a\}$ if $@_{t'} a \mathrel{\hat{\in}} P$, else $\emptyset$.

$Cons^*(\alpha)$ denotes the transitive closure of $Cons(\alpha)$.[1]

Support. The *support* of α is defined by

$$Supp(\alpha) = Supp^+(\alpha) \cup Supp^-(\alpha) \cup Supp^@(\alpha),$$

where $Supp^+$ and $Supp^-$ reflect the support in TMS, i.e., the witnessing of the status of an atom via rules. Formally:

- $Supp^+(\alpha)$ is the *positive support* of α, which is empty if $s(\alpha) \neq in$ and otherwise the union of the bodies $B(r)$ of all rules $r \in P^H(\alpha)$ that are founded valid at t;
- $Supp^-(\alpha)$ is the *negative support* of α, which is empty if $s(\alpha) \neq out$. Else, if no rule $r \in P^H(\alpha)$ is founded valid, it contains for each $r \in P^H(\alpha)$ either an atom $\beta_i \in B^+(r)$ with status out or an atom $\beta_i \in B^-(r)$ with status in;
- $Supp^@(\alpha)$ is the @-*support* of α, which is empty if α is a non-ordinary atom or $s(\alpha) = out$; otherwise, it contains all atoms $@_{t'}\alpha$ with $s(@_{t'}\alpha) = in$. Intuitively, @-support captures label witnesses for an ordinary atom α of the form $@_{t'}\alpha$.

Affected consequences. The *affected consequences* of α are

$$ACons(\alpha) = \{\beta \in Cons(\alpha) \mid \alpha \in Supp(\beta)\} \cup Cons_@(\alpha),$$

i.e., all consequences that have α in their *support* and the ordinary atom extracted from α in case it is an @-atom. Furthermore, $ACons^*(\alpha)$ is the transitive closure of $ACons(\alpha)$. Given a set A of ordinary atoms, we define the affected consequences of A at a time point t as

$$ACons(A, 0) = \textstyle\bigcup_{a \in A} Cons^*(a),$$

$$ACons(A, t) = \textstyle\bigcup_{a \in A} ACons^*(@_t a) \setminus A, \quad \text{if } t > 0.$$

Distinguishing $t > 0$ from $t = 0$ is due to the fact that when starting from scratch at $t = 0$, no support of atoms is established and likewise no affected consequences; we thus rely

[1] I.e., the least set C s.t. $Cons(\alpha) \subseteq C$ and $\bigcup_{\beta \in C} Cons(\beta) \subseteq C$.

Algorithm 1: AnswerUpdate$(t, D, \mathbf{var}\ \mathcal{M})$

```
Input: Time point $t$, data stream $D$, TMS structure $\mathcal{M}$
       reflecting an answer stream for an s-stratified
       program $P = P_0, \ldots, P_n$ on $D$ at time $t' < t$
Output: Update of $\mathcal{M}$ reflecting an updated answer
        stream for $P$ at time $t$ or fail

foreach stratum $\ell := 1 \to n$ do
    $C := \emptyset$ and $L'$ := current labels
    foreach $\langle \alpha, \omega \rangle \in Expired(\ell, t', t)$ do
        ExpireInput$(\alpha, \omega, t)$ and $C := C \cup \{\omega\}$
    foreach $\langle \alpha, \omega, t_1 \rangle \in Fired(\ell, t', t)$ do
        FireInput$(\alpha, \omega, t_1)$ and $C := C \cup \{\omega\}$
    UpdateTimestamps$(C, L', \ell, t)$
    SetUnknown$(\ell, t)$
    repeat
        if SetRule$(\ell, t)$ fails then return fail
        if MakeAssignment$(\ell, t)$ fails then return fail
    until no new assignment made
    SetOpenOrdAtomsOut$(\ell, t)$
    PushUp$(\ell, t)$
```

on the syntactic dependencies given by $Cons$. The affected consequences of A at t restricted to a stratum ℓ are

$$ACons(A, t, \ell) = \{\alpha \in ACons(A, t) \mid \alpha \mathbin{\hat{\in}} P_\ell\}.$$

Example 10 Consider a program P' containing only the following ground instance r_2' of rule (r_2) in Example 5:

$$(r_2') \quad @_{44.1m}\, expTrM \;\leftarrow\; \boxplus^{5m}\, @_{39.1m}\, tramB, on.$$

We have the following consequences:

$$\begin{aligned}
Cons_w(@_{39.1m}\, tramB) &= \{\boxplus^{5m} @_{39.1m}\, tramB\}\\
Cons_h(\boxplus^{5m} @_{39.1m}\, tramB) &= \{@_{44.1m}\, expTrM\}\\
Cons_h(on) &= \{@_{44.1m}\, expTrM\}\\
Cons_@(@_{44.1m}\, expTrM) &= \{expTrM\}
\end{aligned}$$

Suppose we are given the following labels:

$$\begin{aligned}
L(\boxplus^{5m} @_{39.1m}\, tramB) &= (in, [39.1m, 44.1m])\\
L(on) &= (in, [39.7m, 40.7m])
\end{aligned}$$

Then, r_2' is founded valid, i.e., $fVal(r_2')$ holds, and we have:

$$\begin{aligned}
Supp^+(@_{44.1m}\, expTrM) &= \{\boxplus^{5m} @_{39.1m}\, tramB, on\}\\
ACons(\boxplus^{5m} @_{39.1m}\, tramB) &= \{@_{44.1m}\, expTrM\}
\end{aligned}$$

5 Answer Update Algorithm

This section presents the AnswerUpdate algorithm. Let P be an s-stratified LARS program and D be a data stream. Given a time point t, a time point $t' < t$, and an answer stream of P for D at t', reflected in a TMS data structure $\mathcal{M}$. AnswerUpdate modifies $\mathcal{M}$ such that a new (updated) answer stream at time t is reflected in it. To this end, it iterates over the strata of the program P and updates the status of atoms and their temporal validity. At each stratum ℓ, we have to consider two orthogonal aspects:

(O1) For extended atoms with unchanged status, we only update the time labels and set a label to the maximal extent for which the status can be guaranteed. For window atoms, this guarantee comes, e.g., from reappearance of the same tuple in the data stream in case of $\Diamond$, or from absence of tuples in case of $\Box$. For atoms in rule heads, the label update depends on label updates of window atoms in the bodies.

(O2) Re-evaluate the status of atoms affected by the change in the input. This is inspired by the TMS algorithm, which tries to keep the answer untouched as much as possible.

Expire/Fire Input. The algorithm collects two sets $Fired(\ell, t', t)$ of firing atoms and $Expired(\ell, t', t)$ of expired (extended) atoms between time points t' and t at stratum ℓ. Intuitively, $Fired(1, t', t)$ is collected from $v(t' + 1)$ to $v(t)$, i.e., the window atoms of the tuples from the input stream fire at stratum 1. For $\ell > 1$, $Fired(\ell, t', t)$ is computed from the atoms concluded at lower strata. $Expired(\ell, t', t)$ is calculated by a supportive function of each window atom, taking into account the current time point, the content of the window and the fired input. Both functions return an atom α and a window atom ω in which α expires/fires. $Fired(\ell, t', t)$ also returns a time point $t_1 \leq t$ at which the firing happens. Then, ExpireInput and FireInput update the labels of window atoms at ℓ. Note that FireInput can be optimized by not firing incoming inputs that are not anymore relevant at t. The affected window atoms are collected in a set C.

Time and status adjustment. The processing continues to determine the labels of rule heads wrt. the orthogonal aspects (O1) and (O2). For (O1), UpdateTimestamps finds window atoms in C having unchanged labels (by comparing with L'), based on that identifies rules where the applicability does not change and thus the time label of the head can be extended. For (O2), the TMS method is applied to set the labels of remaining atoms. We set the labels of atoms (except window atoms) affected by incoming and expiring atoms to $unknown$ (SetUnknown), identify founded (in)valid rules and set the status of respective head (SetRule), and make assignments for head and $unknown$ body atoms in an unfounded (in)valid rule (MakeAssignment). SetRule and MakeAssignment detect inconsistency when they set the label of an atom to *in*/*out* but that label was already set to *out*/*in* for the same interval. The algorithm then stops and outputs 'fail.'

The repeat loop ensures that rules that become founded will be processed by SetRule. After the loop, ordinary atoms that do not occur as rule heads may still be $unknown$, if some of their @-atoms do occur (e.g., $expTrM$). Such atoms are labeled $(out, [t, t])$ in SetOpenOrdAtomsOut. Finally, rule heads are pushed as input to the next stratum by PushUp(ℓ, t).

Initialization. The first answer stream reflects the initial state with an empty input stream at time 0. To obtain it, we set in $\mathcal{M}$ all atom labels to $(out, [0, 0])$ and all transitive consequences of the input atoms except window atoms to $unknown$. We then call AnswerUpdate$(0, D_0, \mathcal{M})$, where $D_0 = ([0, 0], \{\})$. This results in a labeling at time 0 and supports for atoms in it; from $t \geq 1$, the sets of affected consequences are in place for calling AnswerUpdate$(t, D, \mathcal{M})$.

Example 11 (cont'd) Consider the program in Example 5. The initialization sets all labels to $(out, [0, 0])$ and thus a negative support for all rule heads. Maintenance starts with an empty answer stream and waits for input.

Simple Updates. At $t = 37.2m$, $busG$ arrives as first atom. In the window obtained by $\boxplus^{3m}$, atom $@_{37.2m} busG$ will hold for 3 min. This is reflected in the algorithm by FireInput which sets $L(\boxplus^{3m} @_{37.2m} busG) = (in, [37.2m, 40.2m])$. The relevant ground rule of (r_1),

$$(r_1') \quad @_{40.2m} expBusM \leftarrow \boxplus^{3m} @_{37.2m} busG, on$$

should not fire, since on cannot be derived. Indeed, the status of on is still out and so is $@_{40.2m} expBusM$ after SetRule. At $t = 39.1m$, atom $tramB$ arrives, and we have $L(\boxplus^{5m} @_{39.1} tramB) = (in, [39.1m, 44.1m])$. Similarly, ground rule (r_2') (shown in Example 10) cannot fire, i.e., $@_{44.1m} expTrM$ will be *out*.

Update with rule firing. As shown in Fig. 2, Kurt sends his first request at $t = 39.7m$. FireInput sets the label of $\boxplus^{1m} \diamond request$ to $(in, [39.7m, 40.7m])$. By SetRule, this label is then first carried over to predicate on due to rule (r_3). Furthermore, the labels of $@_{40.2m} expBusM$ and $@_{44.1m} expTrM$ are set to $(in, [39.7m, 40.2m])$ and $(in, [39.7m, 40.7m])$ due to rules (r_1) and (r_2), respectively, where according ground rules are now founded valid.

Pushing these conclusions to the next stratum updates $L(\boxplus^{+5m} \diamond expBusM)$ to $(in, [39.7m, 40.2m])$ and $L(\boxplus^{+5m} \diamond expTrM)$ to $(in, [39.7m, 44.1m])$.

Choices. The last two rules of P are still unfounded valid, since the status of the negative atoms is $unknown$. The choice of MakeAssignment to take the bus materializes as status in for $takeBusM$ and out for $takeTrM$.

Nonmonotonic negation. Later at time $t = 40.0m$ a traffic jam report comes in as $v(40.0m) = \{jam\}$, thus at stratum 2, rule (r_4) is unfounded invalid due to the negation of the window atom $\boxplus^{3m} \diamond jam$ which is set to in by FireInput. The head cannot be concluded anymore, and MakeAssignment flips the previous statuses of $takeBusM$ and $takeTrM$.

Efficient adjustment. Kurt sends another request $t = 40.1m$, so FireInput sets $L(\boxplus^{1m} \diamond request) = (in, [40.1m, 41.1m])$ and $C = \{\boxplus^{1m} \diamond request\}$, since it was in before. Thus, UpdateTimestamps updates the time label $tm(on)$ to $[39.7m, 41.1m]$ due to rule (r_3) and furthermore updates $tm(@_{44.1m} expTrM)$ to $[39.1m, 41.1m]$ due to rule (r_2). The rest on stratum 2 proceeds similarly as before. ■

Correctness. Algorithm AnswerUpdate computes new labels for atoms in an s-stratified program P that get affected by the update of the input stream D at a time point t. Its correctness relies on the generic subroutines FireInput, ExpireInput to faithfully reflect the functionalities of an associated window function w. Under this assertion, the new labeling in $\mathcal{M}$ can be translated into an answer stream from $\mathcal{AS}(P, D, t)$. In particular, for time-based windows, which have simple supportive subroutines, this is formally stated as follows.

Proposition 2 *Let $D = (T, v_D)$ be a data stream, P be an s-stratified program, and $W = \{w_\tau^1, \ldots, w_\tau^n\}$ be time windows. Let $t, t' \in T$ s.t. $t' < t$ and suppose $\mathcal{M}$ reflects some $I \in \mathcal{AS}(P, D|_{t'}, t')$, where $D|_{t'}$ is the restriction of D up to t'. Then, if AnswerUpdate$(t, D, \mathcal{M})$ does not return fail, it updates $\mathcal{M}$ such that for some answer stream $(T, v) \in \mathcal{AS}(P, D, t)$, every $t'' \in T$ satisfies $v(t'') = v_D(t'') \cup \{a \in \mathcal{A}^{\mathcal{I}} \mid s(a) = in \wedge t'' \in tm(a)\}$.*

6 Discussion

The above TMS extension amounts to the (corrected) TMS procedure if (i) no windows and temporal operators are used, or (ii) all windows are time-based and select only the current time point, since this essentially eliminates the timeline.

Runtime complexity. The runtime of AnswerUpdate depends on the program size $|P|$ (number of rules), the number n of strata, the number of extended atoms $|\mathcal{A}^+|$, the stream size, and the specific window functions. For the practically important time- and tuple-based windows, status labels can be maintained efficiently. In the worst-case, UpdateTimestamps runs in $\mathcal{O}(|P|^2 \cdot |\mathcal{A}^+|)$ time but can be made sub-quadratic in $|P|$ using suitable data structures to store the relation between atoms, window atoms, rules, etc. Other subroutines are less costly. Window sizes are small and thus a constant factor.

Head-@-free programs. The useful @-atoms in rule heads complicate evaluation, as intuitively intentional streams must be handled at each stratum. Excluding head occurrences of @ yields a fragment with simple and faster algorithms, where intensional facts are only derivable at the query time. Such head-@-free programs P can be split into a lower part P_l with acyclic rules using windows on extensional facts, and an upper part P_u consisting of ordinary rules; by using ExpireInput and FireInput, the rules of P_l can be evaluated in a deterministic way, and those of P_u by using either an ASP solver or the incremental TMS extension to carry over the time labels.

Completeness. Algorithm AnswerUpdate might fail, even if some answer stream for P wrt. D at the query time t exists. To gain completeness, one needs to use proper backtracking in case an inconsistency (a conflict) is encountered. The backtracking technique in TMS can be extended for this purpose, by finding a 'nogood' assignment that leads to the conflict and going for a different guess from there. Notably, if P has no cyclic negation, no backtracking is needed. Furthermore, one can similarly employ extended backtracking to compute other answer streams. However, this is beyond the scope of this paper and subject for future work.

7 Related Work and Conclusion

In ASP, incremental evaluation as in iclingo (see `potassco.sourceforge.net`) aims at time slicing; reactive ASP as in oclingo [Gebser *et al.*, 2012] considers k-expanded logic programs, which are incrementally evaluated over a single sliding window of size k by incorporating always the next time point. In contrast, we build an answer stream at time t by *adapting* one from $t' < t$. Furthermore, the more expressive language LARS makes temporal data management much more demanding.

ETALIS [Anicic *et al.*, 2012] is a monotonic rule formalism for complex event processing to reason about intervals. No incremental evaluation or windows are considered.

StreamLog [Zaniolo, 2012] extends Datalog for stream reasoning based on XY-stratification, which guarantees a single model on a stream. As in s-stratified programs, rules concluding about the past are excluded. However, neither windows nor incremental evaluation have been considered.

The DRed algorithm [Gupta *et al.*, 1993] for incremental Datalog update deletes all consequences of deleted facts and then adds all rederivable ones from the remainder. It was adapted to RDF streams in [Barbieri *et al.*, 2010], where tuples are tagged with an expiration time. Our more expressive TMS approach makes time management more difficult.

[Ren and Pan, 2011] explored the use of TMS techniques for incremental update in ontology streams. However, windows and time reference were not considered, thus no temporal management is needed, and the setting is monotonic. Model update was also considered in the context of CTL. In [Zhang and Ding, 2008], minimal change criteria were identified and an update algorithm was presented.

Related to our work is also belief revision, which deals with changing a *belief set* [Alchourrón *et al.*, 1985] or *knowledge base* [Katsuno and Mendelzon, 1991] to accommodate new information without introducing inconsistency. In [Dixon and Foo, 1993] it was shown how Assumption-based Truth Maintenance Systems (ATMS) can be simulated in the AGM logic of belief [Gärdenfors, 1988] by means of an epistemic entrenchment relation (i.e., a total pre-order over sentences) to encode justifications of beliefs. Belief revision and belief update have also been studied in relation to logic programs under the answer set semantics in [Delgrande *et al.*, 2008] and [Slota and Leite, 2014].

Outlook. We have presented a generic algorithm for incremental answer update of logic programs for stream reasoning with ASP like semantics, based on stream stratification and an extension of TMS techniques by temporal data management. Further issues concern instantiation of the algorithm for typical window functions and optimization, as well as identifying program classes amenable to highly efficient answer update.

References

[Alchourrón *et al.*, 1985] Carlos E. Alchourrón, Peter Gärdenfors, and David Makinson. On the logic of theory change: Partial meet contraction and revision functions. *J. Symb. Log.*, 50(2):510–530, 1985.

[Anicic *et al.*, 2012] Darko Anicic, Sebastian Rudolph, Paul Fodor, and Nenad Stojanovic. Stream reasoning and complex event processing in ETALIS. *Semantic Web J.*, 2012.

[Apt *et al.*, 1988] Krzysztof R. Apt, Howard A. Blair, and Adrian Walker. Towards a Theory of Declarative Knowledge. In Jack Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 89–148. Morgan Kaufmann Publishers, Inc., Washington DC, 1988.

[Babu and Widom, 2001] Shivnath Babu and Jennifer Widom. Continuous queries over data streams. *SIGMOD Record*, 3(30):109–120, 2001.

[Barbieri *et al.*, 2010] Davide Francesco Barbieri, Daniele Braga, Stefano Ceri, Emanuele Della Valle, and Michael Grossniklaus. Incremental reasoning on streams and rich background knowledge. In *ESWC 2010*, pages 1–15, 2010.

[Beck *et al.*, 2015] Harald Beck, Minh Dao-Tran, Thomas Eiter, and Michael Fink. LARS: A logic-based framework for analyzing reasoning over streams. In *AAAI*, 2015.

[Beierle and Kern-Isberner, 2000] Christoph Beierle and Gabriele Kern-Isberner. Nichtmonotones Schließen I - Truth Maintenance-Systeme. In *Methoden wissensbasierter Systeme*, Computational Intelligence, pages 196–229. Vieweg+Teubner Verlag, 2000.

[Delgrande *et al.*, 2008] James P. Delgrande, Torsten Schaub, Hans Tompits, and Stefan Woltran. Belief revision of logic programs under answer set semantics. *KR*, pages 411–421, 2008.

[Della Valle *et al.*, 2009] Emanuele Della Valle, Stefano Ceri, Frank van Harmelen, and Dieter Fensel. It's a Streaming World! Reasoning upon Rapidly Changing Information. *IEEE Intelligent Systems*, 24:83–89, 2009.

[Dixon and Foo, 1993] Simon Dixon and Norman Y. Foo. Connections between the ATMS and AGM belief revision. *IJCAI*, pages 534–539, 1993.

[Doyle, 1979] Jon Doyle. A Truth Maintenance System. *Artif. Intell.*, 12(3):231–272, 1979.

[Elkan, 1990] Charles Elkan. A rational reconstruction of nonmonotonic truth maintenance systems. *Artif. Intell.*, 43(2):219–234, 1990.

[Gärdenfors, 1988] Peter Gärdenfors. *Knowledge in flux: modeling the dynamics of epistemic states*. MIT Press Cambridge, Mass., 1988.

[Gebser *et al.*, 2012] Martin Gebser, Torsten Grote, Roland Kaminski, Philipp Obermeier, Orkunt Sabuncu, and Torsten Schaub. Stream reasoning with answer set programming. Preliminary report. In *KR*, pages 613–617, 2012.

[Gupta *et al.*, 1993] Ashish Gupta, Inderpal Singh Mumick, and V. S. Subrahmanian. Maintaining views incrementally. *ACM SIGMOD International Conference on Management of Data*, pages 157–166, 1993.

[Katsuno and Mendelzon, 1991] Hirofumi Katsuno and Alberto O. Mendelzon. On the difference between updating a knowledge base and revising it. *KR*, pages 387–394, 1991.

[Ren and Pan, 2011] Yuan Ren and Jeff Z. Pan. Optimising ontology stream reasoning with truth maintenance system. In *CIKM*, pages 831–836, 2011.

[Rich and Knight, 1991] Elaine Rich and Kevin Knight. Artificial intelligence. *McGraw-Hill*, 1991.

[Slota and Leite, 2014] Martin Slota and João Leite. The rise and fall of semantic rule updates based on se-models. *TPLP*, 14(6):869–907, 2014.

[Zaniolo, 2012] Carlo Zaniolo. Logical foundations of continuous query languages for data streams. In *Datalog*, pages 177–189, 2012.

[Zhang and Ding, 2008] Yan Zhang and Yulin Ding. CTL model update for system modifications. *J. Artif. Intell. Res. (JAIR)*, 31:113–155, 2008.